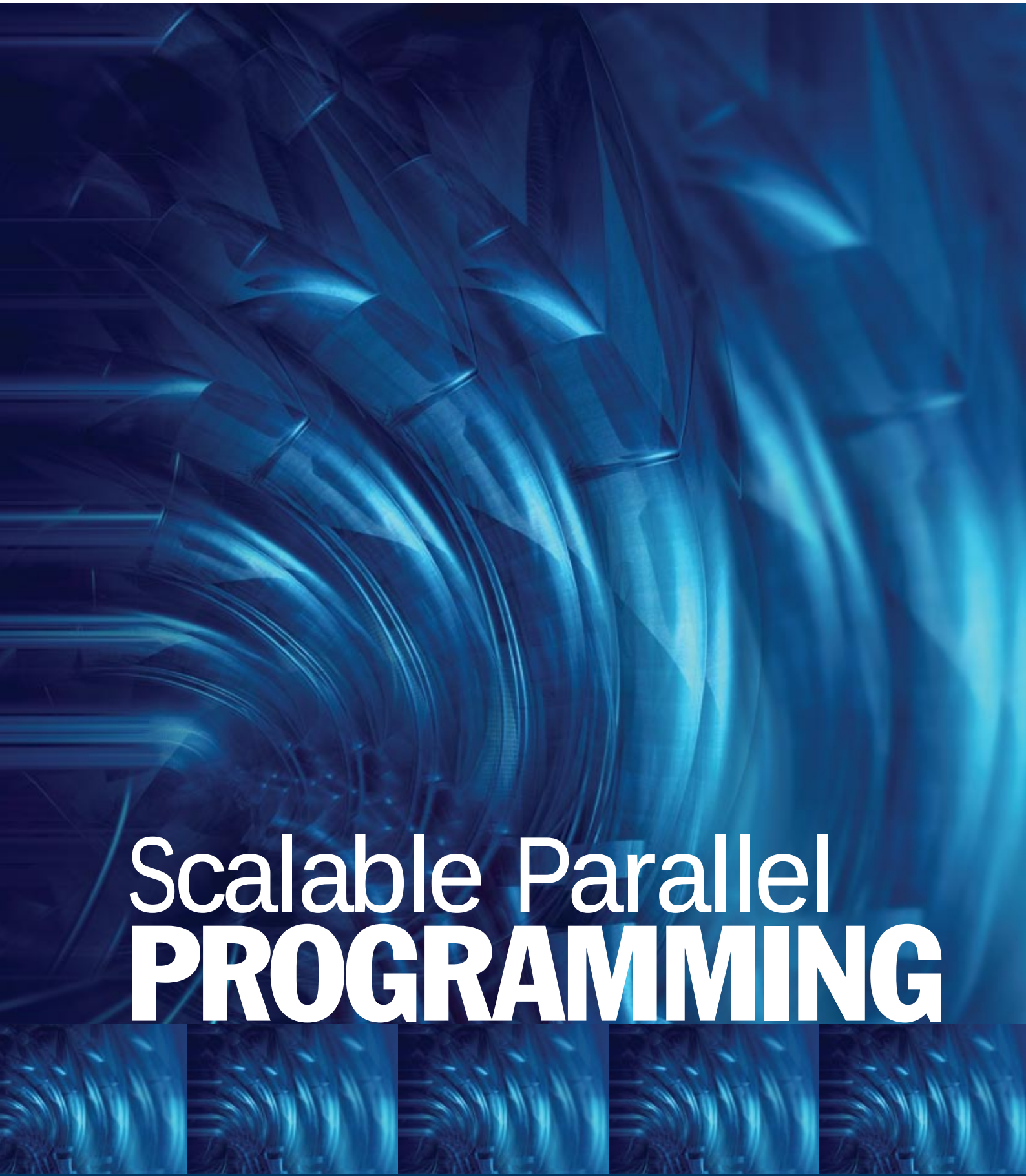
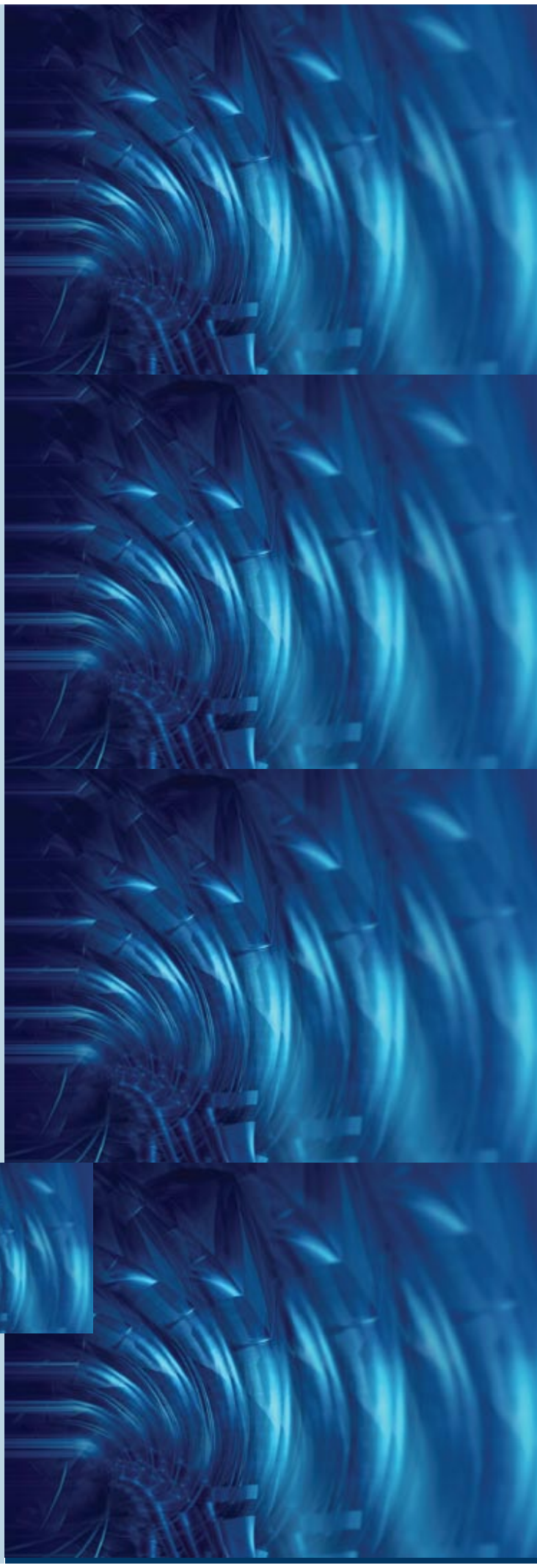
Contents



Scalable Parallel PROGRAMMING

JOHN NICKOLLS, IAN BUCK, AND
MICHAEL GARLAND, NVIDIA,
KEVIN SKADRON, UNIVERSITY OF VIRGINIA



The advent of multicore CPUs and manycore GPUs means that mainstream processor chips are now parallel systems. Furthermore, their parallelism continues to scale with Moore's law. The challenge is to develop mainstream application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to manycore GPUs with widely varying numbers of cores.

According to conventional wisdom, parallel programming is difficult. Early experience with the CUDA[1,2] scalable parallel programming model and C language, however, shows that many sophisticated programs can be readily expressed with a few easily understood abstractions. Since NVIDIA released CUDA in 2007, developers have rapidly developed scalable parallel programs for a wide range of applications, including computational chemistry, sparse matrix solvers, sorting, searching, and physics models. These applications scale transparently to hundreds of processor cores and thousands of concurrent threads. NVIDIA GPUs with the new Tesla unified graphics and computing architecture (described in the GPU sidebar) run CUDA C programs and are widely available in laptops, PCs, workstations, and servers. The CUDA

with CUDA

Is CUDA the parallel programming model that application developers have been waiting for?

Scalable Parallel PROGRAMMING with CUDA

model is also applicable to other shared-memory parallel processing architectures, including multicore CPUs.[3]

CUDA provides three key abstractions—a hierarchy of thread groups, shared memories, and barrier synchronization—that provide a clear parallel structure to conventional C code for one thread of the hierarchy. Multiple levels of threads, memory, and synchronization provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. The abstractions guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel, and then into

UNIFIED GRAPHICS AND COMPUTING GPUS

Driven by the insatiable market demand for realtime, high-definition 3D graphics, the programmable GPU (graphics processing unit) has evolved into a highly parallel, multithreaded, manycore processor. It is designed to efficiently support the graphics shader programming model, in which a program for one thread draws one vertex or shades one pixel fragment. The GPU excels at fine-grained, data-parallel workloads consisting of thousands of independent threads executing vertex, geometry, and pixel-shader program threads concurrently.

The tremendous raw performance of modern GPUs has led researchers to explore mapping more general non-graphics computations onto them. These GPGPU (general-purpose computation on GPUs) systems have produced some impressive results, but the limitations and difficulties of doing this via graphics APIs are legend. This desire to use the GPU as a more general parallel computing device motivated NVIDIA to develop a new unified graphics and computing GPU architecture and the CUDA programming model.

GPU COMPUTING ARCHITECTURE

Introduced by NVIDIA in November 2006, the Tesla unified graphics and computing architecture[1,2] significantly extends the GPU beyond graphics—its massively multithreaded processor array becomes a highly efficient unified platform for *both* graphics and general-purpose parallel computing applications. By scaling the number of processors and memory partitions, the Tesla architecture spans a wide market range—from the high-performance enthusiast GeForce 8800 GPU and professional Quadro and Tesla computing products to a variety of inexpensive, mainstream GeForce GPUs. Its computing features enable straightforward programming of the GPU cores in C with CUDA. Wide availability in laptops, desktops, workstations, and servers, coupled with C programmability and CUDA software, make the Tesla architecture the first ubiquitous supercomputing platform.

The Tesla architecture is built around a scalable array of multithreaded SMs (streaming multiprocessors). Current GPU implementations range from 768 to 12,288 concurrently executing threads. Transparent scaling across this wide range of available parallelism is a key design goal of both the GPU architecture and the CUDA programming model. Figure A shows a GPU with 14 SMs—a total of 112 SP (streaming processor) cores—interconnected with four external DRAM partitions. When a CUDA program on the host CPU invokes a kernel grid, the CWD (compute work distribution) unit enumerates the blocks of the grid and begins distributing them to SMs with available execution capacity. The threads of a thread block execute concurrently on one SM. As thread blocks terminate, the CWD unit launches new blocks on the vacated multiprocessors.

An SM consists of eight scalar SP cores, two SFUs (special function units) for transcendentals, an MT IU (multithreaded instruction unit), and on-chip shared memory. The SM creates, manages, and executes up to 768 concurrent threads in hardware with zero scheduling overhead. It can execute as many as eight CUDA thread blocks concurrently, limited by thread and memory resources. The SM implements the CUDA __syncthreads() barrier synchronization intrinsic with a single instruction. Fast barrier synchronization together with lightweight thread creation and zero-overhead thread scheduling efficiently support very fine-grained parallelism, allowing a new thread to be created to compute each vertex, pixel, and data point.

To manage hundreds of threads running several different programs, the Tesla SM employs a new architecture we call

finer pieces that can be solved cooperatively in parallel. The programming model scales transparently to large numbers of processor cores: a compiled CUDA program executes on any number of processors, and only the runtime system needs to know the physical processor count.

THE CUDA PARADIGM

CUDA is a minimal extension of the C and C++ programming languages. The programmer writes a serial program that calls parallel *kernels*, which may be simple functions or full programs. A kernel executes in parallel across a set of parallel threads. The programmer organizes these threads into a hierarchy of grids of thread blocks. A *thread block* is a set of concurrent threads that can cooperate among themselves through barrier synchronization and shared access to a memory space private to the block. A *grid* is a set of thread blocks that may each be executed independently and thus may execute in parallel.

When invoking a kernel, the programmer specifies the number of threads per block and the number of blocks

SIMT (single-instruction, multiple-thread).[3] The SM maps each thread to one SP scalar core, and each scalar thread executes independently with its own instruction address and register state. The SM SIMT unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. (This term originates from weaving, the first parallel thread technology.) Individual threads composing a

Continued on the next page

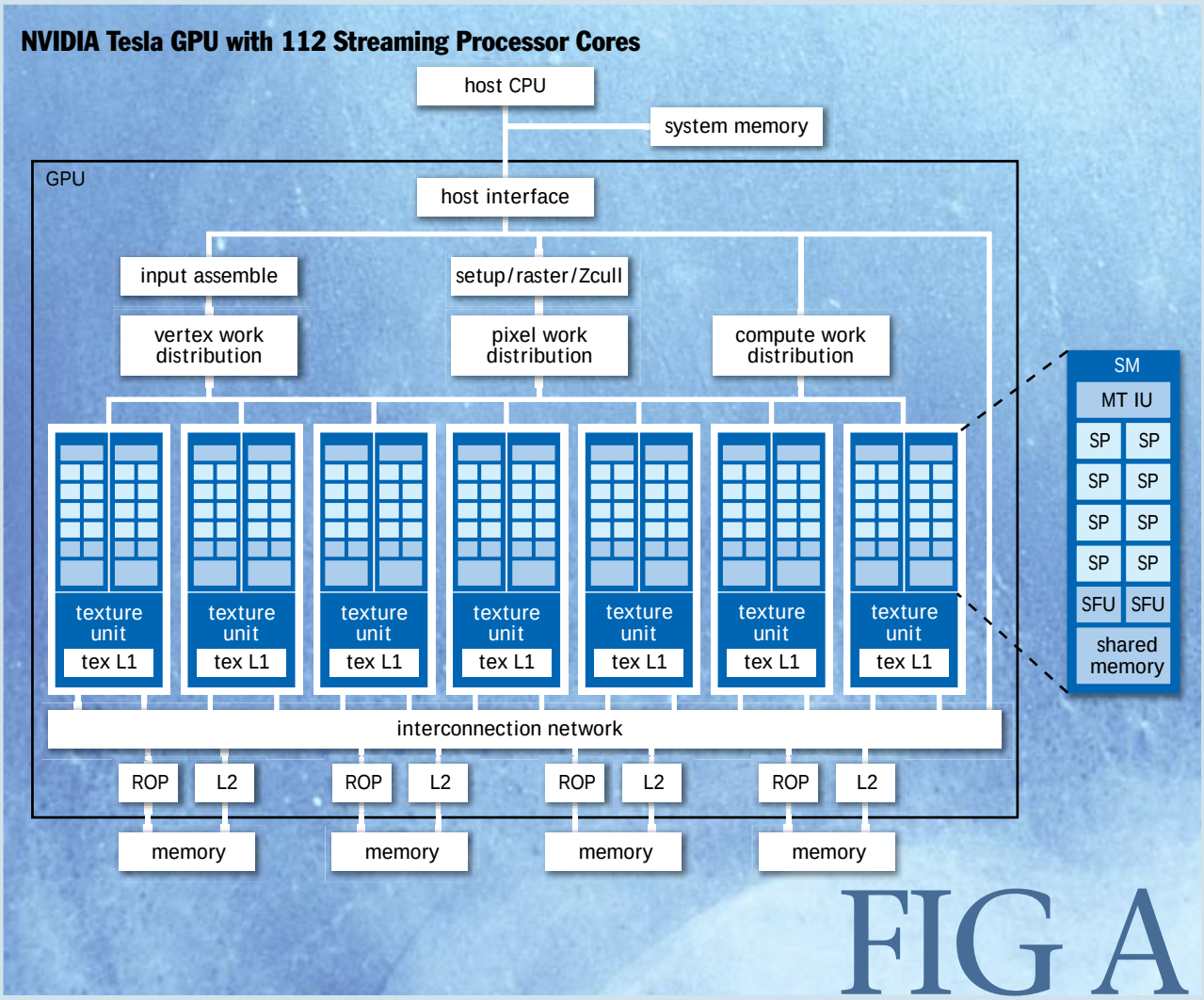


FIG A

Scalable Parallel PROGRAMMING with CUDA

making up the grid. Each thread is given a unique *thread ID* number threadIdx within its thread block, numbered 0, 1, 2, …, blockDim–1, and each thread block is given a unique *block ID* number blockIdx within its grid. CUDA supports thread blocks containing up to 512 threads. For convenience, thread blocks and grids may have one, two, or three dimensions, accessed via .x, .y, and .z index fields.

As a very simple example of parallel programming, suppose that we are given two vectors x and y of n floating-point numbers each and that we wish to compute the result of $y \leftarrow ax + y$, for some scalar value a. This is the

SIMT warp start together at the same program address but are otherwise free to branch and execute independently. Each SM manages a pool of 24 warps of 32 threads per warp, a total of 768 threads.

Every instruction issue time, the SIMT unit selects a warp that is ready to execute and issues the next instruction to the active threads of the warp. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjointed code paths. As a result, the Tesla-architecture GPUs are dramatically more efficient and flexible on branching code than previous-generation GPUs, as their 32-thread warps are much narrower than the SIMD (single-instruction multiple-data) width of prior GPUs.

SIMT architecture is akin to SIMD vector organizations in that a single instruction controls multiple processing elements. A key difference is that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread. In contrast with SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads. For the purposes of correctness, the programmer can essentially ignore the SIMT behavior; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in traditional code: cache line size can be safely ignored when designing for correctness but must be considered in the code structure when designing for peak performance. Vector architectures, on the other hand, require the software to coalesce loads into vectors and manage divergence manually.

A thread's variables typically reside in live registers. The 16KB SM shared memory has very low access latency and high bandwidth similar to an L1 cache; it holds CUDA per-block __shared__ variables for the active thread blocks. The SM provides load/store instructions to access CUDA __device__ variables in GPU external DRAM. It coalesces individual accesses of parallel threads in the same warp into fewer memory-block accesses when the addresses fall in the same block and meet alignment criteria. Because global memory latency can be hundreds of processor clocks, CUDA programs copy data to shared memory when it must be accessed multiple times by a thread block. Tesla load/store memory instructions use integer byte addressing to facilitate conventional compiler code optimizations. The large thread count in each SM, together with support for many outstanding load requests, helps to cover load-to-use latency to the external DRAM. The latest Tesla-architecture GPUs also provide atomic read-modify-write memory instructions, facilitating parallel reductions and parallel-data structure management.

CUDA applications perform well on Tesla-architecture GPUs because CUDA's parallelism, synchronization, shared memories, and hierarchy of thread groups map efficiently to features of the GPU architecture, and because CUDA expresses application parallelism well.

REFERENCES

1. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro* 28(2).
2. Nickolls, J. 2007. NVIDIA GPU parallel computing architecture. In *IEEE Hot Chips* 19 (August 20), Stanford, CA; http://www.hotchips.org/archives/hc19/.
3. See reference 1.

so-called saxpy kernel defined by the BLAS (basic linear algebra subprograms) library. The code for performing this computation on both a serial processor and in parallel using CUDA is shown in figure 1.

The __global__ declaration specifier indicates that the procedure is a kernel entry point. CUDA programs launch parallel kernels with the extended function-call syntax

```
kernel<<<dimGrid, dimBlock>>>(... parameter list ...);
```

where dimGrid and dimBlock are three-element vectors of type dim3 that specify the dimensions of the grid in blocks and the dimensions of the blocks in threads, respectively. Unspecified dimensions default to 1.

In the example, we launch a grid that assigns one thread to each element of the vectors and puts 256 threads in each block. Each thread computes an element index from its thread and block IDs and then performs the desired calculation on the corresponding vector elements. The serial and parallel versions of this code are strikingly similar. This represents a fairly common pattern. The serial code consists of a loop where each iteration is independent of all the others. Such loops can be mechanically transformed into parallel kernels: each loop iteration becomes an independent thread. By assigning a single thread to each output element, we avoid the need for any synchronization among threads when writing results to memory.

Computing y ← ax + y with a Serial Loop

```
void saxpy_serial(int n, float alpha, float *x, float *y)
{
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}

// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Computing y ← ax + y in parallel using CUDA

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;

    if( i<n )  y[i] = alpha*x[i] + y[i];
}

// Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

FIG 1

The text of a CUDA kernel is simply a C function for one sequential thread. Thus, it is generally straightforward to write and is typically simpler than writing parallel code for vector operations. Parallelism is determined clearly and explicitly by specifying the dimensions of a grid and its thread blocks when launching a kernel.

Parallel execution and thread management are automatic. All thread creation, scheduling, and termination are handled for the programmer by the underlying system. Indeed, a Tesla-architecture GPU performs all thread management directly in hardware. The threads of a block execute concurrently and may synchronize at a barrier by calling the __syncthreads() intrinsic. This guarantees that no thread participating in the barrier can proceed until all participating threads have reached the barrier. After passing the barrier, these threads are also guaranteed to see all writes to memory performed by participating threads before the barrier. Thus, threads in a block may communicate with each other by writing and reading per-block shared memory at a synchronization barrier.

Since threads in a block may share local memory and synchronize via barriers, they will reside on the same physical processor or multiprocessor. The number of thread blocks can, however, greatly exceed the number of processors. This virtualizes the processing elements and gives the programmer the flexibility to parallelize at whatever granularity is most convenient. This allows intuitive problem decompositions, as the number of blocks can be dictated by the size of the data being processed rather than by the number of processors in the system. This also allows the same CUDA program to scale to widely varying numbers of processor cores.

To manage this processing element virtualization and provide scalability, CUDA requires that thread blocks execute independently. It must be possible to execute blocks in any order, in parallel or in series. Different blocks have no means of direct communication, although they may *coordinate* their activities using atomic memory operations on the global memory visible to all threads—by atomically incrementing queue pointers, for example.

This independence requirement allows thread blocks to be scheduled in any order across any number of cores, making the CUDA model scalable across an arbitrary

Scalable Parallel PROGRAMMING with CUDA

number of cores, as well as across a variety of parallel architectures. It also helps to avoid the possibility of deadlock.

An application may execute multiple grids either independently or dependently. Independent grids may execute concurrently given sufficient hardware resources. Dependent grids execute sequentially, with an implicit inter-kernel barrier between them, thus guaranteeing that all blocks of the first grid will complete before any block of the second dependent grid is launched.

Threads may access data from multiple memory spaces during their execution. Each thread has a private *local* memory. CUDA uses this memory for thread-private variables that do not fit in the thread's registers, as well as for stack frames and register spilling. Each thread block has a *shared* memory visible to all threads of the block that has the same lifetime as the block. Finally, all threads have access to the same *global* memory. Programs declare variables in shared and global memory with the __shared__ and __device__ type qualifiers. On a Tesla-architecture GPU, these memory spaces correspond to physically separate memories: per-block shared memory is a low-latency on-chip RAM, while global memory resides in the fast DRAM on the graphics board.

Shared memory is expected to be a low-latency memory near each processor, much like an L1 cache. It can, therefore, provide for high-performance communication and data sharing among the threads of a thread block. Since it has the same lifetime as its corresponding thread block, kernel code will typically initialize data in shared variables, compute using shared variables, and copy shared memory results to global memory. Thread blocks of sequentially dependent grids communicate via global memory, using it to read input and write results.

Figure 2 diagrams the nested levels of threads, thread blocks, and grids of thread blocks. It shows the corresponding levels of memory sharing: local, shared, and global memories for per-thread, per-thread-block, and per-application data sharing.

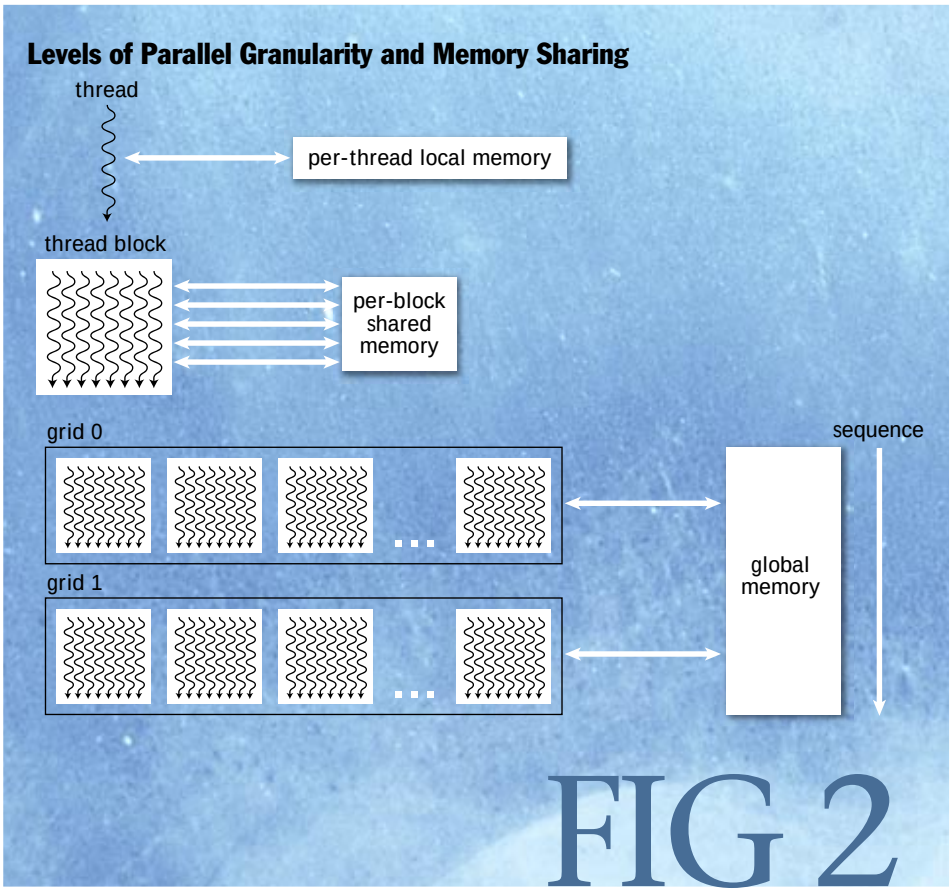


FIG 2

A program manages the global memory space visible to kernels through calls to the CUDA runtime, such as cudaMalloc() and cudaFree(). Kernels may execute on a physically separate device, as is the case when running kernels on the GPU. Consequently, the application must use cudaMemcpy() to copy data between the allocated space and the host system memory.

The CUDA programming model is similar in style to the familiar SPMD (single-program multiple-data) model—it expresses parallelism explicitly, and each kernel executes on a fixed number of threads. CUDA, however, is more flexible than most real-

izations of SPMD, because each kernel call dynamically creates a new grid with the right number of thread blocks and threads for that application step. The programmer can use a convenient degree of parallelism for each kernel, rather than having to design all phases of the computation to use the same number of threads.

Figure 3 shows an example of a SPMD-like CUDA code sequence. It first instantiates kernelF on a 2D grid of 3×2 blocks where each 2D thread block consists of 5×3 threads. It then instantiates kernelG on a 1D grid of four 1D thread blocks with six threads each. Because kernelG depends on the results of kernelF, they are separated by an inter-kernel synchronization barrier.

The concurrent threads of a thread block express fine-grained *data* and *thread* parallelism. The independent thread blocks of a grid express coarse-grained *data* parallelism. Independent grids express coarse-grained *task* parallelism. A *kernel* is simply C code for one thread of the hierarchy.

RESTRICTIONS

When developing CUDA programs, it is important to understand the ways in which the CUDA model is restricted, largely for reasons of efficiency. Threads and thread blocks may be created only by invoking a parallel kernel, not from within a parallel kernel. Together with the required independence of thread blocks, this makes it possible to execute CUDA programs with a simple scheduler that introduces minimal runtime overhead. In fact, the Tesla architecture implements *hardware* management and scheduling of threads and thread blocks.

Task parallelism can be expressed at the thread-block level, but blockwide barriers are not well suited for supporting task parallelism among threads in a block. To enable CUDA programs to run on any number of processors, communication between thread blocks within the same kernel grid is not allowed—they must execute independently. Since CUDA requires that thread blocks be independent and allows blocks to be executed in any

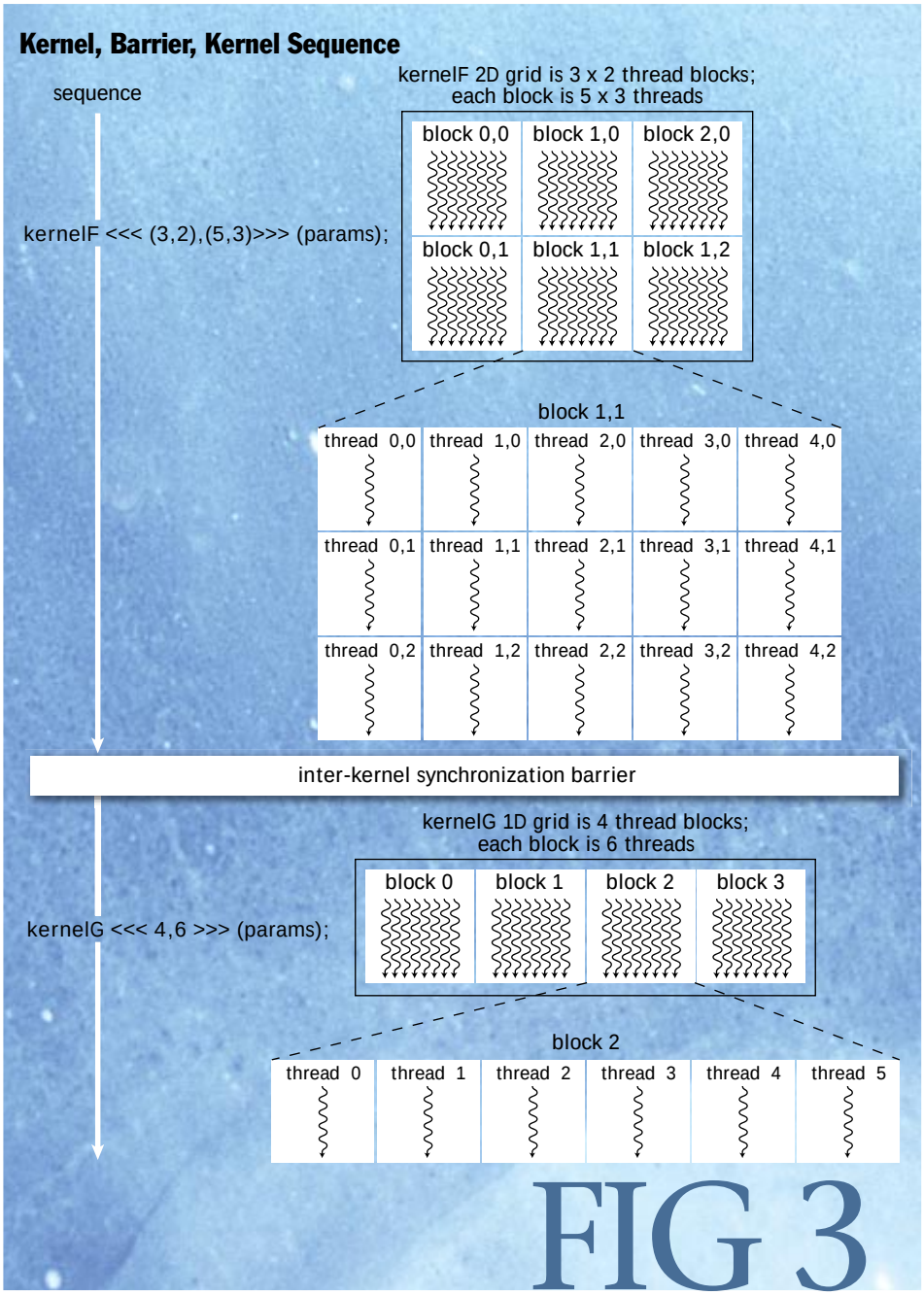


FIG 3

Scalable Parallel PROGRAMMING with CUDA

order, combining results generated by multiple blocks must in general be done by launching a second kernel on a new grid of thread blocks. However, multiple thread blocks can coordinate their work using atomic operations on global memory (e.g., to manage a data structure).

Recursive function calls are not allowed in CUDA kernels. Recursion is unattractive in a massively parallel kernel because providing stack space for the tens of thousands of threads that may be active would require substantial amounts of memory. Serial algorithms that are normally expressed using recursion, such as quicksort, are typically best implemented using nested data parallelism rather than explicit recursion.

To support a heterogeneous system architecture combining a CPU and a GPU, each with its own memory system, CUDA programs must copy data and results between host memory and device memory. The overhead of CPU–GPU interaction and data transfers is minimized by using DMA block-transfer engines and fast interconnects. Of course, problems large enough to need a GPU performance boost amortize the overhead better than small problems.

RELATED WORK

Although the first CUDA implementation targets NVIDIA GPUs, the CUDA abstractions are general and useful for programming multicore CPUs and scalable parallel systems. Coarse-grained thread blocks map naturally to separate processor cores, while fine-grained threads map to multiple-thread contexts, vector operations, and pipelined loops in each core. Stratton et al. have developed a prototype source-to-source translation framework that compiles CUDA programs for multicore CPUs by mapping a thread block to loops within a single CPU thread. They found that CUDA kernels compiled in this way perform and scale well.[4]

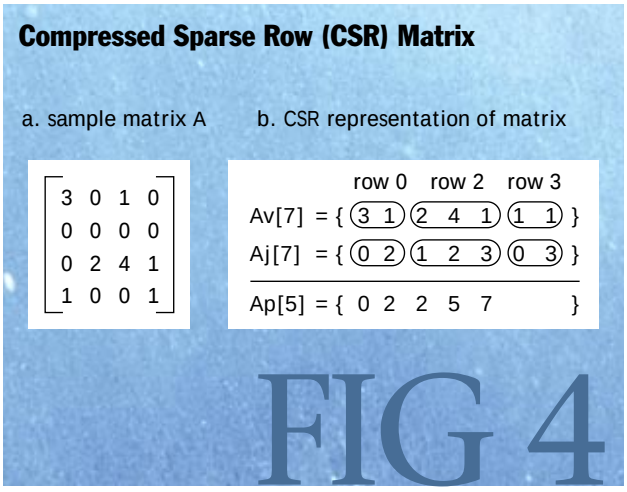
CUDA uses parallel kernels similar to recent GPGPU programming models, but differs by providing flexible thread creation, thread blocks, shared memory, global memory, and explicit synchronization. Streaming languages apply parallel kernels to data records from a stream. Applying a stream kernel to one record is analogous to executing a single CUDA kernel thread, but stream programs do not allow dependencies among kernel threads, and kernels communicate only via FIFO (first-in, first-out) streams. Brook for GPUs differentiates between FIFO input/output streams and random-access *gather* streams, and it supports parallel reductions. Brook is a good fit for earlier-generation GPUs with random access texture units and raster pixel operation units.[5]

Pthreads and Java provide fork-join parallelism but are not particularly convenient for data-parallel applications. OpenMP targets shared memory architectures with parallel execution constructs, including "parallel for" and teams of coarse-grained threads. Intel's C++ Threading Building Blocks provide similar features for multicore CPUs. MPI targets distributed memory systems and uses message passing rather than shared memory.

CUDA APPLICATION EXPERIENCE

The CUDA programming model extends the C language with a small number of additional parallel abstractions. Programmers who are comfortable developing in C can quickly begin writing CUDA programs.

In the relatively short period since the introduction of CUDA, a number of real-world parallel application codes have been developed using the CUDA model. These include FHD-spiral MRI reconstruction,[6] molecular dynamics,[7] and n-body astrophysics simulation.[8] Running on Tesla-architecture GPUs, these applications were able to achieve substantial speedups over alternative implementations running on serial CPUs: the MRI reconstruction was 263 times faster; the molecular dynamics code was 10–100 times faster; and the n-body simulation was 50–250 times faster. These large speedups are a result of the highly parallel nature of the Tesla architecture and its high memory bandwidth.



EXAMPLE: SPARSE MATRIX-VECTOR PRODUCT

A variety of parallel algorithms can be written in CUDA in a fairly straightforward manner, even when the data structures involved are not simple regular grids. SpMV (sparse matrix-vector multiplication) is a good example of an important numerical building block that can be parallelized quite directly using the abstractions provided by CUDA. The kernels we discuss here, when combined with the provided CUBLAS vector routines, make writing iterative solvers such as the conjugate gradient[9] method straightforward.

A sparse $n \times n$ matrix is one in which the number of nonzero entries m is only a small fraction of the total. Sparse matrix representations seek to store only the nonzero elements of a matrix. Since it is fairly typical that a sparse $n \times n$ matrix will contain only m=O(n) nonzero elements, this represents a substantial savings in storage space and processing time.

One of the most common representations for general unstructured sparse matrices is the CSR (compressed sparse row) representation. The m nonzero elements of the matrix A are stored in row-major order in an array Av. A second array Aj records the corresponding column index for each entry of Av. Finally, an array Ap of n+1

```
float multiply_row(unsigned int rowsize,
                   unsigned int *Aj, // column indices for row
                   float *Av,        // non-zero entries for row
                   float *x)         // the RHS vector
{
   float sum = 0;

   for(unsigned int column=0; column<rowsize; ++column)
       sum += Av[column] * x[Aj[column]];

   return sum;
}
```

FIG 5

```
void csrmul_serial(unsigned int *Ap, unsigned int *Aj,
                   float *Av, unsigned int num_rows,
                   float *x, float *y)
{
   for(unsigned int row=0; row<num_rows; ++row)
   {
       unsigned int row_begin = Ap[row];
       unsigned int row_end   = Ap[row+1];

       y[row] = multiply_row(row_end-row_begin, Aj+row_begin,
                             Av+row_begin, x);
   }
}
```

FIG 6

```
__global__
void csrmul_kernel(unsigned int *Ap, unsigned int *Aj,
                   float *Av, unsigned int num_rows,
                   float *x, float *y)
{
   unsigned int row = blockIdx.x*blockDim.x + threadIdx.x;

   if( row<num_rows )
   {
      unsigned int row_begin = Ap[row];
      unsigned int row_end   = Ap[row+1];

      y[row] = multiply_row(row_end-row_begin, Aj+row_begin,
                            Av+row_begin, x);
   }
}
```

FIG 7

Scalable Parallel PROGRAMMING with CUDA

elements records the extent of each row in the previous arrays; the entries for row *i* in Aj and Av extend from index Ap[i] up to, but not including, index Ap[i+1]. This implies that Ap[0] will always be 0 and Ap[n] will always be the number of nonzero elements in the matrix. Figure 4 shows an example of the CSR representation of a simple matrix.

Given a matrix *A* in CSR form, we can compute a single row of the product $y = Ax$ using the multiply_row() procedure shown in figure 5.

Computing the full product is then simply a matter of looping over all rows and computing the result for that row using multiply_row(), as shown in figure 6.

This algorithm can be translated into a parallel CUDA kernel quite easily. We simply spread the loop in csrmul_serial() over many parallel threads. Each thread will compute exactly one row of the output vector *y*. Figure 7 shows the code for this kernel. Note that it looks extremely similar to the serial loop used in the csrmul_serial() procedure. There are really only two points of difference. First, the row index is computed from the block and thread indices assigned to each thread. Second, we have a conditional that evaluates a row product only if the row index is within the bounds of the matrix (this is necessary since the number of rows *n* need not be a multiple of the block size used in launching the kernel).

Assuming that the matrix data structures have already been copied to the GPU device memory, launching this kernel will look like the code in figure 8.

The pattern that we see here is a common one. The original serial algorithm is a loop whose iterations are independent of each other. Such loops can be parallelized quite easily by simply assigning one or more iterations of the loop to each parallel thread. The programming model provided by CUDA makes expressing this type of parallelism particularly straightforward.

This general strategy of decomposing computations into blocks of independent work, and more specifically breaking up independent loop iterations, is not unique to CUDA. This is a common approach used in one form or another by various parallel programming systems, including OpenMP and Intel's Threading Building Blocks.

```
unsigned int blocksize = 128;  // or any size up to 512
unsigned int nblocks   = (num_rows + blocksize - 1) / blocksize;
csrmul_kernel<<<nblocks,blocksize>>>(Ap, Aj, Av, num_rows, x, y);
```

FIG 8

```
__global__
void csrmul_cached(unsigned int *Ap, unsigned int *Aj,
                   float *Av, unsigned int num_rows,
                   const float *x, float *y)
{
   // Cache the rows of x[] corresponding to this block.
   __shared__ float cache[blocksize];

   unsigned int block_begin = blockIdx.x * blockDim.x;
   unsigned int block_end   = block_begin + blockDim.x;
   unsigned int row         = block_begin + threadIdx.x;

   // Fetch and cache our window of x[].
   if( row<num_rows)  cache[threadIdx.x] = x[row];
   __syncthreads();

   if( row<num_rows )
   {
      unsigned int row_begin = Ap[row];
      unsigned int row_end   = Ap[row+1];
      float sum = 0, x_j;

      for(unsigned int col=row_begin; col<row_end; ++col)
      {
          unsigned int j = Aj[col];

          // Fetch x_j from our cache when possible
          if( j>=block_begin && j<block_end )
            x_j = cache[j-block_begin];
          else
            x_j = x[j];

          sum += Av[col] * x_j;
      }

      y[row] = sum;
   }
}
```

FIG 9

CACHING IN SHARED MEMORY

The SpMV algorithms outlined here are fairly simplistic. We can make a number of optimizations in both the CPU and GPU codes that can improve performance, including loop unrolling, matrix reordering, and register blocking.[10] The parallel kernels can also be reimplemented in terms of data-parallel *scan* operations.[11]

One of the important architectural features exposed by CUDA is the presence of the per-block shared memory, a small on-chip memory with very low latency. Taking advantage of this memory can deliver substantial performance improvements. One common way of doing this is to use shared memory as a software-managed cache to hold frequently reused data, shown in figure 9.

In the context of sparse matrix multiplication, we observe that several rows of *A* may use a particular array element x[i]. In many common cases, and particularly when the matrix has been reordered, the rows using x[i] will be rows near row *i*. We can therefore implement a simple caching scheme and expect to achieve some performance benefit. The block of threads processing rows *i* through *j* will load x[i] through x[j] into its shared memory. We will unroll the multiply_row() loop and fetch elements of x from the cache whenever possible. The resulting code is shown in figure 9. Shared memory can also be used to make other optimizations, such as fetching Ap[row+1] from an adjacent thread rather than refetching it from memory.

Because the Tesla architecture provides an explicitly managed on-chip shared memory rather than an implicitly active hardware cache, it is fairly common to add this sort of optimization. Although this can impose some additional development burden on the programmer, it is relatively minor, and the potential performance benefits can be substantial. In the example shown in figure 9, even this fairly simple use of shared memory returns a roughly 20 percent performance improvement on representative matrices derived from 3D surface meshes. The availability of an explicitly managed memory in lieu of an implicit cache also has the advantage that caching and prefetching policies can be specifically tailored to the application needs.

EXAMPLE: PARALLEL REDUCTION

Suppose that we are given a sequence of *N* integers that must be combined in some fashion (e.g., a sum). This occurs in a variety of algorithms, linear algebra being a common example. On a serial processor, we would write a simple loop with a single accumulator variable to construct the sum of all elements in sequence. On a parallel machine, using a single accumulator variable would create a global serialization point and lead to very poor performance. A well-known solution to this problem is the so-called *parallel reduction* algorithm. Each parallel thread sums a fixed-length subsequence of the input. We then collect these partial sums together, by summing

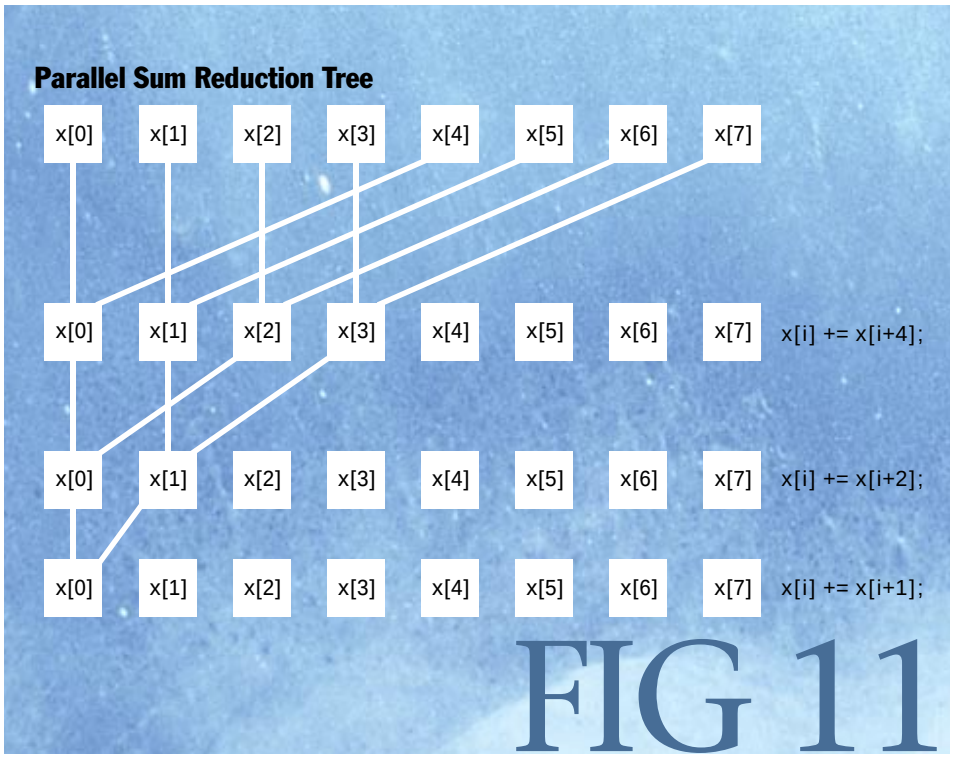
```
__global__
void plus_reduce(int *input, unsigned int N, int *total)
{
   unsigned int tid = threadIdx.x;
   unsigned int i   = blockIdx.x*blockDim.x + threadIdx.x;

   // Each block loads its elements into shared memory, padding
   // with 0 if N is not a multiple of blocksize
   __shared__ int x[blocksize];
   x[tid] = (i<N) ? input[i] : 0;
   __syncthreads();

   // Every thread now holds 1 input value in x[]
   //
   // Build summation tree over elements. See attached figure.
   for(int s=blockDim.x/2; s>0; s=s/2)
   {
       if(tid < s)  x[tid] += x[tid + s];
       __syncthreads();
   }

   // Thread 0 now holds the sum of all input values
   // to this block. Have it add that sum to the running total
   if( tid == 0 )  atomicAdd(total, x[tid]);
}
```

FIG 10

Scalable Parallel PROGRAMMING with CUDA

pairs of partial sums in parallel. Each step of this pair-wise summation cuts the number of partial sums in half and ultimately produces the final sum after $\log_2 N$ steps. Note that this implicitly builds a tree structure over the initial partial sums.

In the example shown in figure 10, each thread simply loads one element of the input sequence (i.e., it initially sums a subsequence of length one). At the end of the reduction, we want thread 0 to hold the sum of all elements initially loaded by the threads of its block. We can achieve this in parallel by summing values in a tree-like pattern. The loop in this kernel implicitly builds a summation tree over the input elements. The action of this loop for the simple case of a block of eight threads is illustrated in figure 11. The steps of the loop are shown as successive levels of the diagram and edges indicate from where partial sums are being read.



At the end of this loop, thread 0 holds the sum of all the values loaded by this block. If we want the final value of the location pointed to by `total` to contain the total of all elements in the array, we must combine the partial sums of all the blocks in the grid. One strategy would be to have each block write its partial sum into a second array and then launch the reduction kernel again, repeating the process until we had reduced the sequence to a single value. A more attractive alternative supported by the Tesla architecture is to use `atomicAdd()`, an efficient atomic read-modify-write primitive supported by the memory subsystem. This eliminates the need for additional temporary arrays and repeated kernel launches.

Parallel reduction is an essential primitive for parallel programming and highlights the importance of per-block shared memory and low-cost barriers in making cooperation among threads efficient. This degree of data shuffling among threads would be prohibitively expensive if done in off-chip global memory.

THE DEMOCRATIZATION OF PARALLEL PROGRAMMING

CUDA is a model for parallel programming that provides a few easily understood abstractions that allow the programmer to focus on algorithmic efficiency and develop scalable parallel applications. In fact, CUDA is an excellent programming environment for teaching parallel programming. The University of Virginia has used it as just a short, three-week module in an undergraduate computer architecture course, and students were able to write a correct k-means clustering program after just three lectures. The University of Illinois has successfully taught a semester-long parallel programming course using CUDA to a mix of computer science and non-computer science majors, with students obtaining impressive speedups on a variety of real applications, including the previously mentioned MRI reconstruction example.

CUDA is supported on NVIDIA GPUs with the Tesla unified graphics and computing architecture of the GeForce 8-series, recent Quadro, Tesla, and future GPUs.

The programming paradigm provided by CUDA has allowed developers to harness the power of these scalable parallel processors with relative ease, enabling them to achieve speedups of 100 times or more on a variety of sophisticated applications.

The CUDA abstractions, however, are general and provide an excellent programming environment for multicore CPU chips. A prototype source-to-source translation framework developed at the University of Illinois compiles CUDA programs for multicore CPUs by mapping a parallel thread block to loops within a single physical thread. CUDA kernels compiled in this way exhibit excellent performance and scalability.[12]

Although CUDA was released less than a year ago, it is already the target of massive development activity—there are tens of thousands of CUDA developers. The combination of massive speedups, an intuitive programming environment, and affordable, ubiquitous hardware is rare in today's market. In short, CUDA represents a democratization of parallel programming. Q

REFERENCES

1. NVIDIA. 2007. CUDA Technology; http://www.nvidia.com/CUDA.
2. NVIDIA. 2007. CUDA Programming Guide 1.1; http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf.
3. Stratton, J.A., Stone, S. S., Hwu, W. W. 2008. M-CUDA: An efficient implementation of CUDA kernels on multicores. IMPACT Technical Report 08-01, University of Illinois at Urbana-Champaign, (February).
4. See reference 3.
5. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P. Brook for GPUs: Stream computing on graphics hardware. 2004. *Proceedings of SIGGRAPH* (August): 777-786; http://doi.acm.org/10.1145/1186562.1015800.
6. Stone, S.S., Yi, H., Hwu, W.W., Haldar, J.P., Sutton, B.P., Liang, Z.-P. 2007. How GPUs can improve the quality of magnetic resonance imaging. The First Workshop on General-Purpose Processing on Graphics Processing Units (October).
7. Stone, J.E., Phillips, J.C., Freddolino, P.L., Hardy, D.J., Trabuco, L.G., Schulten, K. 2007. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry* 28(16): 2618–2640; http://dx.doi.org/10.1002/jcc.20829.
8. Nyland, L., Harris, M., Prins, J. 2007. Fast n-body simulation with CUDA. In *GPU Gems 3*. H. Nguyen, ed. Addison-Wesley.
9. Golub, G.H., and Van Loan, C.F. 1996. *Matrix Computations*, 3rd edition. Johns Hopkins University Press.
10. Buatois, L., Caumon, G., Lévy, B. 2007. Concurrent number cruncher: An efficient sparse linear solver on the GPU. *Proceedings of the High-Performance Computation Conference (HPCC)*, Springer LNCS.
11. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D. 2007. Scan primitives for GPU computing. In *Proceedings of Graphics Hardware* (August): 97–106.
12. See Reference 3.

Links to the latest version of the CUDA development tools, documentation, code samples, and user discussion forums can be found at: http://www.nvidia.com/CUDA.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

JOHN NICKOLLS is director of architecture at NVIDIA for GPU computing. He was previously with Broadcom, Silicon Spice, Sun Microsystems, and was a cofounder of MasPar Computer. His interests include parallel processing systems, languages, and architectures. He has a B.S. in electrical engineering and computer science from the University of Illinois, and M.S. and Ph.D. degrees in electrical engineering from Stanford University.

IAN BUCK works for NVIDIA as the GPU-Compute software manager. He completed his Ph.D. at the Stanford Graphics Lab in 2004. His thesis was titled "Stream Computing on Graphics Hardware," researching programming models and computing strategies for using graphics hardware as a general-purpose computing platform. His work included developing the Brook software tool chain for abstracting the GPU as a general-purpose streaming coprocessor.

MICHAEL GARLAND is a research scientist with NVIDIA Research. Prior to joining NVIDIA, he was an assistant professor in the department of computer science at the University of Illinois at Urbana-Champaign. He received Ph.D. and B.S. degrees from Carnegie Mellon University. His research interests include computer graphics and visualization, geometric algorithms, and parallel algorithms and programming models.

KEVIN SKADRON is an associate professor in the department of computer science at the University of Virginia and is currently on sabbatical with NVIDIA Research. He received his Ph.D. from Princeton University and B.S. from Rice University. His research interests include power- and temperature-aware design, and manycore architecture and programming models. He is a senior member of the ACM.

NVIDIA Tesla: A Unified Graphics and Computing Architecture

To enable flexible, programmable graphics and high-performance computing, NVIDIA has developed the Tesla scalable unified graphics and parallel computing architecture. Its scalable parallel array of processors is massively multithreaded and programmable in C or via graphics APIs.

Erik Lindholm
John Nickolls
Stuart Oberman
John Montrym
NVIDIA

The modern 3D graphics processing unit (GPU) has evolved from a fixed-function graphics pipeline to a programmable parallel processor with computing power exceeding that of multicore CPUs. Traditional graphics pipelines consist of separate programmable stages of vertex processors executing vertex shader programs and pixel fragment processors executing pixel shader programs. (Montrym and Moreton provide additional background on the traditional graphics processor architecture.[1])

NVIDIA's Tesla architecture, introduced in November 2006 in the GeForce 8800 GPU, unifies the vertex and pixel processors and extends them, enabling high-performance parallel computing applications written in the C language using the Compute Unified Device Architecture (CUDA[2–4]) parallel programming model and development tools. The Tesla unified graphics and computing architecture is available in a scalable family of GeForce 8-series GPUs and Quadro GPUs for laptops, desktops, workstations, and servers. It also provides the processing architecture for the Tesla GPU computing platforms introduced in 2007 for high-performance computing.

In this article, we discuss the requirements that drove the unified graphics and parallel computing processor architecture, describe the Tesla architecture, and how it is enabling widespread deployment of parallel computing and graphics applications.

The road to unification

The first GPU was the GeForce 256, introduced in 1999. It contained a fixed-function 32-bit floating-point vertex transform and lighting processor and a fixed-function integer pixel-fragment pipeline, which were programmed with OpenGL and the Microsoft DX7 API.[5] In 2001, the GeForce 3 introduced the first programmable vertex processor executing vertex shaders, along with a configurable 32-bit floating-point fragment pipeline, programmed with DX8[5] and OpenGL.[6] The Radeon 9700, introduced in 2002, featured a programmable 24-bit floating-point pixel-fragment processor programmed with DX9 and OpenGL.[7,8] The GeForce FX added 32-bit floating-point pixel-fragment processors. The XBox 360 introduced an early unified GPU in 2005, allowing vertices and pixels to execute on the same processor.[9]

0272-1732/08/$20.00 © 2008 IEEE Published by the IEEE Computer Society.

Vertex processors operate on the vertices of primitives such as points, lines, and triangles. Typical operations include transforming coordinates into screen space, which are then fed to the setup unit and the rasterizer, and setting up lighting and texture parameters to be used by the pixel-fragment processors. Pixel-fragment processors operate on rasterizer output, which fills the interior of primitives, along with the interpolated parameters.

Vertex and pixel-fragment processors have evolved at different rates: Vertex processors were designed for low-latency, high-precision math operations, whereas pixel-fragment processors were optimized for high-latency, lower-precision texture filtering. Vertex processors have traditionally supported more-complex processing, so they became programmable first. For the last six years, the two processor types have been functionally converging as the result of a need for greater programming generality. However, the increased generality also increased the design complexity, area, and cost of developing two separate processors.

Because GPUs typically must process more pixels than vertices, pixel-fragment processors traditionally outnumber vertex processors by about three to one. However, typical workloads are not well balanced, leading to inefficiency. For example, with large triangles, the vertex processors are mostly idle, while the pixel processors are fully busy. With small triangles, the opposite is true. The addition of more-complex primitive processing in DX10 makes it much harder to select a fixed processor ratio.[10] All these factors influenced the decision to design a unified architecture.

A primary design objective for Tesla was to execute vertex and pixel-fragment shader programs on the same unified processor architecture. Unification would enable dynamic load balancing of varying vertex- and pixel-processing workloads and permit the introduction of new graphics shader stages, such as geometry shaders in DX10. It also let a single team focus on designing a fast and efficient processor and allowed the sharing of expensive hardware such as the texture units. The generality required of a unified processor opened the door to a completely new GPU parallel-computing capability. The downside of this generality was the difficulty of efficient load balancing between different shader types.

Other critical hardware design requirements were architectural scalability, performance, power, and area efficiency.

The Tesla architects developed the graphics feature set in coordination with the development of the Microsoft Direct3D DirectX 10 graphics API.[10] They developed the GPU's computing feature set in coordination with the development of the CUDA C parallel programming language, compiler, and development tools.

Tesla architecture

The Tesla architecture is based on a scalable processor array. Figure 1 shows a block diagram of a GeForce 8800 GPU with 128 streaming-processor (SP) cores organized as 16 streaming multiprocessors (SMs) in eight independent processing units called texture/processor clusters (TPCs). Work flows from top to bottom, starting at the host interface with the system PCI-Express bus. Because of its unified-processor design, the physical Tesla architecture doesn't resemble the logical order of graphics pipeline stages. However, we will use the logical graphics pipeline flow to explain the architecture.

At the highest level, the GPU's scalable streaming processor array (SPA) performs all the GPU's programmable calculations. The scalable memory system consists of external DRAM control and fixed-function raster operation processors (ROPs) that perform color and depth frame buffer operations directly on memory. An interconnection network carries computed pixel-fragment colors and depth values from the SPA to the ROPs. The network also routes texture memory read requests from the SPA to DRAM and read data from DRAM through a level-2 cache back to the SPA.

The remaining blocks in Figure 1 deliver input work to the SPA. The input assembler collects vertex work as directed by the input command stream. The vertex work distri-

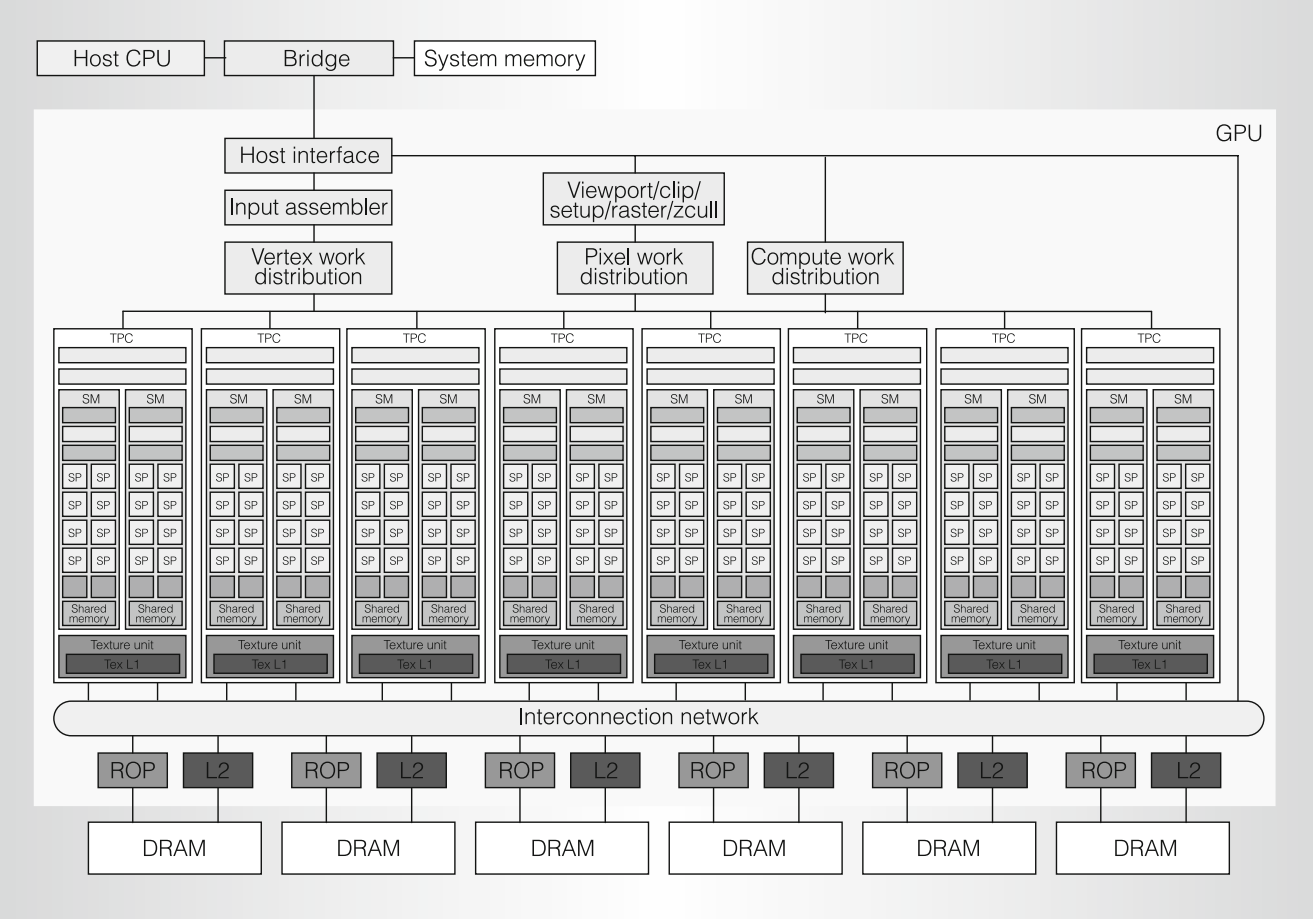


Figure 1. Tesla unified graphics and computing GPU architecture. TPC: texture/processor cluster; SM: streaming multiprocessor; SP: streaming processor; Tex: texture, ROP: raster operation processor.

bution block distributes vertex work packets to the various TPCs in the SPA. The TPCs execute vertex shader programs, and (if enabled) geometry shader programs. The resulting output data is written to on-chip buffers. These buffers then pass their results to the viewport/clip/setup/raster/zcull block to be rasterized into pixel fragments. The pixel work distribution unit distributes pixel fragments to the appropriate TPCs for pixel-fragment processing. Shaded pixel-fragments are sent across the interconnection network for processing by depth and color ROP units. The compute work distribution block dispatches compute thread arrays to the TPCs. The SPA accepts and processes work for multiple logical streams simultaneously. Multiple clock domains for GPU units, processors, DRAM, and other units allow independent power and performance optimizations.

Command processing

The GPU host interface unit communicates with the host CPU, responds to commands from the CPU, fetches data from system memory, checks command consistency, and performs context switching.

The input assembler collects geometric primitives (points, lines, triangles, line strips, and triangle strips) and fetches associated vertex input attribute data. It has peak rates of one primitive per clock and eight scalar attributes per clock at the GPU core clock, which is typically 600 MHz.

The work distribution units forward the input assembler's output stream to the array of processors, which execute vertex, geometry, and pixel shader programs, as well as computing programs. The vertex and compute work distribution units deliver work to processors in a round-robin scheme. Pixel

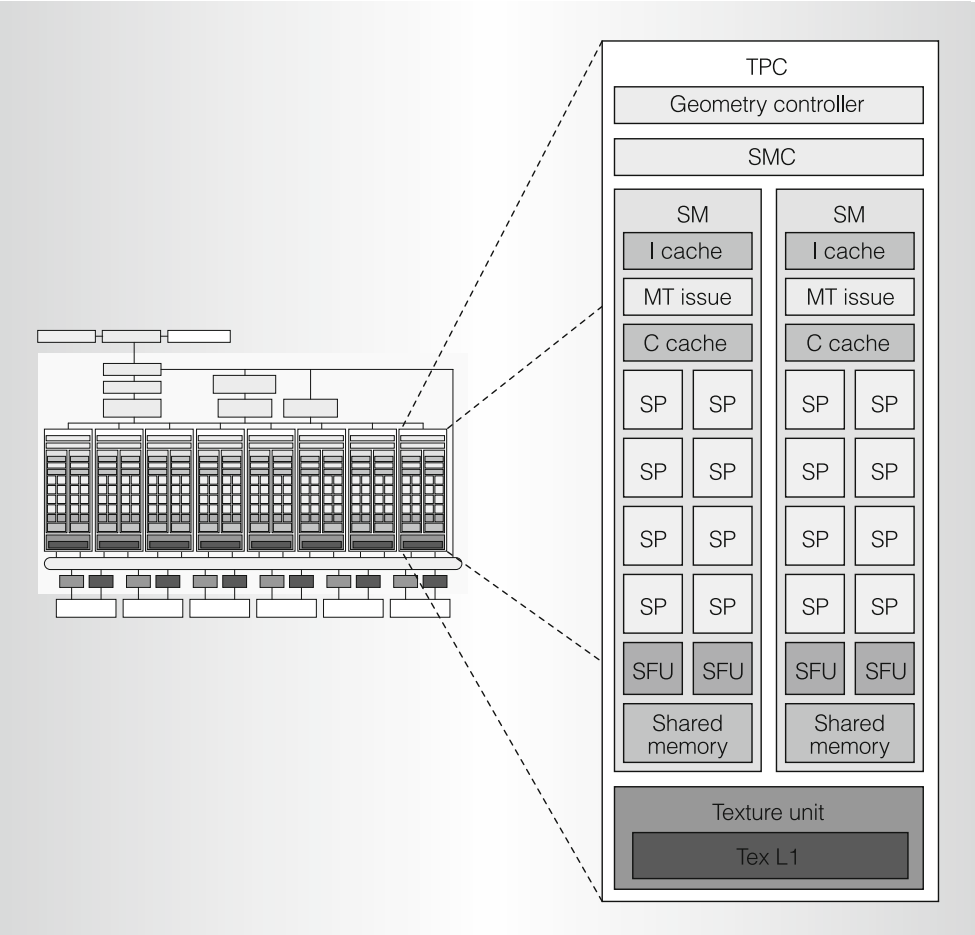


Figure 2. Texture/processor cluster (TPC).

work distribution is based on the pixel location.

Streaming processor array

The SPA executes graphics shader thread programs and GPU computing programs and provides thread control and management. Each TPC in the SPA roughly corresponds to a quad-pixel unit in previous architectures.[1] The number of TPCs determines a GPU's programmable processing performance and scales from one TPC in a small GPU to eight or more TPCs in high-performance GPUs.

Texture/processor cluster

As Figure 2 shows, each TPC contains a geometry controller, an SM controller (SMC), two streaming multiprocessors (SMs), and a texture unit. Figure 3 expands each SM to show its eight SP cores. To balance the expected ratio of math operations to texture operations, one texture unit serves two SMs. This architectural ratio can vary as needed.

Geometry controller

The geometry controller maps the logical graphics vertex pipeline into recirculation on the physical SMs by directing all primitive and vertex attribute and topology flow in the TPC. It manages dedicated on-chip input and output vertex attribute storage and forwards contents as required.

DX10 has two stages dealing with vertex and primitive processing: the vertex shader and the geometry shader. The vertex shader processes one vertex's attributes independently of other vertices. Typical operations are position space transforms and color and texture coordinate generation. The geometry shader follows the vertex shader and deals with a whole primitive and its vertices. Typical operations are edge extrusion for

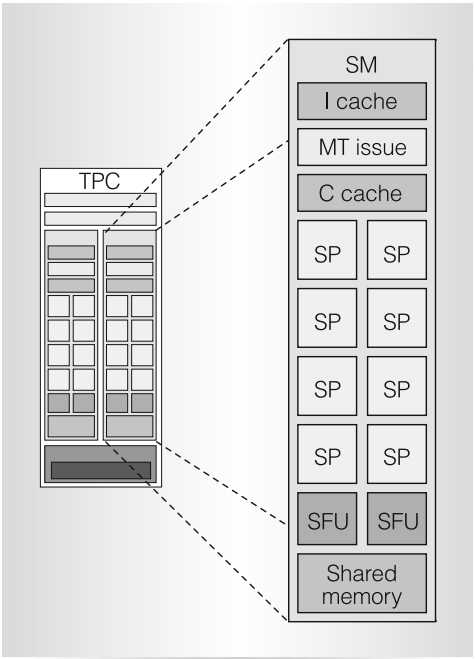


Figure 3. Streaming multiprocessor (SM).

stencil shadow generation and cube map texture generation. Geometry shader output primitives go to later stages for clipping, viewport transformation, and rasterization into pixel fragments.

Streaming multiprocessor

The SM is a unified graphics and computing multiprocessor that executes vertex, geometry, and pixel-fragment shader programs and parallel computing programs. As Figure 3 shows, the SM consists of eight streaming processor (SP) cores, two special-function units (SFUs), a multithreaded instruction fetch and issue unit (MT Issue), an instruction cache, a read-only constant cache, and a 16-Kbyte read/write shared memory.

The shared memory holds graphics input buffers or shared data for parallel computing. To pipeline graphics workloads through the SM, vertex, geometry, and pixel threads have independent input and output buffers. Workloads can arrive and depart independently of thread execution. Geometry threads, which generate variable amounts of output per thread, use separate output buffers.

Each SP core contains a scalar multiply-add (MAD) unit, giving the SM eight MAD units. The SM uses its two SFU units for transcendental functions and attribute interpolation—the interpolation of pixel attributes from vertex attributes defining a primitive. Each SFU also contains four floating-point multipliers. The SM uses the TPC texture unit as a third execution unit and uses the SMC and ROP units to implement external memory load, store, and atomic accesses. A low-latency interconnect network between the SPs and the shared-memory banks provides shared-memory access.

The GeForce 8800 Ultra clocks the SPs and SFU units at 1.5 GHz, for a peak of 36 Gflops per SM. To optimize power and area efficiency, some SM non-data-path units operate at half the SP clock rate.

SM multithreading. A graphics vertex or pixel shader is a program for a single thread that describes how to process a vertex or a pixel. Similarly, a CUDA kernel is a C program for a single thread that describes how one thread computes a result. Graphics and computing applications instantiate many parallel threads to render complex images and compute large result arrays. To dynamically balance shifting vertex and pixel shader thread workloads, the unified SM concurrently executes different thread programs and different types of shader programs.

To efficiently execute hundreds of threads in parallel while running several different programs, the SM is hardware multithreaded. It manages and executes up to 768 concurrent threads in hardware with zero scheduling overhead.

To support the independent vertex, primitive, pixel, and thread programming model of graphics shading languages and the CUDA C/C++ language, each SM thread has its own thread execution state and can execute an independent code path. Concurrent threads of computing programs can synchronize at a barrier with a single SM instruction. Lightweight thread creation, zero-overhead thread scheduling, and fast barrier synchronization support very fine-grained parallelism efficiently.

Single-instruction, multiple-thread. To manage and execute hundreds of threads running

several different programs efficiently, the Tesla SM uses a new processor architecture we call single-instruction, multiple-thread (SIMT). The SM's SIMT multithreaded instruction unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. The term *warp* originates from weaving, the first parallel-thread technology. Figure 4 illustrates SIMT scheduling. The SIMT warp size of 32 parallel threads provides efficiency on plentiful fine-grained pixel threads and computing threads.

Each SM manages a pool of 24 warps, with a total of 768 threads. Individual threads composing a SIMT warp are of the same type and start together at the same program address, but they are otherwise free to branch and execute independently. At each instruction issue time, the SIMT multithreaded instruction unit selects a warp that is ready to execute and issues the next instruction to that warp's active threads. A SIMT instruction is broadcast synchronously to a warp's active parallel threads; individual threads can be inactive due to independent branching or predication.

The SM maps the warp threads to the SP cores, and each thread executes independently with its own instruction address and register state. A SIMT processor realizes full efficiency and performance when all 32 threads of a warp take the same execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads reconverge to the original execution path. The SM uses a branch synchronization stack to manage independent threads that diverge and converge. Branch divergence only occurs within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths. As a result, Tesla architecture GPUs are dramatically more efficient and flexible on branching code than previous generation GPUs, as their 32-thread warps are much narrower than the SIMD width of prior GPUs.[1]

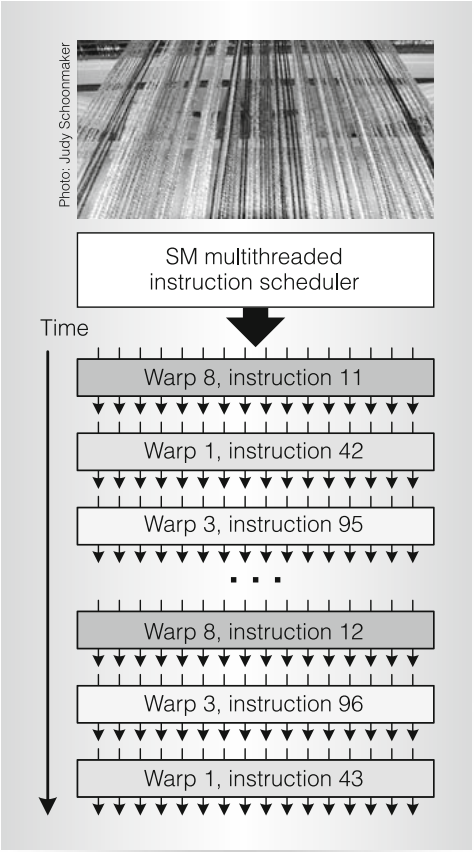


Figure 4. Single-instruction, multiple-thread (SIMT) warp scheduling.

SIMT architecture is similar to single-instruction, multiple-data (SIMD) design, which applies one instruction to multiple data lanes. The difference is that SIMT applies one instruction to multiple independent threads in parallel, not just multiple data lanes. A SIMD instruction controls a vector of multiple data lanes together and exposes the vector width to the software, whereas a SIMT instruction controls the execution and branching behavior of one thread.

In contrast to SIMD vector architectures, SIMT enables programmers to write thread-level parallel code for independent threads as well as data-parallel code for coordinated threads. For program correctness, programmers can essentially ignore SIMT execution attributes such as warps; however, they can achieve substantial performance improvements by writing code that seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in

traditional codes: Programmers can safely ignore cache line size when designing for correctness but must consider it in the code structure when designing for peak performance. SIMD vector architectures, on the other hand, require the software to manually coalesce loads into vectors and to manually manage divergence.

SIMT warp scheduling. The SIMT approach of scheduling independent warps is simpler than previous GPU architectures' complex scheduling. A warp consists of up to 32 threads of the same type—vertex, geometry, pixel, or compute. The basic unit of pixel-fragment shader processing is the 2 $\times$ 2 pixel quad. The SM controller groups eight pixel quads into a warp of 32 threads. It similarly groups vertices and primitives into warps and packs 32 computing threads into a warp. The SIMT design shares the SM instruction fetch and issue unit efficiently across 32 threads but requires a full warp of active threads for full performance efficiency.

As a unified graphics processor, the SM schedules and executes multiple warp types concurrently—for example, concurrently executing vertex and pixel warps. The SM warp scheduler operates at half the 1.5-GHz processor clock rate. At each cycle, it selects one of the 24 warps to execute a SIMT warp instruction, as Figure 4 shows. An issued warp instruction executes as two sets of 16 threads over four processor cycles. The SP cores and SFU units execute instructions independently, and by issuing instructions between them on alternate cycles, the scheduler can keep both fully occupied.

Implementing zero-overhead warp scheduling for a dynamic mix of different warp programs and program types was a challenging design problem. A scoreboard qualifies each warp for issue each cycle. The instruction scheduler prioritizes all ready warps and selects the one with highest priority for issue. Prioritization considers warp type, instruction type, and "fairness" to all warps executing in the SM.

SM instructions. The Tesla SM executes scalar instructions, unlike previous GPU vector instruction architectures. Shader programs are becoming longer and more scalar, and it is increasingly difficult to fully occupy even two components of the prior four-component vector architecture. Previous architectures employed vector packing—combining sub-vectors of work to gain efficiency—but that complicated the scheduling hardware as well as the compiler. Scalar instructions are simpler and compiler friendly. Texture instructions remain vector based, taking a source coordinate vector and returning a filtered color vector.

High-level graphics and computing-language compilers generate intermediate instructions, such as DX10 vector or PTX scalar instructions,[10,2] which are then optimized and translated to binary GPU instructions. The optimizer readily expands DX10 vector instructions to multiple Tesla SM scalar instructions. PTX scalar instructions optimize to Tesla SM scalar instructions about one to one. PTX provides a stable target ISA for compilers and provides compatibility over several generations of GPUs with evolving binary instruction set architectures. Because the intermediate languages use virtual registers, the optimizer analyzes data dependencies and allocates real registers. It eliminates dead code, folds instructions together when feasible, and optimizes SIMT branch divergence and convergence points.

Instruction set architecture. The Tesla SM has a register-based instruction set including floating-point, integer, bit, conversion, transcendental, flow control, memory load/store, and texture operations.

Floating-point and integer operations include add, multiply, multiply-add, minimum, maximum, compare, set predicate, and conversions between integer and floating-point numbers. Floating-point instructions provide source operand modifiers for negation and absolute value. Transcendental function instructions include cosine, sine, binary exponential, binary logarithm, reciprocal, and reciprocal square root. Attribute interpolation instructions provide efficient generation of pixel attributes. Bitwise operators include shift left, shift right, logic operators, and move. Control

flow includes branch, call, return, trap, and barrier synchronization.

The floating-point and integer instructions can also set per-thread status flags for zero, negative, carry, and overflow, which the thread program can use for conditional branching.

Memory access instructions. The texture instruction fetches and filters texture samples from memory via the texture unit. The ROP unit writes pixel-fragment output to memory.

To support computing and C/C++ language needs, the Tesla SM implements memory load/store instructions in addition to graphics texture fetch and pixel output. Memory load/store instructions use integer byte addressing with register-plus-offset address arithmetic to facilitate conventional compiler code optimizations.

For computing, the load/store instructions access three read/write memory spaces:

- *local* memory for per-thread, private, temporary data (implemented in external DRAM);
- *shared* memory for low-latency access to data shared by cooperating threads in the same SM; and
- *global* memory for data shared by all threads of a computing application (implemented in external DRAM).

The memory instructions load-global, store-global, load-shared, store-shared, load-local, and store-local access global, shared, and local memory. Computing programs use the fast barrier synchronization instruction to synchronize threads within the SM that communicate with each other via shared and global memory.

To improve memory bandwidth and reduce overhead, the local and global load/store instructions coalesce individual parallel thread accesses from the same warp into fewer memory block accesses. The addresses must fall in the same block and meet alignment criteria. Coalescing memory requests boosts performance significantly over separate requests. The large thread count, together with support for many outstanding load requests, helps cover load-to-use latency for local and global memory implemented in external DRAM.

The latest Tesla architecture GPUs provide efficient atomic memory operations, including integer add, minimum, maximum, logic operators, swap, and compare-and-swap operations. Atomic operations facilitate parallel reductions and parallel data structure management.

Streaming processor. The SP core is the primary thread processor in the SM. It performs the fundamental floating-point operations, including add, multiply, and multiply-add. It also implements a wide variety of integer, comparison, and conversion operations. The floating-point add and multiply operations are compatible with the IEEE 754 standard for single-precision FP numbers, including not-a-number (NaN) and infinity values. The unit is fully pipelined, and latency is optimized to balance delay and area.

The add and multiply operations use IEEE round-to-nearest-even as the default rounding mode. The multiply-add operation performs a multiplication with truncation, followed by an add with round-to-nearest-even. The SP flushes denormal source operands to sign-preserved zero and flushes results that underflow the target output exponent range to sign-preserved zero after rounding.

Special-function unit. The SFU supports computation of both transcendental functions and planar attribute interpolation.[11] A traditional vertex or pixel shader design contains a functional unit to compute transcendental functions. Pixels also need an attribute-interpolating unit to compute the per-pixel attribute values at the pixel's x, y location, given the attribute values at the primitive's vertices.

For functional evaluation, we use quadratic interpolation based on enhanced minimax approximations to approximate the reciprocal, reciprocal square root, $\log_2 x$, 2^x, and sin/cos functions. Table 1 shows the accuracy of the function estimates. The SFU unit generates one 32-bit floating point result per cycle.

Table 1. Function approximation statistics.

Function	Input interval	Accuracy (good bits)	ULP* error	% exactly rounded	Monotonic
$1/x$	[1, 2)	24.02	0.98	87	Yes
1/sqrt(x)	[1, 4)	23.40	1.52	78	Yes
2^x	[0, 1)	22.51	1.41	74	Yes
$\log_2 x$	[1, 2)	22.57	N/A**	N/A	Yes
sin/cos	[0, π/2)	22.47	N/A	N/A	No

* ULP: unit-in-the-last-place.
** N/A: not applicable.

The SFU also supports attribute interpolation, to enable accurate interpolation of attributes such as color, depth, and texture coordinates. The SFU must interpolate these attributes in the (x, y) screen space to determine the values of the attributes at each pixel location. We express the value of a given attribute U in an (x, y) plane in plane equations of the following form:

$$U(x, y) = (A_U \times x + B_U \times y + C_U)/(A_W \times x + B_W \times y + C_W)$$

where A, B, and C are interpolation parameters associated with each attribute U, and W is related to the distance of the pixel from the viewer for perspective projection. The attribute interpolation hardware in the SFU is fully pipelined, and it can interpolate four samples per cycle.

In a shader program, the SFU can generate perspective-corrected attributes as follows:

- Interpolate $1/W$, and invert to form W.
- Interpolate U/W.
- Multiply U/W by W to form perspective-correct U.

SM controller. The SMC controls multiple SMs, arbitrating the shared texture unit, load/store path, and I/O path. The SMC serves three graphics workloads simultaneously: vertex, geometry, and pixel. It packs each of these input types into the warp width, initiating shader processing, and unpacks the results.

Each input type has independent I/O paths, but the SMC is responsible for load balancing among them. The SMC supports static and dynamic load balancing based on driver-recommended allocations, current allocations, and relative difficulty of additional resource allocation. Load balancing of the workloads was one of the more challenging design problems due to its impact on overall SPA efficiency.

Texture unit

The texture unit processes one group of four threads (vertex, geometry, pixel, or compute) per cycle. Texture instruction sources are texture coordinates, and the outputs are filtered samples, typically a four-component (RGBA) color. Texture is a separate unit external to the SM connected via the SMC. The issuing SM thread can continue execution until a data dependency stall.

Each texture unit has four texture address generators and eight filter units, for a peak GeForce 8800 Ultra rate of 38.4 gigabilerps/s (a bilerp is a bilinear interpolation of four samples). Each unit supports full-speed 2:1 anisotropic filtering, as well as high-dynamic-range (HDR) 16-bit and 32-bit floating-point data format filtering.

The texture unit is deeply pipelined. Although it contains a cache to capture filtering locality, it streams hits mixed with misses without stalling.

Rasterization

Geometry primitives output from the SMs go in their original round-robin input order to the viewport/clip/setup/raster/zcull block. The viewport and clip units clip the primitives to the standard view frustum and to any enabled user clip planes. They transform postclipping vertices into screen (pixel) space and reject whole primitives outside the view volume as well as back-facing primitives.

Surviving primitives then go to the setup unit, which generates edge equations for the rasterizer. Attribute plane equations are also generated for linear interpolation of pixel attributes in the pixel shader. A coarse-rasterization stage generates all pixel tiles that are at least partially inside the primitive.

The zcull unit maintains a hierarchical z surface, rejecting pixel tiles if they are conservatively known to be occluded by previously drawn pixels. The rejection rate is up to 256 pixels per clock. The screen is subdivided into tiles; each TPC processes a predetermined subset. The pixel tile address therefore selects the destination TPC. Pixel tiles that survive zcull then go to a fine-rasterization stage that generates detailed coverage information and depth values for the pixels.

OpenGL and Direct3D require that a depth test be performed after the pixel shader has generated final color and depth values. When possible, for certain combinations of API state, the Tesla GPU performs the depth test and update ahead of the fragment shader, possibly saving thousands of cycles of processing time, without violating the API-mandated semantics.

The SMC assembles surviving pixels into warps to be processed by a SM running the current pixel shader. When the pixel shader has finished, the pixels are optionally depth tested if this was not done ahead of the shader. The SMC then sends surviving pixels and associated data to the ROP.

Raster operations processor

Each ROP is paired with a specific memory partition. The TPCs feed data to the ROPs via an interconnection network. ROPs handle depth and stencil testing and updates and color blending and updates. The memory controller uses lossless color (up to 8:1) and depth compression (up to 8:1) to reduce bandwidth. Each ROP has a peak rate of four pixels per clock and supports 16-bit floating-point and 32-bit floating-point HDR formats. ROPs support double-rate-depth processing when color writes are disabled.

Each memory partition is 64 bits wide and supports double-data-rate DDR2 and graphics-oriented GDDR3 protocols at up to 1 GHz, yielding a bandwidth of about 16 Gbytes/s.

Antialiasing support includes up to 16× multisampling and supersampling. HDR formats are fully supported. Both algorithms support 1, 2, 4, 8, or 16 samples per pixel and generate a weighted average of the samples to produce the final pixel color. Multisampling executes the pixel shader once to generate a color shared by all pixel samples, whereas supersampling runs the pixel shader once per sample. In both cases, depth values are correctly evaluated for each sample, as required for correct interpenetration of primitives.

Because multisampling runs the pixel shader once per pixel (rather than once per sample), multisampling has become the most popular antialiasing method. Beyond four samples, however, storage cost increases faster than image quality improves, especially with HDR formats. For example, a single 1,600 × 1,200 pixel surface, storing 16 four-component, 16-bit floating-point samples, requires 1,600 × 1,200 × 16 × (64 bits color + 32 bits depth) = 368 Mbytes.

For the vast majority of edge pixels, two colors are enough; what matters is more-detailed coverage information. The coverage-sampling antialiasing (CSAA) algorithm provides low-cost-per-coverage samples, allowing upward scaling. By computing and storing Boolean coverage at up to 16 samples and compressing redundant color and depth and stencil information into the memory footprint and bandwidth of four or eight samples, 16× antialiasing quality can be achieved at 4× antialiasing performance. CSAA is compatible with existing rendering

Table 2. Comparison of antialiasing modes.

	Antialiasing mode								
Feature	**Brute-force supersampling**			**Multisampling**			**Coverage sampling**		
Quality level	1×	4×	16×	1×	4×	16×	1×	4×	16×
Texture and shader samples	1	4	16	1	1	1	1	1	1
Stored color and *z* samples	1	4	16	1	4	16	1	4	4
Coverage samples	1	4	16	1	4	16	1	4	16

techniques including HDR and stencil algorithms. Edges defined by the intersection of interpenetrating polygons are rendered at the stored sample count quality (4× or 8×). Table 2 summarizes the storage requirements of the three algorithms.

Memory and interconnect

The DRAM memory data bus width is 384 pins, arranged in six independent partitions of 64 pins each. Each partition owns 1/6 of the physical address space. The memory partition units directly enqueue requests. They arbitrate among hundreds of in-flight requests from the parallel stages of the graphics and computation pipelines. The arbitration seeks to maximize total DRAM transfer efficiency, which favors grouping related requests by DRAM bank and read/write direction, while minimizing latency as far as possible. The memory controllers support a wide range of DRAM clock rates, protocols, device densities, and data bus widths.

Interconnection network. A single hub unit routes requests to the appropriate partition from the nonparallel requesters (PCI-Express, host and command front end, input assembler, and display). Each memory partition has its own depth and color ROP units, so ROP memory traffic originates locally. Texture and load/store requests, however, can occur between any TPC and any memory partition, so an interconnection network routes requests and responses.

Memory management unit. All processing engines generate addresses in a virtual address space. A memory management unit performs virtual to physical translation. Hardware reads the page tables from local memory to respond to misses on behalf of a hierarchy of translation look-aside buffers spread out among the rendering engines.

Parallel computing architecture

The Tesla scalable parallel computing architecture enables the GPU processor array to excel in throughput computing, executing high-performance computing applications as well as graphics applications. Throughput applications have several properties that distinguish them from CPU serial applications:

- extensive data parallelism—thousands of computations on independent data elements;
- modest task parallelism—groups of threads execute the same program, and different groups can run different programs;
- intensive floating-point arithmetic;
- latency tolerance—performance is the amount of work completed in a given time;
- streaming data flow—requires high memory bandwidth with relatively little data reuse;
- modest inter-thread synchronization and communication—graphics threads do not communicate, and parallel computing applications require limited synchronization and communication.

GPU parallel performance on throughput problems has doubled every 12 to 18 months, pulled by the insatiable demands of the 3D game market. Now, Tesla GPUs in laptops, desktops, workstations,

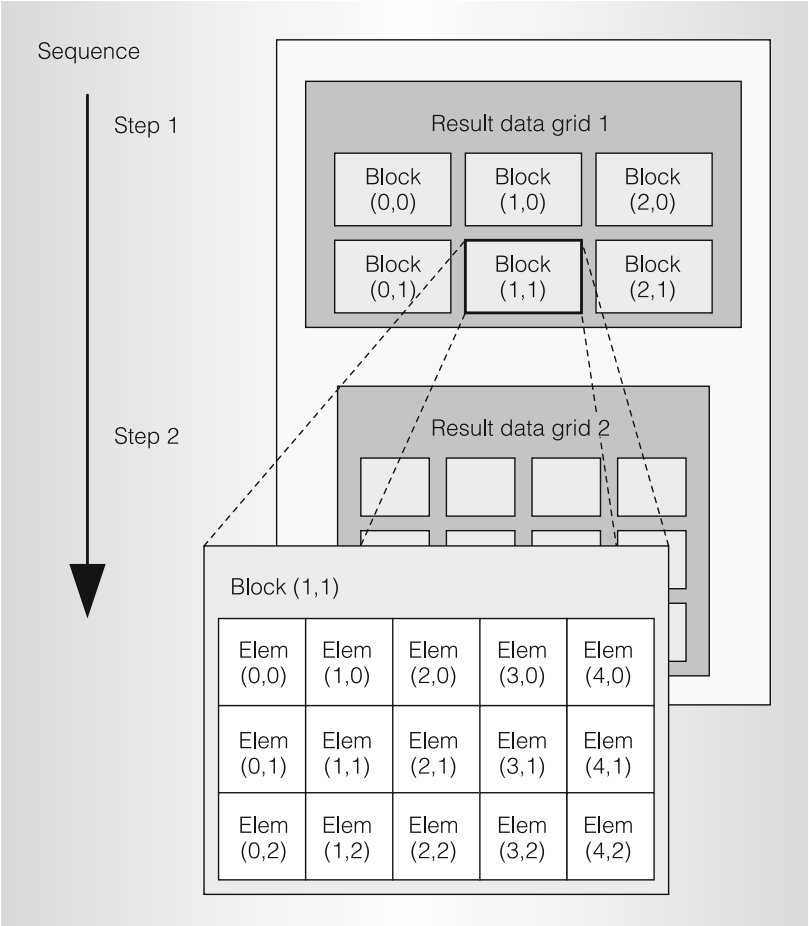


Figure 5. Decomposing result data into a grid of blocks partitioned into elements to be computed in parallel.

and systems are programmable in C with CUDA tools, using a simple parallel programming model.

Data-parallel problem decomposition

To map a large computing problem effectively to a highly parallel processing architecture, the programmer or compiler decomposes the problem into many small problems that can be solved in parallel. For example, the programmer partitions a large result data array into blocks and further partitions each block into elements, so that the result blocks can be computed independently in parallel, and the elements within each block can be computed cooperatively in parallel. Figure 5 shows the decomposition of a result data array into a 3×2 grid of blocks, in which each block is further decomposed into a 5×3 array of elements. The two-level parallel decomposition maps naturally to the Tesla architecture: Parallel SMs compute result blocks, and parallel threads compute result elements.

The programmer or compiler writes a program that computes a sequence of result grids, partitioning each result grid into coarse-grained result blocks that are computed independently in parallel. The program computes each result block with an array of fine-grained parallel threads, partitioning the work among threads that compute result elements.

Cooperative thread array or thread block

Unlike the graphics programming model, which executes parallel shader threads independently, parallel-computing programming models require that parallel threads synchronize, communicate, share data, and cooperate to efficiently compute a result. To manage large numbers of concurrent threads that can cooperate, the Tesla computing architecture introduces the *cooperative thread array* (CTA), called a *thread block* in CUDA terminology.

A CTA is an array of concurrent threads that execute the same thread program and can cooperate to compute a result. A CTA consists of 1 to 512 concurrent threads, and each thread has a unique thread ID (TID), numbered 0 through m. The programmer declares the 1D, 2D, or 3D CTA shape and dimensions in threads. The TID has one, two, or three dimension indices. Threads of a CTA can share data in global or shared memory and can synchronize with the barrier instruction. CTA thread programs use their TIDs to select work and index shared data arrays. Multidimensional TIDs can eliminate integer divide and remainder operations when indexing arrays.

Each SM executes up to eight CTAs concurrently, depending on CTA resource demands. The programmer or compiler declares the number of threads, registers, shared memory, and barriers required by the CTA program. When an SM has sufficient available resources, the SMC creates the CTA and assigns TID numbers to each thread. The SM executes the CTA threads concurrently as SIMT warps of 32 parallel threads.

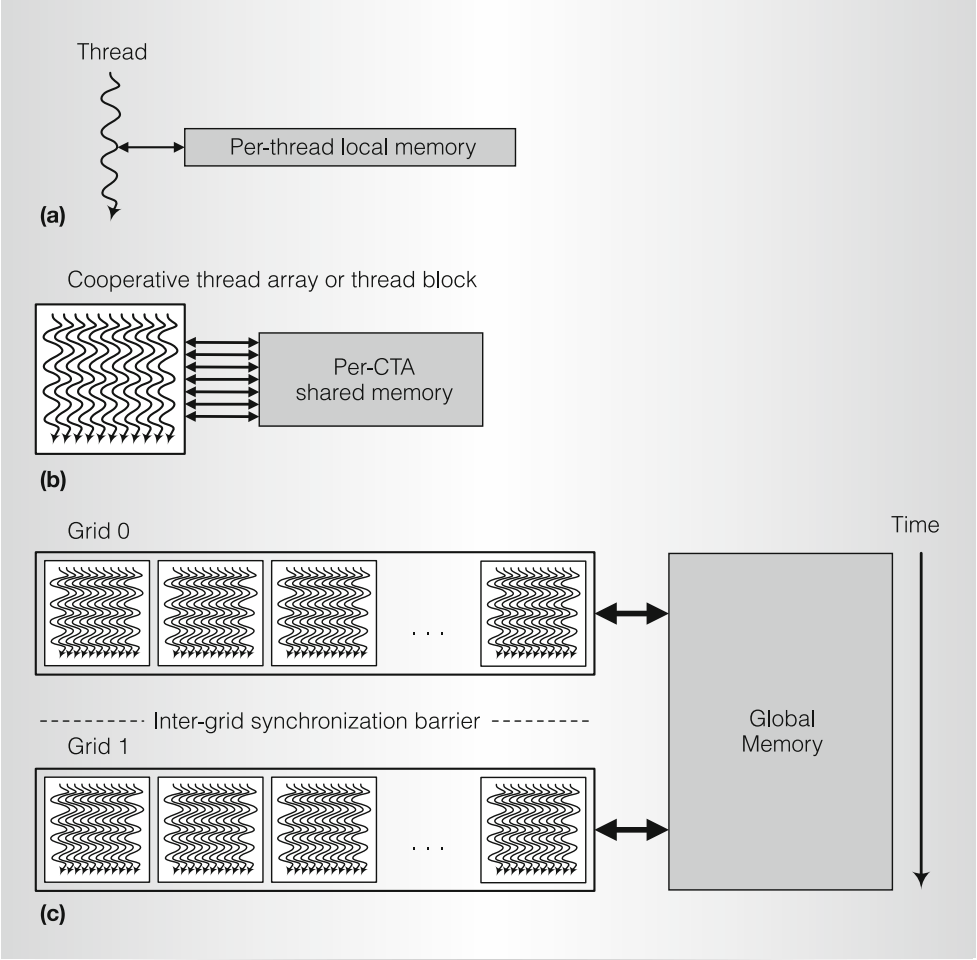


Figure 6. Nested granularity levels: thread (a), cooperative thread array (b), and grid (c). These have corresponding memory-sharing levels: local per-thread, shared per-CTA, and global per-application.

CTA grids

To implement the coarse-grained block and grid decomposition of Figure 5, the GPU creates CTAs with unique CTA ID and grid ID numbers. The compute work distributor dynamically balances the GPU workload by distributing a stream of CTA work to SMs with sufficient available resources.

To enable a compiled binary program to run unchanged on large or small GPUs with any number of parallel SM processors, CTAs execute independently and compute result blocks independently of other CTAs in the same grid. Sequentially dependent application steps map to two sequentially dependent grids. The dependent grid waits for the first grid to complete; then the CTAs of the dependent grid read the result blocks written by the first grid.

Parallel granularity

Figure 6 shows levels of parallel granularity in the GPU computing model. The three levels are

- *thread*—computes result elements selected by its TID;
- *CTA*—computes result blocks selected by its CTA ID;
- *grid*—computes many result blocks, and sequential grids compute sequentially dependent application steps.

Higher levels of parallelism use multiple GPUs per CPU and clusters of multi-GPU nodes.

Parallel memory sharing

Figure 6 also shows levels of parallel read/write memory sharing:

- *local*—each executing thread has a private per-thread local memory for register spill, stack frame, and addressable temporary variables;
- *shared*—each executing CTA has a per-CTA shared memory for access to data shared by threads in the same CTA;
- *global*—sequential grids communicate and share large data sets in global memory.

Threads communicating in a CTA use the fast barrier synchronization instruction to wait for writes to shared or global memory to complete before reading data written by other threads in the CTA. The load/store memory system uses a relaxed memory order that preserves the order of reads and writes to the same address from the same issuing thread and from the viewpoint of CTA threads coordinating with the barrier synchronization instruction. Sequentially dependent grids use a global intergrid synchronization barrier between grids to ensure global read/write ordering.

Transparent scaling of GPU computing

Parallelism varies widely over the range of GPU products developed for various market segments. A small GPU might have one SM with eight SP cores, while a large GPU might have many SMs totaling hundreds of SP cores.

The GPU computing architecture transparently scales parallel application performance with the number of SMs and SP cores. A GPU computing program executes on any size of GPU without recompiling, and is insensitive to the number of SM multiprocessors and SP cores. The program does not know or care how many processors it uses.

The key is decomposing the problem into independently computed blocks as described earlier. The GPU compute work distribution unit generates a stream of CTAs and distributes them to available SMs to compute each independent block. Scalable programs do not communicate among CTA blocks of the same grid; the same grid result is obtained if the CTAs execute in parallel on many cores, sequentially on one core, or partially in parallel on a few cores.

CUDA programming model

CUDA is a minimal extension of the C and C++ programming languages. A programmer writes a serial program that calls parallel kernels, which can be simple functions or full programs. The CUDA program executes serial code on the CPU and executes parallel kernels across a set of parallel threads on the GPU. The programmer organizes these threads into a hierarchy of thread blocks and grids as described earlier. (A CUDA thread block is a GPU CTA.)

Figure 7 shows a CUDA program executing a series of parallel kernels on a heterogeneous CPU–GPU system. **KernelA** and **KernelB** execute on the GPU as grids of **nBlkA and nBlkB** thread blocks (CTAs), which instantiate **nTidA** and **nTidB** threads per CTA.

The CUDA compiler **nvcc** compiles an integrated application C/C++ program containing serial CPU code and parallel GPU kernel code. The CUDA runtime API manages the GPU as a computing device that acts as a coprocessor to the host CPU with its own memory system.

The CUDA programming model is similar in style to a single-program multiple-data (SPMD) software model—it expresses parallelism explicitly, and each kernel executes on a fixed number of threads. However, CUDA is more flexible than most SPMD implementations because each kernel call dynamically creates a new grid with the right number of thread blocks and threads for that application step.

CUDA extends C/C++ with the declaration specifier keywords **__global__** for kernel entry functions, **__device__** for global variables, and **__shared__** for shared-memory variables. A CUDA kernel's text is simply a C function for one sequential thread. The built-in variables **threadIdx.{x, y, z}** and **blockIdx.{x, y, z}** provide the thread ID within a thread block (CTA), while **blockIdx** provides the CTA ID within a grid. The extended function call syntax **kernel<<<nBlocks,nThreads>>>(args);**

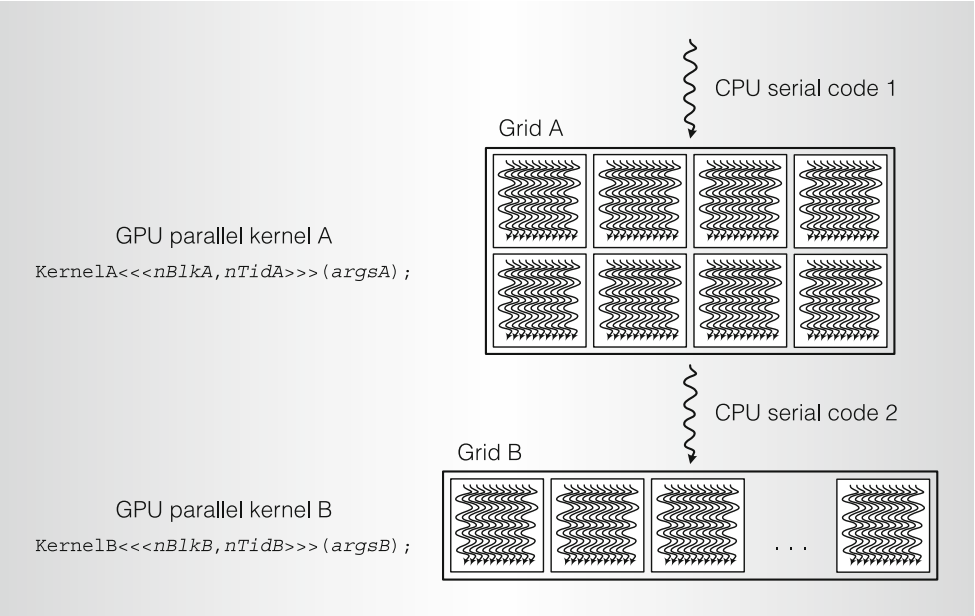


Figure 7. CUDA program sequence of kernel A followed by kernel B on a heterogeneous CPU–GPU system.

invokes a parallel kernel function on a grid of **nBlocks**, where each block instantiates **nThreads** concurrent threads, and **args** are ordinary arguments to function **kernel()**.

Figure 8 shows an example serial C program and a corresponding CUDA C program. The serial C program uses two nested loops to iterate over each array index and compute **c[idx] = a[idx] + b[idx]** each trip. The parallel CUDA C program has no loops. It uses parallel threads to compute the same array indices in parallel, and each thread computes only one sum.

Scalability and performance

The Tesla unified architecture is designed for scalability. Varying the number of SMs, TPCs, ROPs, caches, and memory partitions provides the right mix for different performance and cost targets in the value, mainstream, enthusiast, and professional

```
void addMatrix
    (float *a, float *b, float *c, int N)
{
    int i, j, idx;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            idx = i + j*N;
            c[idx] = a[idx] + b[idx];
        }
    }
}
void  main()
{
    . . .
    addMatrix(a, b, c, N);
}
```

(a)

```
__global__ void addMatrixG
    (float *a, float *b, float *c, int N)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int j = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = i + j*N;
    if (i < N && j < N)
        c[idx] = a[idx] + b[idx];
}

void  main()
{
    dim3 dimBlock (blocksize, blocksize);
    dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
    addMatrixG<<<dimGrid, dimBlock>>>(a, b, c, N);
}
```

(b)

Figure 8. Serial C (a) and CUDA C (b) examples of programs that add arrays.

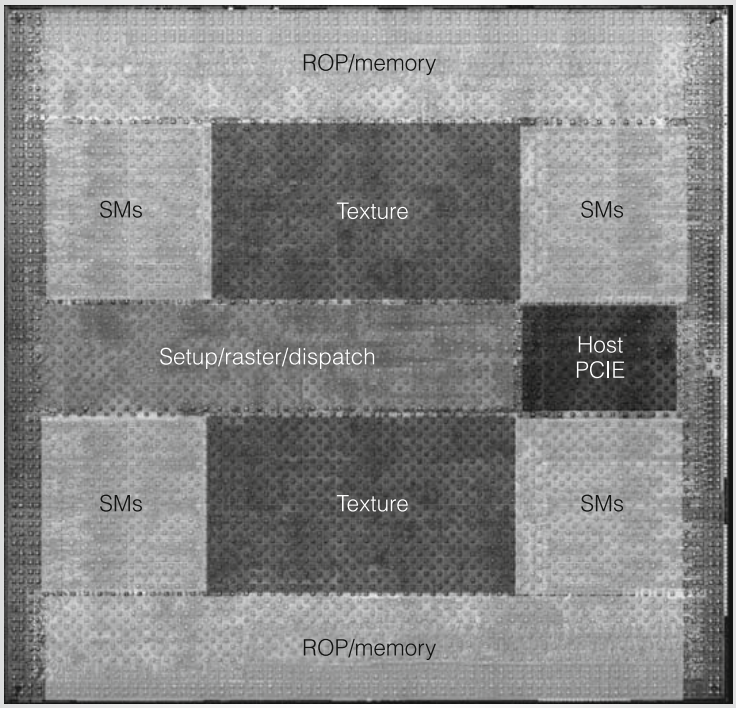


Figure 9. GeForce 8800 Ultra die layout.

market segments. NVIDIA's Scalable Link Interconnect (SLI) enables multiple GPUs to act together as one, providing further scalability.

CUDA C/C++ applications executing on Tesla computing platforms, Quadro workstations, and GeForce GPUs deliver compelling computing performance on a range of large problems, including more than 100× speedups on molecular modeling, more than 200 Gflops on *n*-body problems, and real-time 3D magnetic-resonance imaging.[12–14] For graphics, the GeForce 8800 GPU delivers high performance and image quality for the most demanding games.[15]

Figure 9 shows the GeForce 8800 Ultra physical die layout implementing the Tesla architecture shown in Figure 1. Implementation specifics include

- 681 million transistors, 470 mm^2;
- TSMC 90-nm CMOS;
- 128 SP cores in 16 SMs;
- 12,288 processor threads;
- 1.5-GHz processor clock rate;
- peak 576 Gflops in processors;
- 768-Mbyte GDDR3 DRAM;
- 384-pin DRAM interface;
- 1.08-GHz DRAM clock;
- 104-Gbyte/s peak bandwidth; and
- typical power of 150 W at 1.3 V.

The Tesla architecture is the first ubiquitous supercomputing platform. NVIDIA has shipped more than 50 million Tesla-based systems. This wide availability, coupled with C programmability and the CUDA software development environment, enables broad deployment of demanding parallel-computing and graphics applications.

With future increases in transistor density, the architecture will readily scale processor parallelism, memory partitions, and overall performance. Increased number of multiprocessors and memory partitions will support larger data sets and richer graphics and computing, without a change to the programming model.

We continue to investigate improved scheduling and load-balancing algorithms for the unified processor. Other areas of improvement are enhanced scalability for derivative products, reduced synchronization and communication overhead for compute programs, new graphics features, increased realized memory bandwidth, and improved power efficiency. MICRO

Acknowledgments

We thank the entire NVIDIA GPU development team for their extraordinary effort in bringing Tesla-based GPUs to market.

References

1. J. Montrym and H. Moreton, ''The GeForce 6800,'' *IEEE Micro*, vol. 25, no. 2, Mar./Apr. 2005, pp. 41-51.
2. *CUDA Technology*, NVIDIA, 2007, http://www.nvidia.com/CUDA.
3. *CUDA Programming Guide 1.1*, NVIDIA, 2007; http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf.
4. J. Nickolls, I. Buck, K. Skadron, and M. Garland, ''Scalable Parallel Programming with CUDA,'' *ACM Queue*, vol. 6, no. 2, Mar./Apr. 2008, pp. 40-53.
5. *DX Specification*, Microsoft; http://msdn.microsoft.com/directx.

6. E. Lindholm, M.J. Kilgard, and H. Moreton, ''A User-Programmable Vertex Engine,'' *Proc. 28th Ann. Conf. Computer Graphics and Interactive Techniques* (Siggraph 01), ACM Press, 2001, pp. 149-158.
7. G. Elder, ''Radeon 9700,'' Eurographics/Siggraph Workshop Graphics Hardware, Hot 3D Session, 2002, http://www.graphicshardware.org/previous/www_2002/presentations/Hot3D-RADEON9700.ppt.
8. *Microsoft DirectX 9 Programmable Graphics Pipeline*, Microsoft Press, 2003.
9. J. Andrews and N. Baker, ''Xbox 360 System Architecture,'' *IEEE Micro*, vol. 26, no. 2, Mar./Apr. 2006, pp. 25-37.
10. D. Blythe, ''The Direct3D 10 System,'' *ACM Trans. Graphics*, vol. 25, no. 3, July 2006, pp. 724-734.
11. S.F. Oberman and M.Y. Siu, ''A High-Performance Area-Efficient Multifunction Interpolator,'' *Proc. 17th IEEE Symp. Computer Arithmetic* (Arith-17), IEEE Press, 2005, pp. 272-279.
12. J.E. Stone et al., ''Accelerating Molecular Modeling Applications with Graphics Processors,'' *J. Computational Chemistry*, vol. 28, no. 16, 2007, pp. 2618-2640.
13. L. Nyland, M. Harris, and J. Prins, ''Fast N-Body Simulation with CUDA,'' *GPU Gems 3*, H. Nguyen, ed., Addison-Wesley, 2007, pp. 677-695.
14. S.S. Stone et al., ''How GPUs Can Improve the Quality of Magnetic Resonance Imaging,'' *Proc. 1st Workshop on General Purpose Processing on Graphics Processing Units,* 2007; http://www.gigascale.org/pubs/1175.html.
15. A.L. Shimpi and D. Wilson, ''NVIDIA's GeForce 8800 (G80): GPUs Re-architected for DirectX 10,'' *AnandTech*, Nov. 2006; http://www.anandtech.com/video/showdoc.aspx?i=2870.

Erik Lindholm is a distinguished engineer at NVIDIA, working in the architecture group. His research interests include graphics processor design and parallel graphics architectures. Lindholm has an MS in electrical engineering from the University of British Columbia.

John Nickolls is director of GPU computing architecture at NVIDIA. His interests include parallel processing systems, languages, and architectures. Nickolls has a BS in electrical engineering and computer science from the University of Illinois and MS and PhD degrees in electrical engineering from Stanford University.

Stuart Oberman is a design manager in the GPU hardware group at NVIDIA. His research interests include computer arithmetic, processor design, and parallel architectures. Oberman has a BS in electrical engineering from the University of Iowa and MS and PhD degrees in electrical engineering from Stanford University. He is a senior member of the IEEE.

John Montrym is a chief architect at NVIDIA, where he has worked in the development of several GPU product families. His research interests include graphics processor design, parallel graphics architectures, and hardware-software interfaces. Montrym has a BS in electrical engineering from the Massachusetts Institute of Technology.

Direct questions and comments about this article to Erik Lindholm or John Nickolls, NVIDIA, 2701 San Tomas Expressway, Santa Clara, CA 95050; elindholm@nvidia.com or jnickolls@nvidia.com.

Clipper: A Low-Latency Online Prediction Serving System

Daniel Crankshaw, Xin Wang, and Guilio Zhou, *University of California, Berkeley;*
Michael J. Franklin, *University of California, Berkeley, and The University of Chicago;*
Joseph E. Gonzalez and Ion Stoica, *University of California, Berkeley*

https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

This paper is included in the Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17).

March 27–29, 2017 • Boston, MA, USA

ISBN 978-1-931971-37-9

Open access to the Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation is sponsored by USENIX.

Clipper: A Low-Latency Online Prediction Serving System

Daniel Crankshaw[*], Xin Wang[*], Giulio Zhou[*]
Michael J. Franklin[*†], Joseph E. Gonzalez[*], Ion Stoica[*]
[*]*UC Berkeley* [†]*The University of Chicago*

Abstract

Machine learning is being deployed in a growing number of applications which demand real-time, accurate, and robust predictions under heavy query load. However, most machine learning frameworks and systems only address model training and not deployment.

In this paper, we introduce Clipper, a general-purpose low-latency prediction serving system. Interposing between end-user applications and a wide range of machine learning frameworks, Clipper introduces a modular architecture to simplify model deployment across frameworks and applications. Furthermore, by introducing caching, batching, and adaptive model selection techniques, Clipper reduces prediction latency and improves prediction throughput, accuracy, and robustness without modifying the underlying machine learning frameworks. We evaluate Clipper on four common machine learning benchmark datasets and demonstrate its ability to meet the latency, accuracy, and throughput demands of online serving applications. Finally, we compare Clipper to the Tensorflow Serving system and demonstrate that we are able to achieve comparable throughput and latency while enabling model composition and online learning to improve accuracy and render more robust predictions.

1 Introduction

The past few years have seen an explosion of applications driven by machine learning, including recommendation systems [28, 60], voice assistants [18, 26, 55], and ad-targeting [3, 27]. These applications depend on two stages of machine learning: *training* and *inference*. Training is the process of building a model from data (e.g., movie ratings). Inference is the process of using the model to make a prediction given an input (e.g., predict a user's rating for a movie). While training is often computationally expensive, requiring multiple passes over potentially large datasets, inference is often assumed to be inexpensive. Conversely, while it is acceptable for training to take hours to days to complete, inference must run in real-time, often on orders of magnitude more queries than during training, and is typically part of user-facing applications.

For example, consider an online news organization that wants to deploy a content recommendation service to personalize the presentation of content. Ideally, the service should be able to recommend articles at interactive latencies (<100ms) [64], scale to large and growing user populations, sustain the throughput demands of flash crowds driven by breaking news, and provide accurate predictions as the news cycle and reader interests evolve.

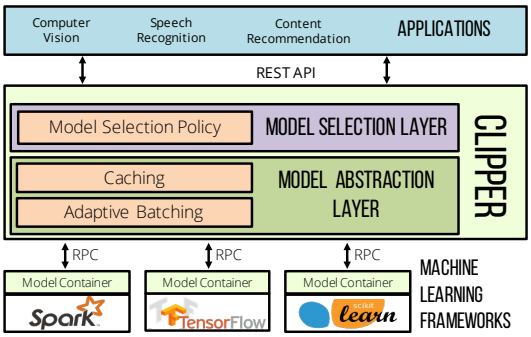


Figure 1: The Clipper Architecture.

The challenges of developing these services differ between the training and inference stages. On the training side, developers must choose from a bewildering array of machine learning frameworks with diverse APIs, models, algorithms, and hardware requirements. Furthermore, they may often need to migrate between models and frameworks as new, more accurate techniques are developed. Once trained, models must be *deployed* to a prediction serving system to provide low-latency predictions at scale.

Unlike model development, which is supported by sophisticated infrastructure, theory, and systems, model deployment and prediction-serving have received relatively little attention. Developers must cobble together the necessary pieces from various systems components, and must integrate and support inference across multiple, evolving frameworks, all while coping with ever-increasing demands for scalability and responsiveness. As a result, the deployment, optimization, and maintenance of machine learning services is difficult and error-prone.

To address these challenges, we propose Clipper, a *layered architecture* system (Figure 1) that reduces the complexity of implementing a prediction serving stack and achieves three crucial properties of a prediction serving system: *low latencies*, *high throughputs*, and *improved accuracy*. Clipper is divided into two layers: (1) the model abstraction layer, and (2) the model selection layer. The first layer exposes a common API that abstracts away the heterogeneity of existing ML frameworks and models.

Consequently, models can be modified or swapped transparently to the application. The model selection layer sits above the model abstraction layer and dynamically selects and combines predictions across competing models to provide more accurate and robust predictions.

To achieve low latency, high throughput predictions, Clipper implements a range of optimizations. In the model abstraction layer, Clipper caches predictions on a per-model basis and implements adaptive batching to maximize throughput given a query latency target. In the model selection layer, Clipper implements techniques to improve prediction accuracy and latency. To improve accuracy, Clipper exploits bandit and ensemble methods to robustly select and combine predictions from multiple models and estimate prediction uncertainty. In addition, Clipper is able to adapt the model selection independently for each user or session. To improve latency, the model selection layer adopts a straggler mitigation technique to render predictions without waiting for slow models. Because of this layered design, neither the end-user applications nor the underlying machine learning frameworks need to be modified to take advantage of these optimizations.

We implemented Clipper in Rust and added support for several of the most widely used machine learning frameworks: Apache Spark MLLib [40], Scikit-Learn [51], Caffe [31], TensorFlow [1], and HTK [63]. While these frameworks span multiple application domains, programming languages, and system requirements, each was added using fewer than 25 lines of code.

We evaluate Clipper using four common machine learning benchmark datasets and demonstrate that Clipper is able to render low and bounded latency predictions (<20ms), scale to many deployed models even across machines, quickly select and adapt the best combination of models, and dynamically trade-off accuracy and latency under heavy query load. We compare Clipper to the Google TensorFlow Serving system [59], an industrial grade prediction serving system tightly integrated with the TensorFlow training framework. We demonstrate that Clipper's modular design and broad functionality impose minimal performance cost, achieving comparable prediction throughput and latency to TensorFlow Serving while supporting substantially more functionality. In summary, our key contributions are:

- A layered architecture that abstracts away the complexity associated with serving predictions in existing machine learning frameworks (§3).
- A set of novel techniques to reduce and bound latency while maximizing throughput that generalize across machine learning frameworks (§4).
- A model selection layer that enables online model selection and composition to provide robust and accurate predictions for interactive applications (§5).

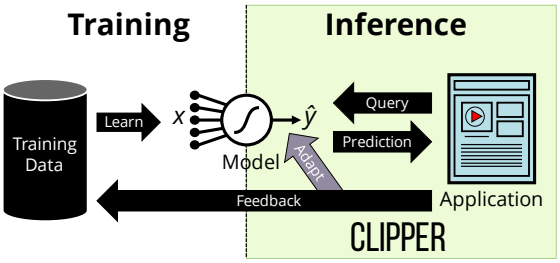


Figure 2: Machine Learning Lifecycle.

2 Applications and Challenges

The machine learning life-cycle (Figure 2) can be divided into two distinct phases: *training* and *inference*. Training is the process of estimating a model from data. Training is often computationally expensive requiring multiple passes over large datasets and can take hours or even days to complete [11, 29, 41]. Much of the innovation in systems for machine learning has focused on model training with the development of systems like Apache Spark [65], the Parameter Server [38], PowerGraph [25], and Adam [14].

A wide range of machine learning frameworks have been developed to address the challenges of training. Many specialize in particular models such as TensorFlow [1] for deep learning or Vowpal Wabbit [34] for large linear models. Others are specialized for specific application domains such as Caffe [31] for computer vision or HTK [63] for speech recognition. Typically, these frameworks leverage advances in parallel and distributed systems to scale the training process.

Inference is the process of evaluating a model to render predictions. In contrast to training, inference does not involve complex iterative algorithms and is therefore generally assumed to be easy. As a consequence, there is little research studying the process of inference and most machine learning frameworks provide only basic support for offline batch inference – often with the singular goal of evaluating the model training algorithm. However, scalable, accurate, and reliable inference presents fundamental system challenges that will likely dominate the challenges of training as machine learning adoption increases. In this paper we focus on the less studied but increasingly important challenges of *inference*.

2.1 Application Workloads

To illustrate the challenges of inference and provide a benchmark on which to evaluate Clipper, we describe two canonical real-world applications of machine learning: *object recognition* and *speech recognition*.

Object Recognition

Advances in deep learning have spurred rapid progress in computer vision, especially in object recognition prob-

lems – the task of identifying and labeling the objects in a picture. Object recognition models form an important building block in many computer vision applications ranging from image search to self-driving cars.

As users interact with these applications, they provide feedback about the accuracy of the predictions, either by explicitly labeling images (e.g., tagging a user in an image) or implicitly by indicating whether the provided prediction was correct (e.g., clicking on a suggested image in a search). Incorporating this feedback quickly can be essential to eliminating failing models and providing a more personalized experience for users.

Benchmark Applications: We use the well studied MNIST [35], CIFAR-10 [32], and ImageNet [49] datasets to evaluate increasingly difficult object recognition tasks with correspondingly larger inputs. For each dataset, the prediction task requires identifying the correct label for an image based on its pixel values. MNIST is a common baseline dataset used to demonstrate the potential of a new algorithm or technique, and both deep learning and more classical machine learning models perform well on MNIST. On the other hand, for CIFAR-10 and Imagenet, deep learning significantly outperforms other methods. By using three different datasets, we evaluate Clipper's performance when serving models that have a wide variety of computational requirements and accuracies.

Automatic Speech Recognition

Another successful application of machine learning is automatic speech recognition. A speech recognition model is a function from a spoken audio signal to the corresponding sequence of words. Speech recognition models can be relatively large [10] and are often composed of many complex sub-models trained using specialized speech recognition frameworks (e.g., HTK [63]). Speech recognition models are also often personalized to individual users to accommodate variations in dialect and accent.

In most applications, inference is done online as the user speaks. Providing real-time predictions is essential to user experience [4] and enables new applications like real-time translation [56]. However, inference in speech models can be costly [10] requiring the evaluation of large tensor products in convolutional neural networks.

As users interact with speech services, they provide implicit signal about the quality of the speech predictions which can be used to identify the dialect. Incorporating this feedback quickly improves user experience by allowing us to choose models specialized for a user's dialect.

Benchmark Application: To evaluate the benefit of personalization and online model-selection on a dataset with real user data, we built a speech recognition service with the widely used TIMIT speech corpus [24] and the HTK [63] machine learning framework. This dataset consists of voice recordings for 630 speakers in eight dialects of English. We randomly drew users from the test corpus and simulated their interaction with our speech recognition service using their pre-recorded speech data.

2.2 Challenges

Motivated by the above applications, we outline the key challenges of prediction serving and describe how Clipper addresses these challenges.

Complexity of Deploying Machine Learning

There is a large and growing number of machine learning frameworks [1,7,13,16,31]. Each framework has strengths and weaknesses and many are optimized for specific models or application domains (e.g., computer vision). Thus, there is no dominant framework and often multiple frameworks may be used for a single application (e.g., speech recognition and computer vision in automatic captioning). Furthermore, machine learning is an iterative process and the best framework may change as an application evolves over time (e.g., as a training dataset grows to require distributed model training). Although common model exchange formats have been proposed [47, 48], they have never achieved widespread adoption because of the rapid and fundamental changes in state-of-the-art techniques and additional source of errors from parallel implementations for training and serving. Finally, machine learning frameworks are often developed by and for machine learning experts and are therefore heavily optimized towards model development rather than deployment. As a consequence of these design decisions, application developers are forced to accept reduced accuracy by forgoing the use of a model well-suited to the task or to incur the substantially increased complexity of integrating and supporting multiple machine learning frameworks.

Solution: Clipper introduces a model abstraction layer and common prediction interface that isolates applications from variability in machine learning frameworks (§4) and simplifies the process of deploying a new model or framework to a running application.

Prediction Latency and Throughput

The *prediction latency* is the time it takes to render a prediction given a query. Because prediction serving is often on the critical path, predictions must both be fast and have bounded tail latencies to meet service level objectives [64]. While simple linear models are fast, more sophisticated and often more accurate models such as support vector machines, random forests, and deep neural networks are much more computationally intensive and can have substantial latencies (50-100ms) [13] (see Figure 11 for details). In many cases accuracy can be improved by combining models but at the expense of stragglers and increased tail latencies. Finally, most machine learning frameworks are optimized for offline batch processing and not single-input prediction latency. More-

over, the low and bounded latency demands of interactive applications are often at odds with the design goals of machine learning frameworks.

The computational cost of sophisticated models can substantially impact prediction throughput. For example, a relatively fast neural network which is able to render 100 predictions per second is still orders of magnitude slower than a modern web-server. While batching prediction requests can substantially improve throughput by exploiting optimized BLAS libraries, SIMD instructions, and GPU acceleration it can also adversely affect prediction latency. Finally, under heavy query load it is often preferable to marginally degrade accuracy rather than substantially increase latency or lose availability [3, 23].

Solution: Clipper automatically and adaptively batches prediction requests to maximize the use of batch-oriented system optimizations in machine learning frameworks while ensuring that prediction latency objectives are still met (§4.3). In addition, Clipper employs straggler mitigation techniques to reduce and bound tail latency, enabling model developers to experiment with complex models without affecting serving latency (§5.2.2).

Model Selection

Model development is an iterative process producing many models reflecting different feature representations, modeling assumptions, and machine learning frameworks. Typically developers must decide which of these models to deploy based on offline evaluation using stale datasets or engage in costly online A/B testing. When predictions can influence future queries (e.g., content recommendation), offline evaluation techniques can be heavily biased by previous modeling results. Alternatively, A/B testing techniques [2] have been shown to be statistically inefficient — requiring data to grow *exponentially* in the number of candidate models. The resulting choice of model is typically static and therefore susceptible to changes in model performance due to factors such as feature corruption or concept drift [52]. In some cases the best model may differ depending on the context (e.g., user or region) in which the query originated. Finally, predictions from more than one model can often be combined in ensembles to boost prediction accuracy and provide more robust predictions with confidence bounds.

Solution: Clipper leverages adaptive online model selection and ensembling techniques to incorporate feedback and automatically select and combine predictions from models that can span multiple machine learning frameworks.

2.3 Experimental Setup

Because we include microbenchmarks of many of Clipper's features as we introduce them, we present the experimental setup now. For each of the three object recognition benchmarks, the prediction task is predicting the correct label given the raw pixels of an unlabeled image as input. We used a variety of models on each of the object recognition benchmarks. For the speech recognition benchmark, the prediction task is predicting the phonetic transcription of the raw audio signal. For this benchmark, we used the HTK Speech Recognition Toolkit [63] to learn Hidden Markov Models whose outputs are sequences of phonemes representing the transcription of the sound. Details about each dataset are presented in Table 1.

Dataset	Type	Size	Features	Labels
MNIST [35]	Image	70K	28x28	10
CIFAR [32]	Image	60k	32x32x3	10
ImageNet [49]	Image	1.26M	299x299x3	1000
Speech [24]	Sound	6300	5 sec.	39

Table 1: Datasets. The collection of real-world benchmark datasets used in the experiments.

Unless otherwise noted, all experiments were conducted on a single server. All machines used in the experiments contain 2 Intel Haswell-EP CPUs and 256 GB of RAM running Ubuntu 14.04 on Linux 4.2.0. TensorFlow models were executed on a Nvidia Tesla K20c GPUs with 5 GB of GPU memory and 2496 cores. In the scaling experiment presented in Figure 6, the servers in the cluster were connected with both a 10Gbps and 1Gbps network. For each network, all the servers were located on the same switch. Both network configurations were investigated.

3 System Architecture

Clipper is divided into *model selection* and *model abstraction* layers (see Figure 1). The model abstraction layer is responsible for providing a common prediction interface, ensuring resource isolation, and optimizing the query workload for batch oriented machine learning frameworks. The model selection layer is responsible for dispatching queries to one or more models and combining their predictions based on feedback to improve accuracy, estimate uncertainty, and provide robust predictions.

Before presenting the detailed Clipper system design we first describe the path of a prediction request through the system. Applications issue prediction requests to Clipper through application facing REST or RPC APIs. Prediction requests are first processed by the model selection layer. Based on properties of the prediction request and recent feedback, the model selection layer dispatches the prediction request to one or more of the models through the model abstraction layer.

The model abstraction layer first checks the prediction cache for the query before assigning the query to an adaptive batching queue associated with the desired model. The adaptive batching queue constructs batches of queries that are tuned for the machine learning framework and model. A cross language RPC is used to send the

batch of queries to a model container hosting the model in its native machine learning framework. To simplify deployment, we host each model container in a separate Docker container. After evaluating the model on the batch of queries, the predictions are sent back to the model abstraction layer which populates the prediction cache and returns the results to the model selection layer. The model selection layer then combines one or more of the predictions to render a final prediction and confidence estimate. The prediction and confidence estimate are then returned to the end-user application.

Any feedback the application collects about the quality of the predictions is sent back to the model selection layer through the same application-facing REST/RPC interface. The model selection layer joins this feedback with the corresponding predictions to improve how it selects and combines future predictions.

We now present the model abstraction layer and the model selection layer in greater detail.

4 Model Abstraction Layer

The **Model Abstraction Layer** (Figure 1) provides a common interface across machine learning frameworks. It is composed of a prediction cache, an adaptive query-batching component, and a set of model containers connected to Clipper via a lightweight RPC system. This modular architecture enables caching and batching mechanisms to be shared across frameworks while also scaling to many concurrent models and simplifying the addition of new frameworks.

4.1 Overview

At the top of the model abstraction layer is the prediction cache (§4.2). The prediction caches provides a partial pre-materialization mechanism for frequent queries and accelerates the adaptive model selection techniques described in §5 by enabling efficient joins between recent predictions and feedback.

The batching component (§4.3) sits below the prediction cache and aggregates point queries into mini-batches that are dynamically resized for each model container to maximize throughput. Once a mini-batch is constructed for a given model it is dispatched via the RPC system to the container for evaluation.

Models deployed in Clipper are each encapsulated within their own lightweight container (§4.4), communicating with Clipper through an RPC mechanism that provides a uniform interface to Clipper and simplifies the deployment of new models. The lightweight RPC system minimizes the overhead of the container-based architecture and simplifies cross-language integration.

In the following sections we describe each of these components in greater detail and discuss some of the key algorithmic innovations associated with each.

4.2 Caching

For many applications (e.g., content recommendation), predictions concerning popular items are requested frequently. By maintaining a prediction cache, Clipper can serve these frequent queries without evaluating the model. This substantially reduces latency and system load by eliminating the additional cost of model evaluation.

In addition, caching in Clipper serves an important role in model selection (§5). To select models intelligently Clipper needs to join the original predictions with any feedback it receives. Since feedback is likely to return soon after predictions are rendered [39], even infrequent or unique queries can benefit from caching.

For example, even with a small ensemble of four models (a random forest, logistic regression model, and linear SVM trained in Scikit-Learn and a linear SVM trained in Spark), prediction caching increased feedback processing throughput in Clipper by 1.6x from roughly 6K to 11K observations per second.

The prediction cache acts as a function cache for the generic prediction function:

```
Predict(m: ModelId, x: X) -> y: Y
```

that takes a model id *m* along with the query *x* and computes the corresponding model prediction *y*. The cache exposes a simple non-blocking *request* and *fetch* API. When a prediction is needed, the *request* function is invoked which notifies the cache to compute the prediction if it is not already present and returns a boolean indicating whether the entry is in the cache. The *fetch* function checks the cache and returns the query result if present.

Clipper employs an LRU eviction policy for the prediction cache, using the standard CLOCK [17] cache eviction algorithm. With an adequately sized cache, frequent queries will not be evicted and the cache serves as a partial pre-materialization mechanism for hot items. However, because adaptive model selection occurs *above the cache* in Clipper, changes in predictions due to model selection do not invalidate cache entries.

4.3 Batching

The Clipper batching component transforms the concurrent stream of prediction queries received by Clipper into batches that more closely match the workload assumptions made by machine learning frameworks while simultaneously amortizing RPC and system overheads. Batching improves throughput and utilization of often costly physical resources such as GPUs, but it does so at the expense of increased latency by requiring all queries in the batch to complete before returning a single prediction.

We exploit an explicitly stated latency service level objective (SLO) to *increase latency* in exchange for substantially improved throughput. By allowing users to specify a latency objective, Clipper is able to tune batched query

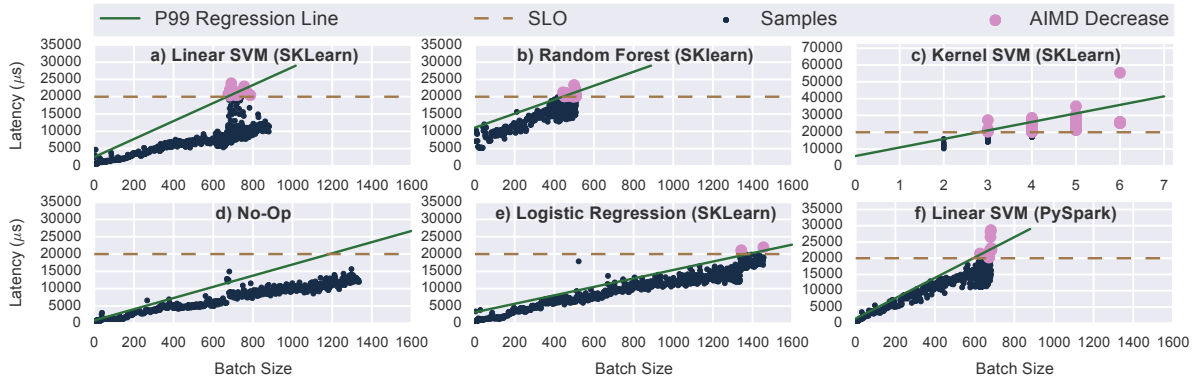


Figure 3: Model Container Latency Profiles. We measured the batching latency profile of several models trained on the MNIST benchmark dataset. The models were trained using Scikit-Learn (SKLearn) or Spark and were chosen to represent several of the most widely used types of models. The No-Op Container measures the system overhead of the model containers and RPC system.

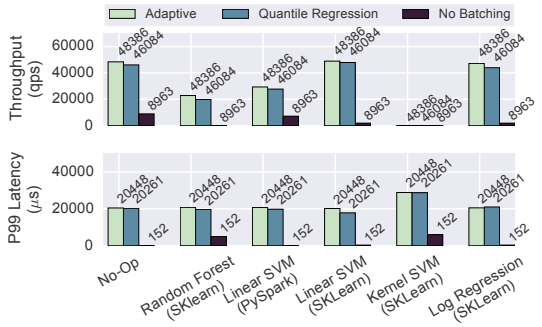


Figure 4: Comparison of Dynamic Batching Strategies.

evaluation to maximize throughput while still meeting the latency requirements of interactive applications. For example, requesting predictions in sufficiently large batches can improve throughput by up to 26x (the Scikit-Learn SVM in Figure 4) while meeting a 20ms latency SLO.

Batching increases throughput via two mechanisms. First, batching amortizes the cost of RPC calls and internal framework overheads such as copying inputs to GPU memory. Second, batching enables machine learning frameworks to exploit existing data-parallel optimizations by performing batch inference on many inputs simultaneously (e.g., by using the GPU or BLAS acceleration).

As the model selection layer dispatches queries for model evaluation, they are placed on queues associated with model containers. Each model container has its own adaptive batching queue tuned to the latency profile of that container and a corresponding thread to process predictions. Predictions are processed in batches by removing as many queries as possible from a queue up to the maximum batch size *for that model container* and sending the queries as a single batch prediction RPC to the container for evaluation. Clipper imposes a *maximum* batch size to ensure that latency objectives are met and avoid excessively delaying the first queries in the batch.

Frameworks that leverage GPU acceleration such as TensorFlow often enforce static batch sizes to maintain a consistent data layout across evaluations of the model. These frameworks typically encode the batch size directly into the model definition in order to fully exploit GPU parallelism. When rendering fewer predictions than the batch size, the input must be padded to reach the defined size, reducing model throughput without any improvement in prediction latency. Careful tuning of the batch size should be done to maximize inference performance, but this tuning must be done offline and is fixed by the time a model is deployed.

However, most machine learning frameworks can efficiently process variable-sized batches at serving time. Yet differences between the framework implementation and choice of model and inference algorithm can lead to orders of magnitude variation in model throughput and latency. As a result, the latency profile – the expected time to evaluate a batch of a given size – varies substantially between model containers. For example, in Figure 3 we see that the maximum batch size that can be executed within a 20ms latency SLO differs by 241x between the linear SVM which does a very simple vector-vector multiply to perform inference and the kernel SVM which must perform a sequence of expensive nearest-neighbor calculations to evaluate the kernel. As a consequence, the linear SVM can achieve throughput of nearly 30,000 qps while the kernel SVM is limited to 200 qps under this SLO. Instead of requiring application developers to manually tune the batch size for each new model, Clipper employs a simple adaptive batching scheme to dynamically find and adapt the maximum batch size.

4.3.1 Dynamic Batch Size

We define the optimal batch size as the batch size that maximizes throughput subject to the constraint that the batch evaluation latency is under the target SLO. To automati-

cally find the optimal maximum batch size for each model container we employ an additive-increase-multiplicative-decrease (AIMD) scheme. Under this scheme, we additively increase the batch size by a fixed amount until the latency to process a batch exceeds the latency objective. At this point, we perform a small multiplicative backoff, reducing the batch size by 10%. Because the optimal batch size does not fluctuate substantially, we use a much smaller backoff constant than other Additive-Increase, Multiplicative-Decrease schemes [15].

Early performance measurements (Figure 3) suggested a stable linear relationship between batch size and latency across several of the modeling frameworks. As a result, we also explored the use of quantile regression to estimate the 99th-percentile (P99) latency as a function of batch size and set the maximum batch size accordingly. We compared the two approaches on a range of commonly used Spark and Scikit-Learn models in Figure 4. Both strategies provide significant performance improvements over the baseline strategy of no batching, achieving up to a 26x throughput increase in the case of the Scikit-Learn linear SVM, demonstrating the performance gains that batching provides. While the two batching strategies perform nearly identically, the AIMD scheme is significantly simpler and easier to tune. Furthermore, the ongoing adaptivity of the AIMD strategy makes it robust to changes in throughput capacity of a model (e.g., during a garbage collection pause in Spark). As a result, Clipper employs the AIMD scheme as the default.

4.3.2 Delayed Batching

Under moderate or bursty loads, the batching queue may contain less queries than the maximum batch size when the next batch is ready to be dispatched. For some models, briefly delaying the dispatch to allow more queries to arrive can significantly improve throughput under bursty loads. Similar to the motivation for Nagle's algorithm [44], the gain in efficiency is a result of the ratio of the fixed cost for sending a batch to the variable cost of increasing the size of a batch.

In Figure 5, we compare the gain in efficiency (measured as increased throughput) from delayed batching for two models. Delayed batching provides no increase in throughput for the Spark SVM because Spark is already relatively efficient at processing small batch sizes and can keep up with the moderate serving workload using batches much smaller than the optimal batch size. In contrast, the Scikit-Learn SVM has a high fixed cost for processing a batch but employs BLAS libraries to do efficient parallel inference on many inputs at once. As a consequence, a 2ms batch delay provides a 3.3x improvement in throughput and allows the Scikit-Learn model container to keep up with the throughput demand while remaining well below the 10-20ms latency objectives needed for interactive applications.

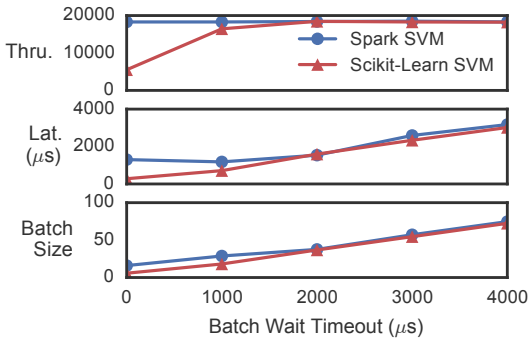


Figure 5: Throughput Increase from Delayed Batching.

```
interface Predictor<X,Y> {
  List<List<Y>> pred_batch(List<X> inputs);
}
```

Listing 1: Common Batch Prediction Interface for Model Containers. The batch prediction function is called via the RPC interface to compute the predictions for a batch of inputs. The return type is a nested list because each input may produce multiple outputs.

4.4 Model Containers

Model containers encapsulate the diversity of machine learning frameworks and model implementations within a uniform "narrow waist" remote prediction API. To add a new type of model to Clipper, model builders only need to implement the standard batch prediction interface in Listing 1. Clipper includes language specific container bindings for C++, Java, and Python. The model container implementations for most of the models in this paper only required a few lines of code.

To achieve process isolation, each model is managed in a separate Docker container. By placing models in separate containers, we ensure that variability in performance and stability of relatively immature state-of-the-art machine learning frameworks does not interfere with the overall availability of Clipper. Any state associated with a model, such as the model parameters, is provided to the container during initialization and the container itself is stateless after initialization. As a result, resource intensive machine learning frameworks can be replicated across multiple machines or given access to specialized hardware (e.g., GPUs) when needed to meet serving demand.

4.4.1 Container Replica Scaling

Clipper supports replicating model containers, both locally and across a cluster, to improve prediction throughput and leverage additional hardware accelerators. Because different replicas can have different performance characteristics, particularly when spread across a cluster, Clipper performs adaptive batching independently for

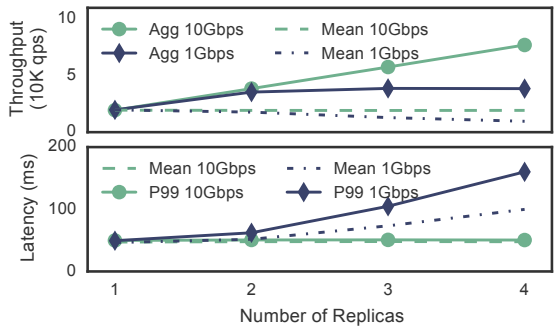


Figure 6: Scaling the Model Abstraction Layer Across a GPU Cluster. The solid lines refer to aggregate throughput of all the model replicas and the dashed lines refer to the mean per-replica throughput.

each replica.

In Figure 6 we demonstrate the linear throughput scaling that Clipper can achieve by replicating model containers across a cluster. With a four-node GPU cluster connected through a 10Gbps Ethernet switch, Clipper gets a 3.95x throughput increase from 19,500 qps when using a single model container running on a local GPU to 77,000 qps when using four replicas each running on a different machine. Because the model containers in this experiment are computationally intensive and run on the GPU, GPU throughput is the bottleneck and Clipper's RPC system can easily saturate the GPUs. However, when the cluster is connected through a 1Gbps switch, the aggregate throughput of the GPUs is higher than 1Gbps and so the network becomes saturated when replicating to a second remote machine. As machine-learning applications begin to consume increasingly bigger inputs, scaling from handcrafted features to large images, audio signals, or even video, the network will continue to be a bottleneck to scaling out prediction serving applications. This suggests the need for research into efficient networking strategies for remote predictions on large inputs.

5 Model Selection Layer

The **Model Selection Layer** uses feedback to dynamically select one or more of the deployed models and combine their outputs to provide more accurate and robust predictions. By allowing many candidate models to be deployed simultaneously and relying on feedback to adaptively determine the best model or combination of models, the model selection layer simplifies the deployment process for new models. By continuously learning from feedback throughout the lifetime of an application, the model selection layer automatically compensates for failing models without human intervention. By combining predictions from multiple models, the model selection layer boosts application accuracy and estimates prediction confidence.

There are a wide range of techniques for model selection and composition that span a tradeoff space of computational overhead and application accuracy. However, most of these techniques can be expressed with a simple *select*, *combine*, and *observe* API. We capture this API in the model selection policy interface (Listing 2) which governs the behavior of the model selection layer and allows users to introduce new model selection techniques themselves.

```
interface SelectionPolicy<S, X, Y> {
  S init();
  List<ModelId> select(S s, X x);
  pair<Y, double> combine(S s, X x,
    Map<ModelId, Y> pred);
  S observe(S s, X x, Y feedback,
    Map<ModelId, Y> pred);
}
```

Listing 2: Model Selection Policy Interface.

The model selection policy (Listing 2) defines four essential functions as well as a few basic types. In addition to the query and prediction types X and Y, the state type S encodes the learned state of the selection algorithm. The *init* function returns an initial instance of the selection policy state. We isolate the selection policy state and require an initialization function to enable Clipper to efficiently instantiate many instances of the selection policy for fine-grained contextualized model selection (§5.3). The *select* and *combine* functions are responsible for choosing which models to query and how to combine the results. In addition, the *combine* function can compute other information about the predictions. For example, in §5.2.1 we leverage the *combine* function to provide a prediction confidence score. Finally, the *observe* function is used to update the state S based on feedback from front-end applications.

In the current implementation of Clipper we provide two generic model selection policies based on robust bandit algorithms developed by Auer et al. [6]. These algorithms span a trade-off between computation overhead and accuracy. The single model selection policy (§5.1) leverages the Exp3 algorithm to optimally *select* the best model based on noisy feedback with minimal computational overhead. The ensemble model selection policy (§5.2) is based on the Exp4 algorithm which adaptively *combines* the predictions to improve prediction accuracy and estimate confidence at the expense of increased computational cost from evaluating all models for each query. By implementing model selection policies that provide different cost-accuracy tradeoffs, as well as an API for users to implement their own policies, Clipper provides a mechanism to easily navigate the tradeoffs between accuracy and computational cost on a per-application basis. Furthermore, users can modify this choice over time as application workloads evolve and resources become more or less constrained.

Framework	Model	Size (Layers)
Caffe	VGG [54]	13 Conv. and 3 FC
Caffe	GoogLeNet [57]	96 Conv. and 5 FC
Caffe	ResNet [29]	151 Conv. and 1 FC
Caffe	CaffeNet [22]	5 Conv. and 3 FC
TensorFlow	Inception [58]	6 Conv, 1 FC, & 3 Incept.

Table 2: Deep Learning Models. The set of deep learning models used to evaluate the ImageNet ensemble selection policy.

5.1 Single Model Selection Policy

We can cast the model-selection process as a multi-armed bandit problem [43]. The multi-armed bandit[1] problem refers the task of optimally choosing between k possible actions (e.g., models) each with a stochastic reward (e.g., feedback). Because only the reward for the *selected* action can be observed, solutions to the multi-armed bandit problem must address the trade-off between *exploring* possible actions and *exploiting* the estimated best action.

There are numerous algorithms for the multi-armed bandits problem with a wide range of trade-offs. In this work we first explore the use of the simple randomized Exp3 [6] algorithm which makes few assumptions about the problem setting and has strong optimality guarantees. The Exp3 algorithm associates a weight $s_i = 1$ for each of the k deployed models and then randomly selects model i with probability $p_i = s_i / \sum_{j=1}^{k} s_j$. For each prediction $\hat{y}$, Clipper observes a loss $L(y,\hat{y}) \in [0,1]$ with respect to the true value y (e.g., the fraction of words that were transcribed correctly during speech recognition). The Exp3 algorithm then updates the weight, $s_i \leftarrow s_i \exp(-\eta L(y,\hat{y})/p_i)$, corresponding to the selected model i. The constant η determines how quickly Clipper responds to recent feedback.

The Exp3 algorithm provides several benefits over manual experimentation and A/B testing, two common ways of performing model-selection in practice. Exp3 is both simple and robust, scaling well to model selection over a large number of models. It is a lightweight algorithm that requires only a single model evaluation for each prediction and thus performs well under heavy loads with negligible computational overhead. And Exp3 has strong theoretical guarantees that ensure it will quickly converge to an optimal solution.

5.2 Ensemble Model Selection Policies

It is a well-known result in machine learning [8,12,30,43] that prediction accuracy can be improved by combining predictions from multiple models. For example, bootstrap aggregation [9] (a.k.a., bagging) is used widely to reduce variance and thereby improve generalization performance. More recently, ensembles were used to win the Netflix challenge [53], and a carefully crafted ensemble of deep neural networks was used to achieve state-of-the-art accuracy on the speech recognition corpus Google uses to power their acoustic models [30]. The ensemble model selection policies adaptively combine the predictions from *all* available models to improve accuracy, rather than select individual models.

[1]The term bandits refers to pull-lever slot machines found in casinos.

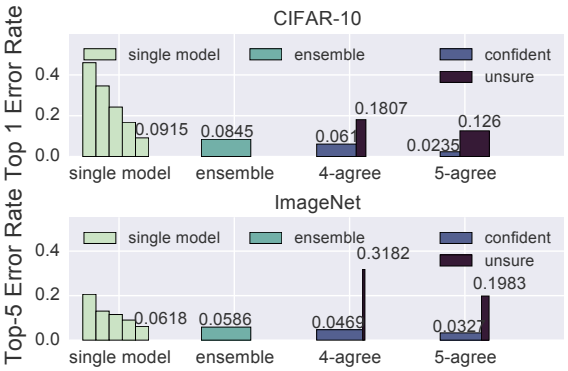


Figure 7: Ensemble Prediction Accuracy. The linear ensembles are composed of five computer vision models (Table 2) applied to the CIFAR and ImageNet benchmarks. The 4-agree and 5-agree groups correspond to ensemble predictions in which the queries have been separated by the ensemble prediction confidence (four or five models agree) and the width of each bar defines the proportion of examples in that category.

In Clipper we use linear ensemble methods which compute a weighted average of the base model predictions. In Figure 7, we show the prediction error rate of linear ensembles on two benchmarks. In both cases linear ensembles are able to marginally reduce the overall error rate. In the ImageNet benchmark, the ensemble formulation achieves a 5.2% relative reduction in the error rate simply by combining off-the-shelf models (Table 2). While this may seem small, on the difficult computer vision tasks for which these models are used, a lot of time and energy is spent trying to achieve even small reductions in error, and marginal improvements are considered significant [49].

There are many methods for estimating the ensemble weights including linear regression, boosting [43], and bandit formulations. We adopt the bandits approach and use the Exp4 algorithm [6] to learn the weights. Unlike Exp3, Exp4 constructs a weighted *combination* of all base model predictions and updates weights based on the individual model prediction error. Exp4 confers many of the same theoretical guarantees as Exp3. But while the accuracy when using Exp3 is bounded by the accuracy of the single best model, Exp4 can further improve prediction accuracy as the number of models increases. The extent to which accuracy increases depends on the relative accuracies of the set of base models, as well as the independence of their predictions. This increased accuracy comes at the cost of increased computational resources consumed by each prediction in order to evaluate all the base models.

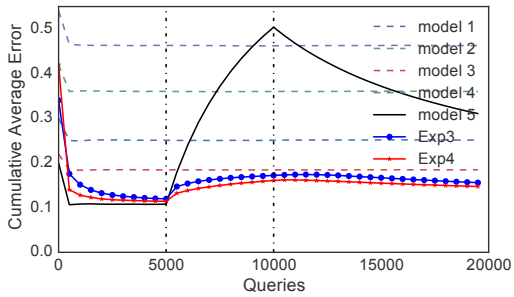


Figure 8: Behavior of Exp3 and Exp4 Under Model Failure. After 5K queries the performance of the lowest-error model is severely degraded, and after 10k queries performance recovers. Exp3 and Exp4 quickly compensate for the failure and achieve lower error than any static model selection.

The accuracy of a deployed model can silently degrade over time. Clipper's online selection policies can automatically detect these failures using feedback and compensate by switching to another model (Exp3) or down-weighting the failing model (Exp4). To evaluate how quickly and effectively the model selection policies react in the presence of changes in model accuracy, we simulated a severe model degradation while receiving real-time feedback. Using the CIFAR dataset we trained five different Caffe models with varying levels of accuracy to perform object recognition. During a simulated run of 20K sequential queries with immediate feedback, we degraded the accuracy of the best-performing model after 5K queries and then allowed the model to recover after 10K queries.

In Figure 8 we plot the cumulative average error rate for each of the five base models as well as the single (Exp3) and ensemble (Exp4) model selection policies. In the first 5K queries both model selection policies quickly converge to an error rate near the best performing model (model 5). When we degrade the predictions from model 5 its cumulative error rate spikes. The model selection policies are able to quickly mitigate the consequences of the increase in errors by learning to divert queries to the other models. When model 5 recovers after 10K queries the model selection policies also begin to improve by gradually sending queries back to model 5.

5.2.1 Robust Predictions

The advantages of online model selection go beyond detecting and mitigating model failures to leveraging new opportunities to improve application accuracy and performance. For many real-time decision-making applications, knowing the confidence of the prediction can significantly improve the end-user experience of the application.

For example, in many settings, applications have a sensible default action they can take when a prediction is unavailable. This is critical for building highly available applications that can survive partial system failures or when building applications where a mistake can be costly. Rather than blindly using all predictions regardless of the confidence in the result, applications can choose to only accept predictions above a confidence threshold by using the robust model selection policy. When the confidence in a prediction for a query falls below the confidence threshold, the application can instead use the sensible default decision for the query and avoid a costly mistake.

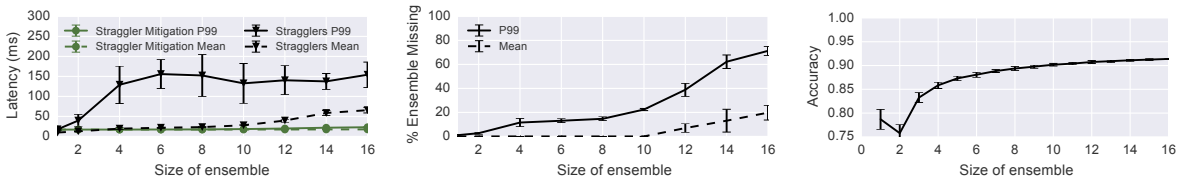
By evaluating predictions from multiple competing models concurrently we can obtain an estimator of the confidence in our predictions. In settings where models have high variance or are trained on random samples from the training data (e.g., bagging), agreement in model predictions is an indicator of prediction confidence. When evaluating the *combine* function in the ensemble selection policy we compute a measure of confidence by calculating the number of models that agree with the final prediction. End user applications can use this confidence score to decide whether to rely on the prediction. If we only consider predictions where multiple models agree, we can substantially reduce the error rate (see Figure 7) while declining to predict a small fraction of queries.

5.2.2 Straggler Mitigation

While the ensemble model selection policy can improve prediction accuracy and help quantify uncertainty, it introduces additional system costs. As we increase the size of the ensemble the computational cost of rendering a prediction increases. Fortunately, we can compensate for the increased prediction cost by scaling-out the model abstraction layer. Unfortunately, as we add model containers we increase the chance of stragglers adversely affecting tail latencies.

To evaluate the cost of stragglers, we deployed ensembles of increasing size and measured the resulting prediction latency (Figure 9a) under moderate query load. Even with small ensembles we observe the effect of stragglers on the P99 tail latency, which rise sharply to well beyond the 20ms latency objective. As the size of the ensemble increases and the system becomes more heavily loaded, stragglers begin to affect the mean latency.

To address stragglers, Clipper introduces a simple best-effort straggler-mitigation strategy motivated by the design choice that rendering a *late* prediction is worse than rendering an *inaccurate* prediction. For each query the model selection layer maintains a latency deadline determined by the latency SLO. At the latency deadline the *combine* function of the model selection policy is invoked with the *subset* of the predictions that are available. The model selection policy must render a final prediction using only the available base model predictions and communicate the potential loss in accuracy in its confidence score. Currently, we substitute missing predictions with their average value and define the confidence as the



(a) Latency **(b)** Missing Predictions **(c)** Accuracy

Figure 9: Increase in stragglers from bigger ensembles. The **(a)** latency, **(b)** percentage of missing predictions, and **(c)** prediction accuracy when using the ensemble model selection policy on SK-Learn Random Forest models applied to MNIST. As the size of an ensemble grows, the prediction accuracy increases but the latency cost of blocking until all predictions are available grows substantially. Instead, Clipper enforces bounded latency predictions and transforms the latency cost of waiting for stragglers into a reduction in accuracy from using a smaller ensemble.

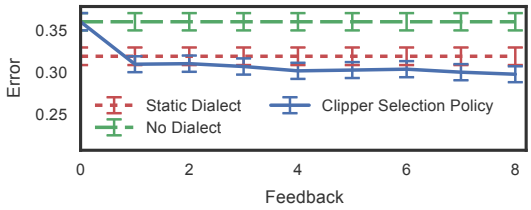


Figure 10: Personalized Model Selection. Accuracy of the ensemble selection policy on the speech recognition benchmark.

fraction of models that agree on the prediction.

The best-effort straggler-mitigation strategy prevents model container tail latencies from propagating to front-end applications by maintaining the latency objective as additional models are deployed. However, the straggler mitigation strategy reduces the size of the ensemble. In Figure 9b we plot the reduction in ensemble size and find that while tail latencies increase significantly with even small ensembles, most of the predictions arrive by the latency deadline. In Figure 9c we plot the effect of ensemble size on accuracy and observe that this ensemble can tolerate the loss of small numbers of component models with only a slight reduction in accuracy.

5.3 Contextualization

In many prediction tasks the accuracy of a particular model may depend heavily on context. For example, in speech recognition a model trained for one dialect may perform well for some users and poorly for others. However, selecting the right model or composition of models can be difficult and is best accomplished online in the model selection layer through feedback. To support context specific model selection, the model selection layer can be configured to instantiate a unique model selection state for each user, context, or session. The context specific session state is managed in an external database system. In our current implementation we use Redis.

To demonstrate the potential gains from personalized model selection we hosted a collection of TIMIT [24] voice recognition models each trained for a different dialect. We then evaluated (Figure 10) the prediction error rates using a single model trained across all dialects, the users' reported dialect model, and the Clipper ensemble selection policy. We first observe that the dialect-specific models out-perform the dialect-oblivious model, demonstrating the value of context to improve prediction accuracy. We also observe that the ensemble selection policy is able to quickly identify a combination of models that out-performs even the users' designated dialect model by using feedback from the serving workload.

6 System Comparison

In addition to the microbenchmarks presented in §4 and §5, we compared Clipper's performance to TensorFlow Serving and evaluate latency and throughput on three object recognition benchmarks.

TensorFlow Serving [59] is a recently released prediction serving system created by Google to accompany their TensorFlow machine learning training framework. Similar to Clipper, TensorFlow Serving is designed for serving machine learning models in production environments and provides a high-performance prediction API to simplify deploying new algorithms and experimenting with new models without modifying frontend applications. TensorFlow Serving supports general TensorFlow models with GPU acceleration through direct integration with the TensorFlow machine learning framework and tightly couples the model and serving components in the same process.

TensorFlow Serving also employs batching to accelerate prediction serving. Batch sizes in TensorFlow Serving are static and rely on a purely timeout based mechanism to avoid starvation. TensorFlow Serving does not explicitly incorporate prediction latency objectives which must be achieved by manually tuning the batch size. Furthermore, TensorFlow Serving was designed to serve one model at a time and therefore does not directly support feedback, dynamic model selection, or composition.

To better understand the performance overheads intro-

duced by Clipper's layered architecture and decoupled model containers, we compared the serving performance of Clipper and TensorFlow Serving on three TensorFlow object recognition deep networks of varying computational cost: a 4-layer convolutional neural network trained on the MNIST dataset [42], the 8-layer AlexNet [33] architecture trained on CIFAR-10 [32], and Google's 22-layer Inception-v3 network [58] trained on ImageNet. We implemented two Clipper model containers for each TensorFlow model, one that calls TensorFlow from the more standard and widely used Python API and one that calls TensorFlow from the more efficient C++ API. All models were run on a GPU using hand-tuned batch sizes (MNIST: 512, CIFAR: 128, ImageNet: 16) to maximize the throughput of TensorFlow Serving. The serving workload measured the maximum sustained throughput and corresponding prediction latency for each system.

Despite Clipper's modular design, we are able to achieve comparable throughput to TensorFlow Serving across all three models (Figure 11). The Python model containers suffer a 15-18% performance hit compared to the throughput of TensorFlow Serving, but the C++ model containers achieve nearly identical performance. This suggests that the high-level Python API for TensorFlow imposes a significant performance cost in the context of low-latency prediction-serving but that Clipper does not impose any additional performance degradation.

For these serving workloads, the throughput bottleneck is inference on the GPU. Both systems utilize additional queuing in order to saturate the GPU and therefore maximize throughput. For the Clipper model containers, we decomposed the prediction latency into component functions to demonstrate the overhead of the modular system design. The *predict* bar is the time spent performing inference within TensorFlow framework code. The *queue* bar is time spent queued within the model container waiting for the GPU to become available. The top bar includes the remaining system overhead, including query serialization and deserialization as well as copying into and out of the network stack. As Figure 11 illustrates, the RPC overheads are minimal on these workloads and the next prediction batch is queued as soon as the current batch is dispatched to the GPU for inference. TensorFlow Serving utilizes a similar queueing method to saturate the GPU, but because of the tight integration between TensorFlow Serving and the TensorFlow inference code, they are able to push the queueing into the TensorFlow framework code itself running in the same process.

By achieving comparable performance across this range of models, we have demonstrated that through careful design and implementation of the system, the modular architecture and substantially broader set of features in Clipper do not come at a cost of reduced performance on core prediction-serving tasks.

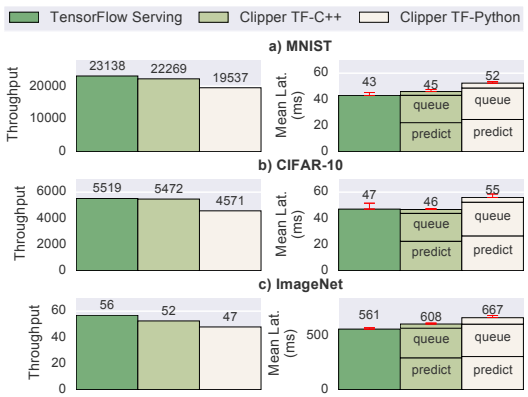


Figure 11: TensorFlow Serving Comparison. Comparison of peak throughput and latency (p99 latencies shown in error bars) on three TensorFlow models of varying inference cost. TF-C++ uses TensorFlow's C++ API and TF-Python the Python API.

7 Limitations

While Clipper attempts to address many challenges in the context of prediction serving there are a few key limitations when compared to other designs like TensorFlow Serving. Most of these limitations follow directly from the design of the Clipper architecture which assumes models are below Clipper in the software stack, and thus are treated as black-box components.

Clipper does not optimize the execution of the models within their respective machine learning frameworks. Slow models will remain slow when served from Clipper. In contrast, TensorFlow Serving is tightly integrated with model evaluation, and hence is able to leverage GPU acceleration and compilation techniques to speedup inference on models created with TensorFlow.

Similarly, Clipper does not manage the training or retraining of the base models within their respective frameworks. As a consequence, if all models are out-of-date or inaccurate Clipper will be unable to improve accuracy beyond what can be accomplished through ensembles.

8 Related Work

The closest projects to Clipper are LASER [3], Velox [19], and TensorFlow Serving [59]. The LASER system was developed at LinkedIn to support linear models for ad-targeting applications. Velox is a UC Berkeley research project to study personalized prediction serving with Apache Spark. TensorFlow Serving is the open-source prediction serving system developed by Google for TensorFlow models. In our experiments we only compare against TensorFlow Serving, because LASER is not publicly available, and the current prototype of Velox has very limited functionality.

All three systems propose mechanisms to address latency and throughput. Both LASER and Velox utilize

caching at various levels in their systems. In addition, LASER also uses a straggler mitigation strategy to address slow feature evaluation. Neither LASER or Velox discuss batching. Conversely, TensorFlow Serving does not employ caching and instead leverages batching and hardware acceleration to improve throughput.

LASER and Velox both exploit a form of model decomposition to incorporate feedback and context similar to the linear ensembles in Clipper. However, LASER does not incorporate feedback in real-time, Velox does not support bandits and neither system supports cross framework learning. Moreover, the techniques used for online learning and contextualization in both of these systems are captured in the more general Clipper selection policy. In contrast, TensorFlow Serving has no mechanism to achieve personalization or adapt to real-time feedback.

Finally, LASER, Velox, and TensorFlow Serving are all vertically integrated; they focused on serving predictions from a single model or framework. In contrast, Clipper supports a wide range of machine learning models and frameworks and simultaneously addresses latency, throughput, and accuracy in a single serving system.

Application Specific Prediction Serving: There has been considerable prior work in application and model specific prediction-serving. Much of this work has focused on content recommendation, including video-recommendation [20], ad-targeting [27, 39], and product-recommendations [37]. Outside of content recommendation, there has been recent success in speech recognition [36, 55] and internet-scale resource allocation [23]. While many of these applications require real-time predictions, the solutions described are highly application-specific and tightly coupled to the model and workload characteristics. As a consequence, much of this work solves the same systems challenges in different application areas. In contrast, Clipper is a general-purpose system capable of serving many of these applications.

Parameter Server: There has been considerable work in the learning systems community on parameter-servers [5, 21, 38, 62]. While parameter-servers do focus on reduced latency and caching, they do so in the context of *model training*. In particular they are a specialized type of key-value store used to coordinate updates to model parameters in a distributed training system. They are not typically used to serve predictions.

General Serving Systems: The high-performance serving architecture of Clipper draws from prior work on highly-concurrent serving systems [45, 46, 50, 61]. The division of functionality into vertical stages introduced by [61] is similar to the division of Clipper's architecture into independent layers. Notably, while the dominant cost in data-serving systems tends to be IO, in prediction serving it is computation. This changes both physical resource allocation and batching and latency-hiding strategies.

9 Conclusion

In this work we identified three key challenges of prediction serving: latency, throughput, and accuracy, and proposed a new layered architecture that addresses these challenges by interposing between end-user applications and existing machine learning frameworks.

As an instantiation of this architecture, we introduced the Clipper prediction serving system. Clipper isolates end-user applications from the variability and diversity in machine learning frameworks by providing a common prediction interface. As a consequence, new machine learning frameworks and models can be introduced without modifying end-user applications.

We addressed the challenges of prediction serving latency and throughput within the Clipper Model Abstraction layer. The model abstraction layer lifts caching and adaptive batching strategies above the machine learning frameworks to achieve up to a 26x improvement in throughput while maintaining strict bounds on tail latency and providing mechanisms to scale serving across a cluster. We addressed the challenges of accuracy in the Clipper Model Selection Layer. The model selection layer enables many models to be deployed concurrently and then dynamically selects and combines predictions from each model to render more robust, accurate, and contextualized predictions while mitigating the cost of stragglers.

We evaluated Clipper using four standard machine-learning benchmark datasets spanning computer vision and speech recognition applications. We demonstrated Clipper's capacity to bound latency, scale heavy workloads across nodes, and provide accurate, robust, and contextual predictions. We compared Clipper to Google's TensorFlow Serving system and achieved parity on throughput and latency performance, demonstrating that the modular container-based architecture and substantial additional functionality in Clipper can be achieved with minimal performance penalty.

Acknowledgments

We would like to thank Peter Bailis, Alexey Tumanov, Noah Fiedel, Chris Olston, our shepherd Mike Dahlin, and the anonymous reviewers for their feedback. This research is supported in part by DHS Award HSHQDC-16-3-00083, DOE Award SN10040 DE-SC0012463, NSF CISE Expeditions Award CCF-1139158, and gifts from Ant Financial, Amazon Web Services, CapitalOne, Ericsson, GE, Google, Huawei, Intel, IBM, Microsoft and VMware.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015. *Software available from tensorflow. org*.

[2] A. Agarwal, S. Bird, M. Cozowicz, L. Hoang, J. Langford, S. Lee, J. Li, D. Melamed, G. Oshri, O. Ribas, et al. A multiworld testing decision service. *arXiv preprint arXiv:1606.03966*, 2016.

[3] D. Agarwal, B. Long, J. Traupman, D. Xin, and L. Zhang. Laser: A scalable response prediction platform for online advertising. In *WSDM*, pages 173–182, 2014.

[4] S. Agarwal and J. R. Lorch. Matchmaking for online games and other latency-sensitive p2p systems. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 315–326. ACM, 2009.

[5] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *WSDM*, pages 123–132, 2012.

[6] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM J. Comput.*, 32(1):48–77, Jan. 2003.

[7] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, page 3. Austin, TX, 2010.

[8] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag, Secaucus, NJ, USA, 2006.

[9] L. Breiman. Bagging predictors. *Mach. Learn.*, 24(2):123–140, Aug. 1996.

[10] C. Chelba, D. Bikel, M. Shugrina, P. Nguyen, and S. Kumar. Large scale language modeling in automatic speech recognition. *arXiv preprint arXiv:1210.8440*, 2012.

[11] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting Distributed Synchronous SGD. *arXiv.org*, Apr. 2016.

[12] T. Chen and C. Guestrin. XGBoost: A Scalable Tree Boosting System. *arXiv.org*, Mar. 2016.

[13] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[14] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, pages 571–582, 2014.

[15] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Comput. Netw. ISDN Syst.*, 17(1):1–14, June 1989.

[16] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[17] F. J. Corbato. A paging experiment with the multics system. 1968.

[18] Microsoft Cortana. `https://www.microsoft.com/en-us/mobile/experiences/cortana/`.

[19] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. In *CIDR 2015*, 2015.

[20] J. Davidson, B. Liebald, J. Liu, P. Nandy, T. Van Vleet, U. Gargi, S. Gupta, Y. He, M. Lambert, B. Livingston, and D. Sampath. The YouTube video recommendation system. *RecSys*, pages 293–296, 2010.

[21] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. aurelio Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, pages 1223–1231. 2012.

[22] J. Donahue. Caffenet. `https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet`.

[23] A. Ganjam, F. Siddiqui, J. Zhan, X. Liu, I. Stoica, J. Jiang, V. Sekar, and H. Zhang. C3: Internet-Scale Control Plane for Video Quality Optimization. *NSDI '15*, pages 131–144, 2015.

[24] J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, D. S. Pallett, and N. L. Dahlgren. Darpa timit acoustic phonetic continuous speech corpus cdrom, 1993.

[25] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. OSDI, pages 17–30, 2012.

[26] Google Now. `https://www.google.com/landing/now/`.

[27] T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich. Web-Scale Bayesian Click-Through rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine. *ICML*, pages 13–20, 2010.

[28] h20. `http://www.h2o.ai`.

[29] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[30] G. Hinton, O. Vinyals, and J. Dean. Distilling the Knowledge in a Neural Network. *arXiv.org*, Mar. 2015.

[31] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pages 675–678. ACM, 2014.

[32] A. Krizhevsky and G. Hinton. Cifar-10 dataset. `https://www.cs.toronto.edu/~kriz/cifar.html`.

[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1097–1105, 2012.

[34] J. Langford, L. Li, and A. Strehl. Vowpal wabbit online learning project, 2007.

[35] Y. LeCun, C. Cortes, and C. J. Burges. MNIST handwritten digit database. 1998.

[36] X. Lei, A. W. Senior, A. Gruenstein, and J. Sorensen. Accurate and compact large vocabulary speech recognition on mobile devices. *INTERSPEECH*, pages 662–665, 2013.

[37] R. Lerallut, D. Gasselin, and N. Le Roux. Large-Scale Real-Time Product Recommendation at Criteo. In *RecSys*, pages 232–232, 2015.

[38] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, 2014.

[39] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, S. Chikkerur, D. Liu, M. Wattenberg, A. M. Hrafnkelsson, T. Boulos, and J. Kubica. Ad click prediction: a view from the trenches. In *KDD*, page 1222, 2013.

[40] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

[41] V. Mnih, N. Heess, A. Graves, et al. Recurrent models of visual attention. In *NIPS*, pages 2204–2212, 2014.

[42] Deep MNIST for Experts. `https://www.tensorflow.org/versions/r0.10/tutorials/mnist/pros/index.html`.

[43] K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[44] J. Nagle. Congestion control in ip/tcp internetworks. *SIGCOMM Comput. Commun. Rev.*, 14(4):11–17, Oct. 1984.

[45] nginx [engine x]. `http://nginx.org/en/`.

[46] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. *USENIX Annual Technical Conference, General Track*, pages 199–212, 1999.

[47] Portable Format for Analytics (PFA). `http://dmg.org/pfa/index.html`.

[48] PMML 4.2. `http://dmg.org/pmml/v4-2-1/GeneralStructure.html`.

[49] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[50] D. C. Schmidt. Pattern languages of program design. chapter Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching, pages 529–545. 1995.

[51] Scikit-Learn machine learning in python. `http://scikit-learn.org`.

[52] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison. Hidden Technical Debt in Machine Learning Systems. *NIPS*, 2015.

[53] J. Sill, G. Takács, L. Mackey, and D. Lin. Feature-weighted linear stacking, 2009.

[54] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[55] Apple Siri. `http://www.apple.com/ios/siri/`.

[56] Skype real time translator. `https://www.skype.com/en/features/skype-translator/`.

[57] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *CVPR*, pages 1–9, 2015.

[58] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. *arXiv preprint arXiv:1512.00567*, 2015.

[59] TensorFlow Serving. `https://tensorflow.github.io/serving`.

[60] Turi. `https://turi.com`.

[61] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *SOSP*, pages 230–243, 2001.

[62] E. P. Xing, Q. Ho, W. Dai, J.-K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. In *KDD*, pages 1335–1344. ACM, 2015.

[63] S. J. Young, G. Evermann, M. J. F. Gales, T. Hain, D. Kershaw, G. Moore, J. Odell, D. Ollason, D. Povey, V. Valtchev, and P. C. Woodland. *The HTK Book, version 3.4*. Cambridge University Engineering Department, Cambridge, UK, 2006.

[64] J.-M. Yun, Y. He, S. Elnikety, and S. Ren. Optimal aggregation policy for reducing tail latency of web search. In *SIGIR*, pages 63–72, 2015.

[65] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

SPECIAL ISSUE

Articles

DATA PARALLEL ALGORITHMS

Parallel computers with tens of thousands of processors are typically programmed in a data parallel style, as opposed to the control parallel style used in multiprocessing. The success of data parallel algorithms—even on problems that at first glance seem inherently serial—suggests that this style of programming has much wider applicability than was previously thought.

W. DANIEL HILLIS and GUY L. STEELE, JR.

In this article we describe a series of algorithms appropriate for fine-grained parallel computers with general communications. We call these algorithms *data parallel* algorithms because their parallelism comes from simultaneous operations across large sets of data, rather than from multiple threads of control. The intent is not so much to present new algorithms (most have been described earlier in other contexts), but rather to demonstrate a style of programming that is appropriate for a machine with tens of thousands or even millions of processors. The algorithms described all use $O(N)$ processors to solve problems of size N, typically in $O(\log N)$ time. Some of the examples solve problems that at first sight seem inherently serial, such as parsing strings and finding the end of a linked list. In many cases, we include actual run times measured on the 65,536-processor Connection Machine® system. A key feature of this hardware is a general-purpose communications network connecting the processors that frees the programmer from detailed concerns of the mapping between data and hardware.

MODEL OF THE MACHINE

Our model of a fine-grained parallel machine with general-purpose communication is based on the Connection Machine system [14]. Current Connection Machine systems have either 16,384 or 65,536 processors, each with 4,096 bits of memory. There is nothing special about these particular numbers, other than being powers of two, but they are indicative of the scale of the machine. (The model is more sensible with thousands rather than tens of processors.) The system has two parts: a front-end computer of the usual von Neumann style, and an array of Connection Machine processors. Each processor in the array has a small amount of local memory, and to the front end, the processor array looks like a memory. A typical front-end processor is a VAX® or a Symbolics 3600®.

The processor array is connected to the memory bus of the front end so that the local processor memories can be random accessed directly by the front end, one word at a time, just as if it were any other memory. This section of the memory on the front end, however, is "smart" in the sense that the front end can issue special commands that cause many parts of the memory to be operated upon simultaneously, or cause data to move around within the memory. The processor array therefore effectively extends the instruction set of the front-end processor to include instructions that operate on large

Connection Machine is a registered trademark of Thinking Machines Corporation.

VAX is a trademark of Digital Equipment Corporation.

Symbolics 3600 is a trademark of Symbolics, Inc.

amounts of data simultaneously. In this way, the processor array serves a function similar to a floating-point accelerator unit, except that it accelerates general parallel computation and not just floating-point arithmetic.

The control structure of a program running on a Connection Machine system is executed by the front end in the usual way. An important practical benefit of this approach is that the program is developed and executed within the already-familiar programming environment of the front end. The program can perform computations in the usual serial way on the front end and also issue commands to the processor array.

The processor array executes commands in SIMD fashion. There is a single instruction stream coming from the front end; these instructions act on multiple data items, on the order of one (or a few) per processor. Most instructions are executed conditionally: That is, each processor has state bits that determine which instructions the processor will execute. A processor whose state bit is set is said to be *selected*. The state bit is called the *context flag* because the set of selected processors is often referred to as the *context* within which instructions are executed. For example, the front end might arrange for all odd-numbered processors to have their context flags set, and even-numbered processors to have their context flags cleared; issuing an **ADD** instruction would then cause each of the selected processors (the odd-numbered ones) to add one word of local memory into another word. The deselected (even-numbered) processors would do nothing, and their local memories would remain unchanged.

Contexts may be saved in memory and later restored, with each processor saving or restoring its own bit in parallel with the others. There are a few instructions that are unconditional: They are executed by every processor regardless of the value of the context flag. Such instructions are required for saving and restoring contexts.

A context, or a set of values for all the context flags, represents a set: namely, a set of selected processors. Forming the intersection, union, or complement of such sets is simple and fast; it requires only a one-bit logical **AND, OR,** or **NOT** operation issued to the processors. Locally viewed, each processor performs just one logical operation on one or two single-bit operands; viewed globally, an operation on sets is performed. (On the current Connection Machine hardware, such an operation takes about a microsecond.)

The processors can individually perform all the usual operations on logical, integer, and floating-point operands: add, subtract, multiply, divide, compare, max, min, not, and, or, exclusive or, shift, square root, and so on. In addition, single values computed in the front end can be broadcast from the front end to all processors at once (essentially by including them as immediate data in the instruction stream).

A number of other computing systems have been constructed with the characteristics we have already described, namely, a large number of parallel processors, each of which has local memory and the ability to execute instructions of more or less the usual sort, as broadcast from a master controller. These include ILLIAC IV [8], the Goodyear MPP [3], the Non-Von [23]; and the ICL DAP [12]; among others [13]. There are two additional characteristics of the Connection Machine programming model, however, which distinguish it from these other systems: *general, pointer-based communication,* and *virtual processors.*

Previous parallel computing systems of this fine-grained SIMD style have restricted interprocessor communication to certain patterns wired into the hardware; typically this pattern is a two-dimensional rectangular grid, or a tree. The Connection Machine model allows any processor to communicate directly with any other processor in unit time, while other processors also communicate concurrently. Communication is implemented via a **SEND** instruction.

Within each processor, the **SEND** instruction takes two operands: One addresses—within the processor—the field that contains the data to be sent; the other addresses a processor pointer (i.e., the number of the processor to which the datum is to be sent and the destination field within that processor, into which the data will be placed). The communications system is very much like a postal system, where you can send a message to anyone else directly, provided you know the address, and where many letters can be routed at the same time. The **SEND** instruction can also be viewed as a parallel "store indirect" instruction that allows each processor to store anywhere in the entire memory, not just in its own local memory.

The **SEND** instruction can also take one additional operand that specifies what happens if two or more messages are sent to the same destination. The options are to deliver to the destination the sum, maximum, minimum, bitwise **AND**, or bitwise **OR** of the messages; to deliver one message and discard all others; or to produce an error.

From a global point of view, the **SEND** instruction performs something like an arbitrary permutation on an array of items, although it is actually more

general than a permutation because more than one item may be sent to the same destination. The pattern is not wired into the hardware, but is encoded as an array of pointers, and is completely under software control; the same encoding, once constructed, can be used over and over again to transfer data repeatedly in the same pattern. To implement a regular communication pattern, such as a two-dimensional grid, the pointers are typically computed when needed rather than stored.

The Connection Machine programming model is carefully abstracted from the details of the hardware that supports it, and, in particular, the number and size of its hardware processors. Programs are described in terms of virtual processors. In actual implementations, hardware processors are multiplexed as necessary to support this abstraction; indeed, the abstraction is supported at a very low level by a microcoded controller interposed between the front end and the processor array, so that the front end always deals in virtual processors.

The benefits of the virtual processor abstraction are twofold. The first is that the same program can be run unchanged on different sizes of the Connection Machine system, notwithstanding the linear trade-off between the number of hardware processors and execution time. For example, a program that requires 2^{16} virtual processors can run at top speed on a system with the same number of hardware processors, but it can also be executed on one-fourth that amount of hardware (2^{14} processors) at one-fourth the speed, provided it can fit into one-fourth the amount of memory as well.

The second benefit is that for many purposes the number of processors may be regarded as expandable rather than fixed, so that it becomes natural to write programs using the Lisp, Pascal, or C style of storage allocation rather than the Fortran style. By this we mean that there is a procedure one can call to allocate a "fresh" processor as if from thin air while the program is running. In Fortran, all storage is preallocated before program execution begins, whereas Lisp has the **cons** operation to allocate a new list cell (as well as other operations for constructing other objects); Pascal has the **new** operation; and C has the **malloc** function. In each case, a new object is allocated and a pointer to this new object is returned. Of course, in the underlying implementation, the address space (physical or virtual) is actually a fixed resource pool from which all such requests are satisfied, but the point is that the language supports the abstraction of newly created storage. In the Connection Machine model, one may similarly allocate fresh storage using the operation **processor-cons**; the difference is that the newly allocated storage comes with its own processor attached.

In the ensuing discussion, we shall assume that the size and number of processors are sufficient to allocate one processor for each data element in the problem being solved. This allows us to adopt a model of the machine in which the following are counted as unit-time operations:

- any conventional word-at-a-time operation;
- any such operation applied to all the data elements concurrently, or to some selected subset;
- any communications step that involves the broadcast of information to all data elements;
- any communications step that involves no more than a single message transmission from each data element.

For purposes of analysis, it is also often useful to treat **processor-cons** as a unit-time operation, although it may be implemented in terms of more primitive operations, as described in more detail on page 1176.

EXAMPLES OF PARALLEL PROGRAMMING

To show some of the possibilities of data-parallel programming, we present here several algorithms currently in use on the Connection Machine system. Most of these algorithms are not new: Some of the ideas represented here appear in the languages APL [11, 15] and FP [1], while others are based on algorithms designed for other parallel machines, in particular, the Ultracomputer [22], and still others have been developed by our coworkers on the Connection Machine [4, 5, 7, 9, 10, 19].

Beginning with some very simple examples to familiarize the reader with the model and the notation, we then proceed to more elaborate and less obvious examples.

Sum of an Array of Numbers

The sum of n numbers can be computed in time $O(\log n)$ by organizing the addends at the leaves of a binary tree and performing the sums at each level of the tree in parallel. There are several ways of organizing an array into a binary tree. Figure 1 illustrates one such method on an array of 16 elements named x_0 through x_{15}. In this algorithm, for purposes of simplicity, the number of elements to be summed is assumed to be an integral power of two. There are as many processors as elements, and the statement **for all** k **in parallel do** s **od** causes all processors to execute the same statement s in synchrony, but the variable k has a different value for each processor,

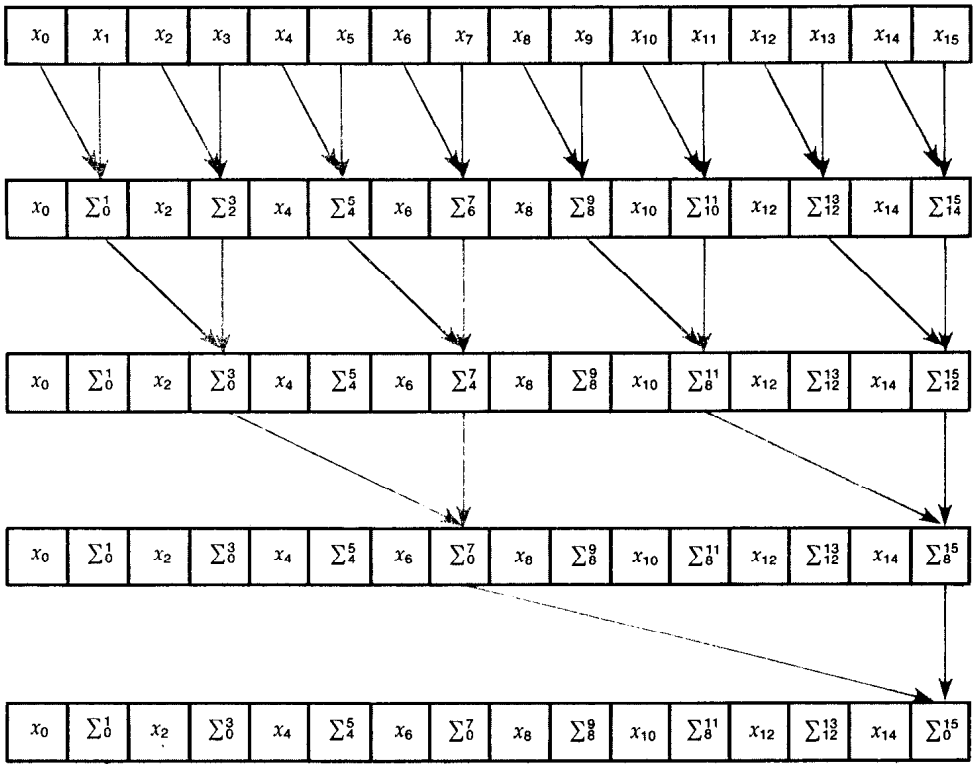


FIGURE 1. Computing the Sum of an Array of 16 Elements

namely, the index of that processor within the array.

```
for j := 1 to log2 n do
  for all k in parallel do
    if ((k + 1) mod 2^j) = 0 then
      x[k] := x[k − 2^(j−1)] + x[k]
    fi
  od
od
```

At the end of the process, x_{n-1} contains the sum of the n elements. On the Connection Machine, an optimized version of this algorithm for 65,536 elements takes about 200 microseconds.

All Partial Sums of an Array

A frequently useful operation is computing all partial sums of an array of numbers. In APL, this computation is called a plus-scan; in other contexts, it is called the "sum-prefix" operation because it computes sums over all prefixes of the array. For example, if you put into an array your initial checkbook balance, followed by the amounts of the checks you have written as negative numbers and deposits as positive numbers, then computing the partial sums produces all the intermediate and final balances.

It might seem that computing such partial sums is an inherently serial process, because one must add up the first k elements before adding in element $k + 1$. Indeed, with only one processor, one might as well do it that way, but with many processors one can do better, essentially because in $\log n$ time with n processors one can do $n \log n$ individual additions; serialization is avoided by performing logically redundant additions.

Looking again at the simple summation algorithm given on the facing page, we see that most of the processors are idle most of the time: During iteration j, only $n/2^j$ processors are active, and, indeed, half of the processors are never used. However, by putting the idle processors to good use by allowing more processors to operate, the summation algorithm can compute all partial sums of the array in the same amount of time it took to compute the single sum. In defining $\sum_j^k$ to mean $\sum_{i=j}^k x_i$, note that $\sum_j^k + \sum_{k+1}^m = \sum_j^m$. The partial-sums algorithm replaces each x_k by $\sum_0^k$: that is, the sum of all elements preceding and including x_k. In Figure 2 (on the following page), this process is illustrated for an array of 16 elements.

```
for j := 1 to log2 n do
  for all k in parallel do
    if k ≥ 2^j then
      x[k] := x[k − 2^(j−1)] + x[k]
    fi
  od
od
```

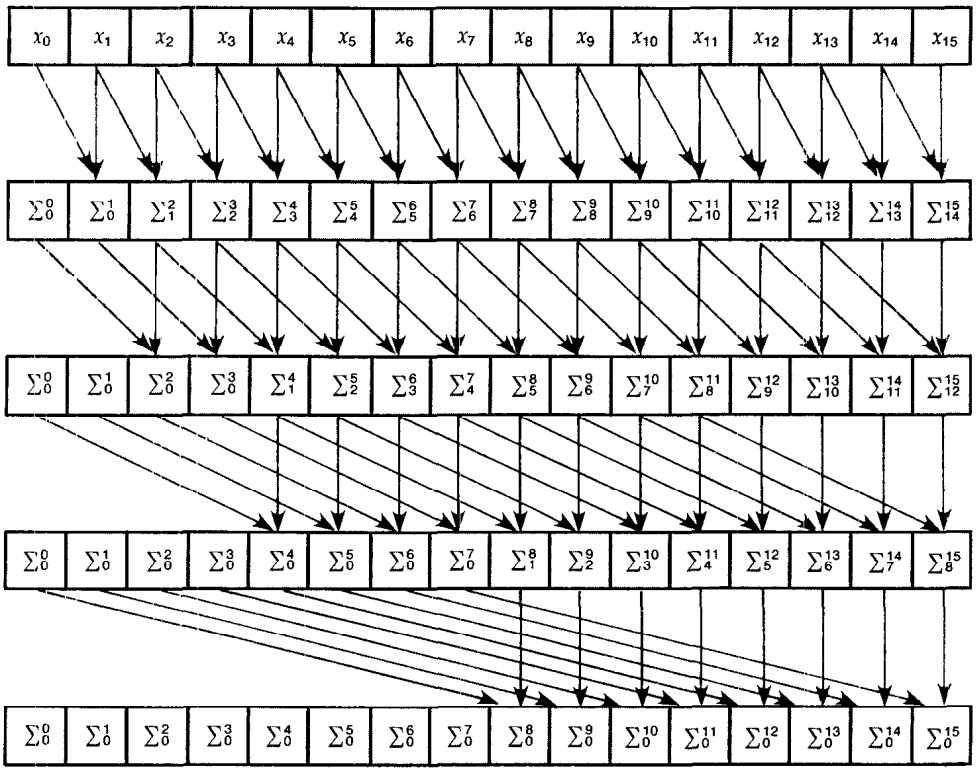


FIGURE 2. Computing Partial Sums of an Array of 16 Elements

The only difference between this algorithm and the earlier one is the test in the **if** statement in the partial-sums algorithm that determines whether a processor will perform the assignment. This algorithm keeps more processors active: During step j, $n - 2^{j-1}$ processors are in use; after step j, element number k has become $\sum_a^k$ where $a = \max(0, k - 2^j + 1)$.

This technique can be generalized from summation to any associative combining operation. Some obvious choices are product, maximum, minimum, and logical **AND**, **OR**, and **EXCLUSIVE OR**. Some programming languages provide such reduction and parallel-prefix operations on arrays as primitives. The current proposal for Fortran 8x, for example, provides reduction operations called **SUM**, **PRODUCT**, **MAXVAL**, **MINVAL**, **ANY**, and **ALL**. The corresponding reduction functions in APL are +/, ×/, ⌈/, ⌊/, ∨/, and ∧/; APL also provides other reduction operations and all the corresponding scan (prefix) operations. The combining functions for all these operations happen to be commutative as well, but the algorithm does not depend on commutativity. This was no accident; we took care to write

$$x[k] := x[k - 2^{j-1}] + x[k]$$

instead of the more usual

$$x[k] := x[k] + x[k - 2^{j-1}]$$

precisely in order to preserve the correctness of the algorithm when + is replaced by an associative but noncommutative operator. Under "Parsing a Regular Language" (facing page), we discuss the use of parallel-prefix computations with a noncommutative combining function for a nonnumerical application, specifically, the division of a character string into tokens. Another associative noncommutative operator of particular practical importance is matrix multiplication. We have found this technique useful in multiplying long chains of matrices.

Counting and Enumerating Active Processors

After some subset of the processors has been selected according to some condition (i.e., by using the result of a test to set the context flags), two operations are frequently useful: determining how many processors are active, and assigning a distinct integer to each processor. We call the first operation **count** and the second **enumerate**: Both are easily implemented in terms of summation and sum-prefix.

To **count** the active processors, we have every processor unconditionally examine its context flag and compute the integer 1 if the flag is set and 0 if it is clear. (Remember that an unconditional operation is performed by every processor regardless of whether or not its context flag is set.) We then perform an unconditional summation of these integer values.

To **enumerate** the active processors, we have every processor unconditionally compute a 1 or 0 in the same manner, but then we perform an unconditional sum-prefix calculation with the result that every processor receives a count of the number of active processors that precede it (including itself) in the ordering. We then revert to conditional operation; in effect, the selected processors have received distinct integers, and values computed for the deselected processors are henceforth simply ignored. Finally, it is technically convenient to have every selected processor subtract one from its value, so that the n selected processors will receive values from 0 to $n - 1$ rather than from 1 to n.

These operations each take about 200 microseconds on a Connection Machine of 65,536 elements. Because these operations are so fast, programming models of the Connection Machine have been suggested that treat counting and enumeration as unit-time operations [6, 7].

Radix Sort

Sorting is a communications-intensive operation. In parallel computers with fixed patterns of communication, the pattern of physical connections usually suggests the use of a particular sorting algorithm. For example, Batcher's bitonic sort [2] fits nicely on processors connected in a perfect shuffle pattern, bubble sorts [16] work well on one-dimensionally connected processing, and so on. In the Connection Machine model, the ability to access data in parallel in any pattern often eliminates the need to sort data. When it is necessary to sort, however, the generality of communications provides an embarrassment of riches.

Upon implementing several sorting algorithms on the Connection Machine and comparing them for speed, we found that, for the current hardware implementation, Batcher's method has good performance for large sort keys, whereas for small sort keys a version of radix sort is usually faster. (The break-even point is for keys about 25 to 32 bits in length. With either algorithm, sorting 65, 536 32-bit numbers on the Connection Machine takes about 30 milliseconds.)

To illustrate the use of **count** and **enumerate**, we present here the radix sort algorithm. In the interest of simplicity, we will assume that all processors (n) are active, that sort keys are unsigned integers, and that *maxint* is the value of the largest representable key value.

```
for j := 1 to 1 + ⌊log2 maxint⌋ do
  for all k in parallel do
    if (x[k] mod 2^j) < 2^(j-1) then
      comment The bit with weight 2^(j-1) is zero.
      tnemmoc
      y[k] := enumerate
      c := count
    fi
    if (x[k] mod 2^j) ≥ 2^(j-1) then
      comment The bit with weight 2^(j-1) is one.
      tnemmoc
      y[k] := enumerate + c
    fi
    x[y[k]] := x[k]
  od
od
```

At this point, an explanation of a fine point concerning the intended semantics of our algorithmic notation is in order. An **if** statement that is executed for all processors is always assumed to execute its **then** part, even if no processors are selected, because some front-end computations (such as **count**) might be included. When the **then** part is executed, the active processors are those previously active processors that computed **true** for the condition.

The radix sort requires a logarithmic number of passes, where each pass essentially examines one bit of each key. All keys that have a 0 in that bit are counted (call the count c) and then enumerated in order to assign them distinct integers y_k ranging from 0 to $c - 1$. All keys that have a 1 in that bit are then enumerated, and c is added to the result, thereby assigning these keys distinct integers y_k ranging from c to $n - 1$. The values y_k are then used to permute the keys so that all keys with a 0 bit precede all keys with a 1 bit. (This is the step that takes particular advantage of general communication.) This permutation is stable: The order of any two keys that both have 0 bits or both 1 bits is preserved. This stability property is important because the keys are sorted by least significant bit first and most significant bit last.

Parsing a Regular Language

To illustrate the use of a parallel-prefix computation in a nonnumerical application, consider the problem of parsing a regular language. For a concrete practical instance, let us consider the problem of breaking

up a long string of characters into tokens, which is usually the first thing a compiler does when processing a program. A string of characters such as

```
if x <=n then print("x = ", x);
```

must be broken up into the following tokens, with redundant white space eliminated:

This process is sometimes called *lexing a string.*

Any regular language of this type can be parsed by a finite-state automaton that begins in a certain state and makes a transition from one state to another (possibly the same one) as each character is read. Such an automaton can be represented as a two-dimensional array that maps the old state and the character read to the new state. Some of the states correspond to the start of a token; if the automaton is in one of those states just after reading a character, then that character is the first character of a token. Some characters may not be part of any token; White-space characters, for example, are typically not part of a token unless they occur within a string; such delimiting characters may also be identified by the automaton state just after the character is read. To divide a string up into tokens, then, means merely determining the state of the automaton after each character has been processed.

Table I shows the automaton array for a simple language in which a token may be one of three things: a sequence of alphabetic characters, a string surrounded by double quotes (where an embedded double quote is represented by two consecutive double quotes), or any of +, −, *, =, <, >, <=, and >=. Spaces and newlines delimit tokens, but are not part of any token except quoted strings. The automaton has nine states: *N* is the initial state; *A* is the start of an alphabetic token; Z is the continuation of an alphabetic token; * is a single-special-character token; < is a < or > character; = is an = that follows a < or > character (an = that does not follow < or > will produce state *); *Q* is the double quote that starts a string; *S* is a character within a string; and *E* is the double quote that ends a string, or the first of two that indicate an embedded double quote. The states *A*, *, <, and *Q* indicate that the character just read is the first character of a token.

Although, like the computation of partial sums, this may appear at first glance to be an inherently serial process, it too can be put into the form of a parallel-prefix computation. Rather than regarding the lexing automaton as a monolithic process, let us regard the individual characters of the string as unary functions that map an automaton state onto another state. By indicating the application of the character Y to state *N* as *N*Y, we may then write *N*Y = *A*. By extension, it is also possible to regard a string as a function that maps a state *p* to another state *q*; *q* is the state you end up in if you start the automaton in state *p* and then let the automaton read the entire string one character at a time. The result of applying the string Y″+ to the state Z may be written as ZY″+ = ((ZY)″)+ = (Z″)+ = *Q*+ = *S*. It is not too hard to see that the function corresponding to a string is simply the composition of the functions for the individual characters.

A function from a state to a state can be represented as a one-dimensional array indexed by states whose elements are states. The columns of the array in Table I are in fact exactly such representations for the functions for individual characters. Composing the columns for two characters or strings to produce a new column for the concatenation of the strings is fairly straightforward: You simply replace every entry of one column with the result of using that entry to index into the other column.

Since this composition operation is associative, we may compute the automaton state after every character in a string as follows:

1. Replace every character in the string with the array representation of its state-to-state function.
2. Perform a parallel-prefix operation. The combining function is the composition of arrays as described above. The net effect is that, after this step, every character *c* of the original string has been replaced by an array representing the state-to-state function for that prefix of the original string that ends at (and includes) *c*.
3. Use the initial automaton state (*N* in our example) to index into all these arrays. Now every character has been replaced by the state the automaton would have after that character.

If we implement this algorithm on a Connection Machine system and allot one processor per character, the first and third steps will take constant time, and the second step will take time logarithmic in the length of the string. Naturally, this algorithm performs much more computation per character than the straightforward serial algorithm using the two-dimensional array, but, for sufficiently large amounts of text, the parallel algorithm will be faster because its time complexity is logarithmic instead of linear. An implementation of this algorithm in Connection Machine Lisp can be found in [24].

PARALLEL PROCESSING OF POINTERS

Processor-cons

To illustrate pointer manipulation algorithms, we will consider the implementation of the **processor-cons** primitive, which allows a set of processors to establish pointers to a set of new processors allocated from free storage. In a serial computer, the equivalent problem is usually solved by keeping the free storage in an ordered list and allocating new storage from the beginning of the list. In the Connection Machine, this would not suffice since we wish to allocate many elements concurrently. Instead, the **processor-cons** primitive is implemented in terms of **enumerate** by using a rendezvous technique: Two sets of m processors are brought into one-to-one communication by using the processors numbered 0 through $m - 1$ as rendezvous points.

Assuming that every processor has a Boolean variable called *free*, $free_k$ becomes **true** if processor k is available for allocation and **false** otherwise. Every selected processor is to receive, in a variable called *new-processor*, the number of a distinct free processor. Free processors that are so allocated have their *free* bits reset to **false** in the process. If there are fewer free processors than selected processors, then as many requests are satisfied as possible, and some selected processors will have their *new-processor* variables set to *null* instead of the number of a free processor, as shown below.

```
for all k in parallel do
  required := count
  unconditionally
    if free[k] then
      available := count
      free-processor[k] := enumerate
      if free-processor[k] < required then
        free[k] := false
      fi
      rendezvous[free-processor[k]] := k
    requestor[k] := enumerate
    fi
  yllanoitidnocnu
  if requestor[k] < available then
    new-processor := rendezvous[requestor[k]]
  else
    new-processor := null
  fi
od
```

In this way, the total number of processors is managed as a finite resource, but with an interface that presents the illusion of creating new processors on demand. (Of course, we have not indicated how processors are returned to the pool of free processors. Some technique such as reference counting or garbage collection must also be designed and coded.) Other algorithms for **processor-cons** are described in [9, 10].

TABLE I. A Finite-State Automaton for Recognizing Tokens

Old State	Character Read													
•	A	B	...	Y	Z	+	−	*	<	>	=	"	Space	New line
N	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
A	Z	Z	...	Z	Z	*	*	*	<	<	*	Q	N	N
Z	Z	Z	...	Z	Z	*	*	*	<	<	*	Q	N	N
*	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
<	A	A	...	A	A	*	*	*	<	<	=	Q	N	N
=	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
Q	S	S	...	S	S	S	S	S	S	S	S	E	S	S
S	S	S	...	S	S	S	S	S	S	S	S	E	S	S
E	E	E	...	E	E	*	*	*	<	<	*	S	N	N

Parallel Combinator Reduction

A topic of much current interest in the area of functional programming is parallel combinator reduction [25]. It is also particularly interesting in this context because it shows how data parallel algorithms can be used to simulate control parallelism, or, equivalently, how SIMD machines with general communication can simulate MIMD machines.

Combinators are a way of encoding an applicative language. Their appeal lies in the fact that a program can be executed simply by performing successive local transformations on a tree structure, moreover, it is possible to perform many independent transformations simultaneously in the same tree. A combinator tree is made up of pairs, where each of the *left* and *right* components of a pair may point to another pair or else be an atom, the name of a combinator. Standard names for combinators include S, K, I, B, and C. Figure 3 (next page) shows one possible set of four transformations that suffices for program interpretation. When a subtree is transformed, the root pair of the subtree is used as the root pair of the result (by altering its components), but it is not permissible to alter any of the other pairs involved; therefore, the transformation involving the S combinator requires the allocation of fresh pairs. For our purposes, we ignore the semantics of the combinators and simply observe that such graph transformations can easily be carried out in parallel by a Connection Machine system by letting each processor contain one pair, and using **processor-cons** to allocate new pair-processors as needed.

```
while want or need to reduce some more do
  for all k in parallel do
    lf := left[k]
    if pair(lf) then
      if left[lf] = 'K' then
        left[k] := 'I'
      fi
      if left[lf] = 'I' then
        left[k] := right[lf]
      fi
      if pair(left[lf]) and left[left[lf]] = 'S' then
        p := processor-cons
        q := processor-cons
        if p ≠ null and q ≠ null then
          left[p] := right[left[lf]]
          right[p] := right[lf]
          left[q] := right[left[lf]]
          right[q] := right[k]
          left[k] := p
          right[k] := q
        fi
      fi
    fi
    rt := right[k]
    if pair(rt) and left[rt] = 'I' then
      right[k] := right[rt]
    fi
  od
  possibly perform garbage collection
od
```

It is easy to write such parallel code as a Connection Machine program. However, there are some difficult resource-management issues that have been glossed over, such as when a garbage collection should occur; whether, at any given time, one should prefer to reduce S combinators or K combinators; and, given that S combinators are to be reduced, which ones should be given preference. (The issues are that there may be more of one kind of combinator than the other at any instant. One idea is to process whichever kind is preponderant, but S combinators consume pairs and K combinators *may* release pairs, so the best processing strategy may need to take the number of free processors available for allocation into account. Furthermore, if there are not enough free processors to satisfy all outstanding S combinators at once, then the computation may diverge—even if normal-order serial reduction would converge—if preference is consistently given to the wrong ones.)

Finding the End of a Linked List

When we first began to work with pointer structures in the Connection Machine model, we believed that balanced trees would be important because information can be propagated from the root of a tree to its leaves—or from the leaves to the root—by parallel methods that take time logarithmic in the number of leaves. This was correct. However, our intuition also told us that linear linked lists would be useless. We could understand how to process an array in logarithmic time, because one can use address arithmetic to compute the number of any processor and then communicate with it directly, but it seemed to us that a linked list must be processed serially because in general one cannot compute the address of the ninth processor in a list without following all eight of the intervening pointers.

As is so often true in computer science, intuition was misleading: Essentially, we overlooked the power of having many processors working on the problem at once. It is true that one must follow all eight pointers, but by using many processors one can still achieve this in time logarithmic in the number of pointers to be traversed. Although we found this algorithm surprising, it had been discovered in other contexts several times before (e.g., see chapter 9 of [20]).

As a simple example, consider finding the last cell of a linearly linked list. Imagine each cell to have a *next* pointer that points to the next cell in the list,

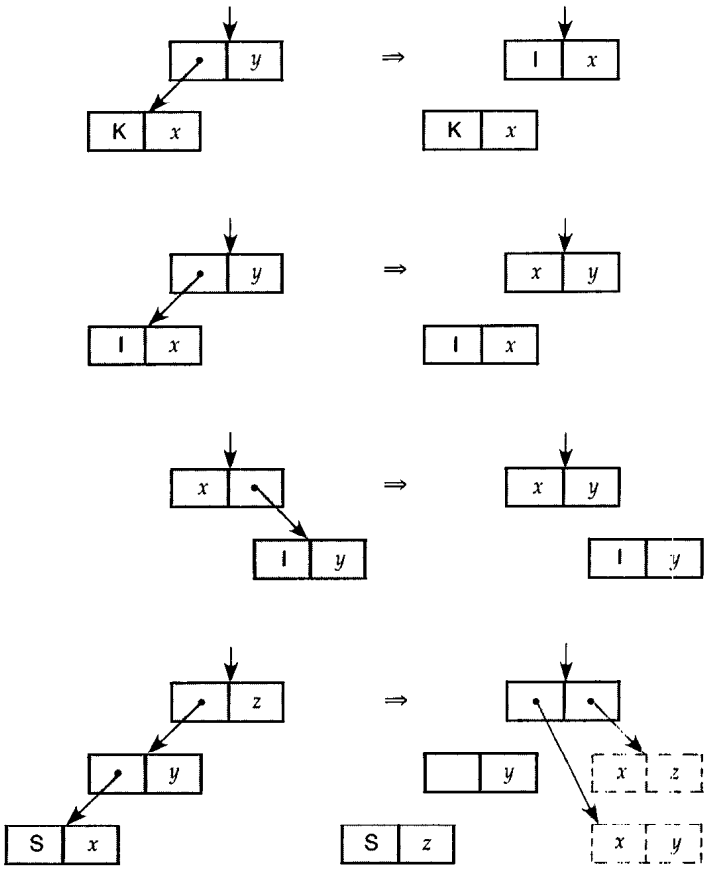


FIGURE 3. Patterns of Combinator Reduction

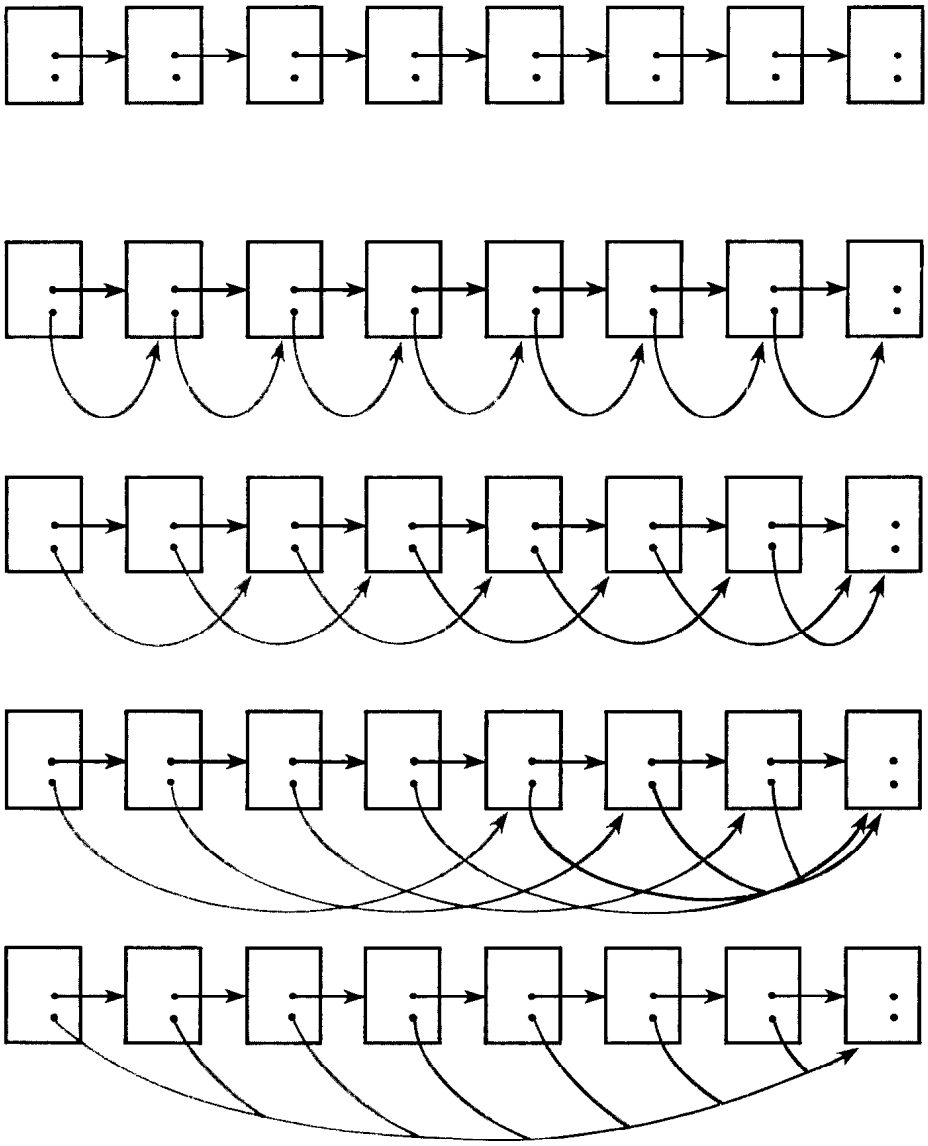


FIGURE 4. Finding the End of a Serially Linked List

while the last cell has the special value *null* in its *next* component. To accommodate other information as well, we will assume that in each cell there is another pointer component called *chum* that may be used for temporary purposes.

The basic idea is as follows: Each processor sets its *chum* component equal to a copy of its *next* component, so *chum* points to the next cell in the list. Each processor then repeatedly replaces its *chum* by its *chum's chum*. However, if its *chum* is *null*, then it remains *null*.) Initially, the *chum* of a processor is the next cell in the list; after the first step, its *chum* is the second cell following; after the second step, its *chum* is the fourth cell following; after the third step, its *chum* is the eighth cell following; and so on.

To ensure that the first cell of a list finds the last cell of a list, we carry out this procedure with the modification that a processor does not change its *chum* if its *chum's chum* is *null*, as shown below. The process is illustrated graphically in Figure 4.

```
for all k in parallel do
  chum[k] := next[k]
  while chum[k] ≠ null and chum[chum[k]] ≠ null do
    chum[k] := chum[chum[k]]
  od
od
```

The meaning of the **while** loop is that at each iteration a processor becomes deselected if it computes **false** for the test expression; the loop terminates when all processors have become deselected (whereupon the original context, as of the beginning of the loop, is restored). When this process terminates, *every* cell of the list except the last will have the last cell as its *chum*. If there are many lists in the machine, they can all be processed simultaneously,

and the total processing time will be proportional to the logarithm of the length of the longest such list.

All Partial Sums of a Linked List

The partial sums of a linked list may be computed by the same technique

```
for all k in parallel do
  chum[k] := next[k]
  while chum[k] ≠ null do
    value[chum[k]] := value[k] + value[chum[k]]
    chum[k] := chum[chum[k]]
  od
od
```

as illustrated in Figure 5. Comparing Figure 5 to Figure 2 (computing partial sums), we see that the same patterns of pointers among elements are constructed on the fly by using address arithmetic in the case of an array and by following pointer chains in the case of a linked list. An advantage of the linked-list representation is that it can simultaneously process many linked lists of different lengths without any change to the code.

Matching Up Elements of Two Linked Lists

An even more exotic effect is obtained by the following algorithm, which matches up corresponding elements in two linked lists. If we will call corresponding elements of a list "friends", this algorithm assigns to each list cell a pointer to its friend in the other list; of course, the result is obtained in logarithmic time.

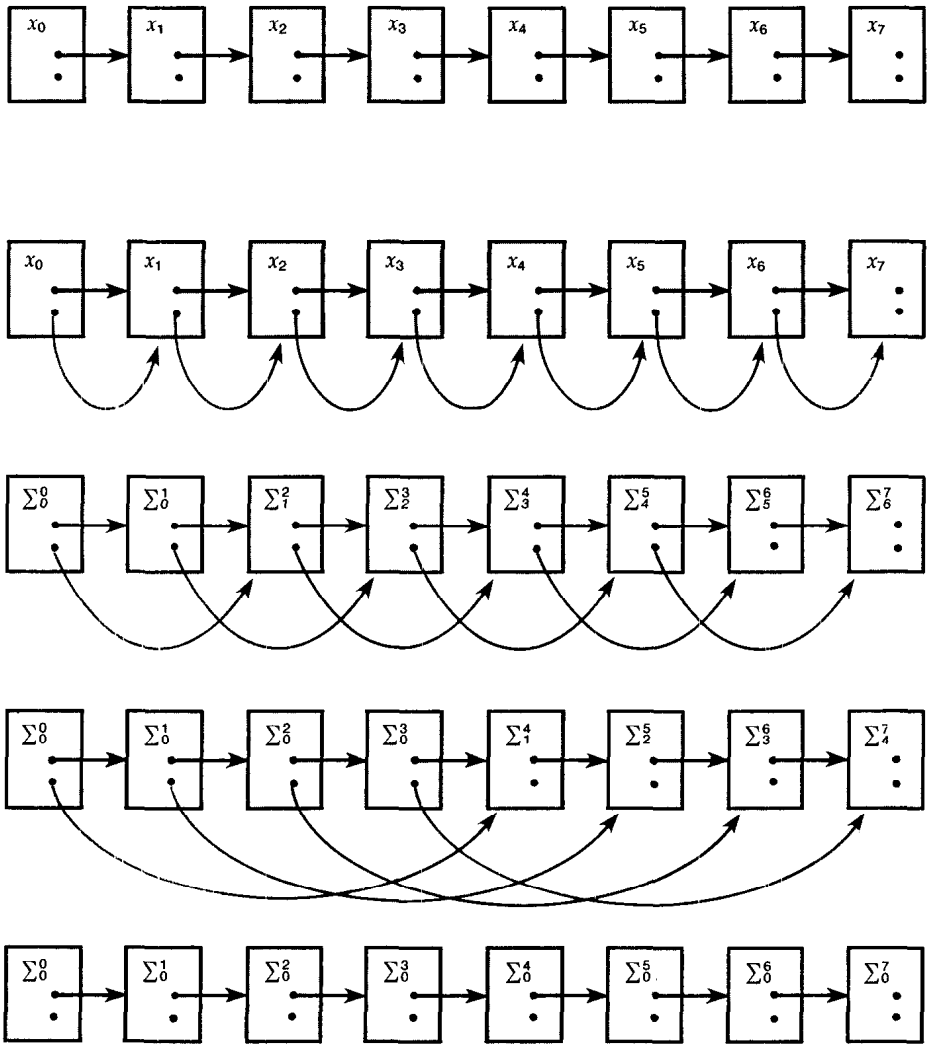


FIGURE 5. Computing Prefix Sums of a Serially Linked List

```
for all k in parallel do
  friend[k] := null
od
friend[list1] := list2
friend[list2] := list1
for all k in parallel
  chum[k] := next[k]
  while chum[k] ≠ null do
    if friend[k] ≠ null then
      friend[chum[k]] := chum[friend[k]]
    chum[k] := chum[chum[k]]
    fi
  od
od
```

The first part of the above algorithm is initialization: The component named *friend* is initialized to *null* in every cell; then the first cells of the two lists are introduced, so that they become *friends*. The second part plays the familiar logarithmic *chums* game, but at every iteration, a cell that has both a *chum* and a *friend* will cause its *friend's chum* to become its *chum's friend*. Believe it or not, when the dust has finally settled, the desired result does appear.

This algorithm has three additional interesting properties: First, it is possible to match up two lists of unequal length; the algorithm simply causes each extra cell at the end of the longer list to have no *friend* (that is, a *null friend*) (see Figure 6). Second, if, in the initialization, one makes the first cell of *list2* the *friend* of the first cell of *list1*, but not vice versa, then at the end all the cells of *list1* will have pointers to their *friends*, but the cells of *list2* are unaffected (their *friend* components remain *null*). Third, like the other linked-list algorithms, this one can process many lists (or pairs of lists) simultaneously.

With this primitive, one can efficiently perform such operations as componentwise addition of two vectors represented as linked lists.

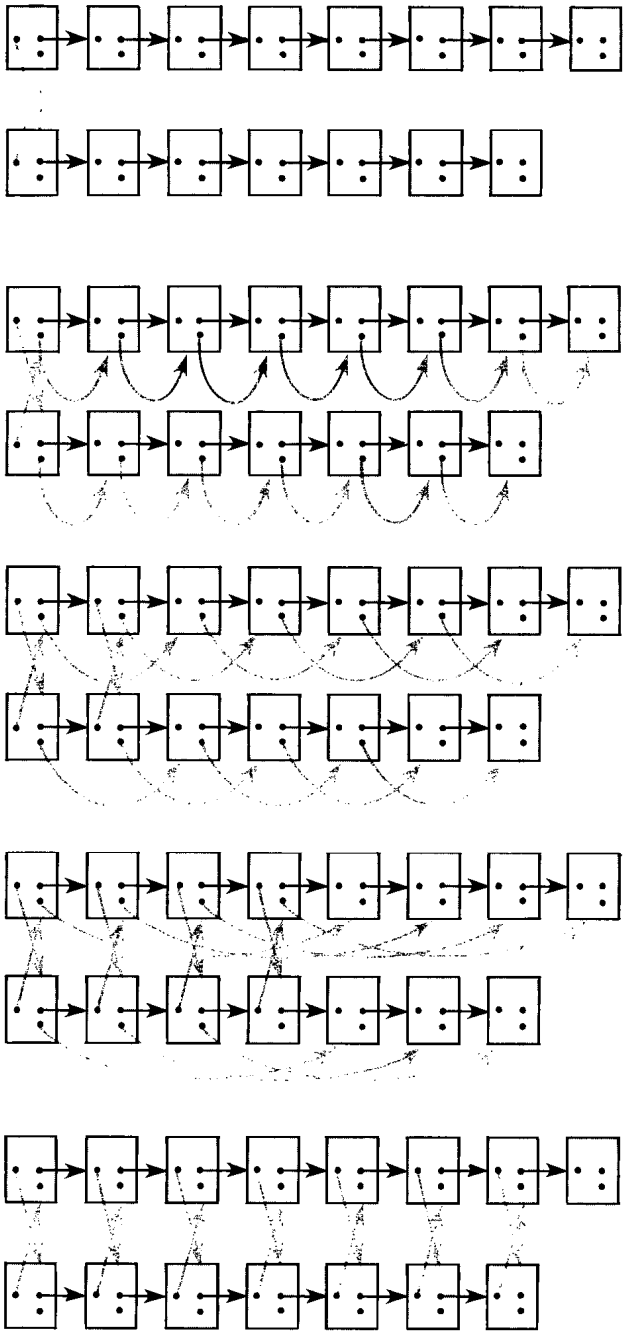


FIGURE 6. Matching Up Components of Two Lists

Region Labeling

How are linked-list operations used in practical applications? One such application is region labeling, where, given a two-dimensional image (a grid of pixel values), one must identify and uniquely label all regions. A *region* is a maximal set of pixels that are connected and all have the same value. Each region in the image must be assigned a different label, and this label should be distributed to every pixel in the region.

An obvious approach is to let each processor of a parallel computer hold one pixel. On a parallel computer with N processors communicating in a fixed two-dimensional pattern, so that each processor can communicate directly only with its four neighbors, this problem can be solved simply for an N-pixel image in the following manner: Since every processor has an address and knows its own address, a region will be labeled with the largest address of any processor in that region. To begin with, let each processor have a variable called *largest*, initialized to its own address, and then repeat the following step until there is no overall change of state. Each

processor trades *largest* values with all neighbors that have the same pixel value, and replaces its own *largest* value with the maximum of its previous value and any values received from neighbors. The address of the largest processor in a region therefore spreads out to fill the region.

Although the idea is simple, the algorithm takes time $O(\sqrt{N})$ in simple cases, and time $O(N)$ in the worst case (for images that contain a long "snake" that fills the picture). Lim [19] has devised algorithms for the Connection Machine that use linked-list techniques to solve the problem in time $O(\log N)$. In one of these algorithms, the basic idea is that every pixel can determine by communication with its two-dimensional neighbors whether it is on the boundary of its region, and, if so, which of its neighbors are also on the boundary. Each boundary pixel creates a pointer to each neighbor that is also a boundary pixel, and voilà: Every boundary has become a linked list (actually, a doubly linked list) of pixels. If the processor addresses are of the obvious form $x + Wy$, where x and y are two-dimensional coordinates and W is the width of the image, then the processor with the largest address for any region will be on its boundary. Using logarithmic-time linked-list algorithms, all the processors on a region boundary can agree on what the label for the region should be (by performing a maximum reduction on the linked list and then spreading the result back over the list). Since all the boundaries can be processed in parallel, it is then simply a matter of propagating the labels inward from boundaries to interior pixels. This is accomplished by a process similar to a parallel-prefix computation on the rows of the image. (There are many nasty details having to do with orienting the boundaries so that each boundary pixel knows which side is the interior and handling the possibility that regions may be nested within other regions, but these details can also be handled in logarithmic time.)

This application has the further property that the linked-list structure is not preexistent; rather, it is constructed dynamically as a function of the content of the image being processed. There is therefore no way to cleverly allocate or encode the structure ahead of time (e.g., as an array). The general communication facility of the Connection Machine model is therefore essential to the efficient execution of the algorithm.

Recursive Data Parallelism

We have often found situations where data parallelism can be applied recursively to achieve multiplicative effects. To multiply together a long chain of large matrices (a commonplace calculation in the study of systems modeled by Markov processes), we can use the associative scan operation to multiply together N matrices with $\log N$ matrix multiplications. In each matrix multiplication, the opportunity for parallelism is obvious, since matrix multiplication is defined in terms of operations on vectors. Another possibility would be to multiply the matrices using a systolic array-type algorithm [18], which will always run efficiently on a computer of the Connection Machine type. If the matrices are sparse, then we use the Pan–Reif algorithm [21], a data parallel algorithm that multiplies sparse matrices represented as trees. This algorithm fits well on a fine-grained parallel computer as long as it has capabilities for general communications. If the entries of the matrices contain high-precision numbers, there is yet another opportunity for parallelism within the arithmetic operations themselves. For example, using a scan-type algorithm for carry propagation, we can add two n-digit numbers in $O(\log n)$ time. Using a pipelined carry-save addition scheme [17], we can multiply in linear time, again by performing operations on all the data elements (digits) in parallel.

Summary and Conclusions

In discussing what kinds of computations are appropriate for data parallel algorithms, we initially assumed—when we began our work with the Connection Machine—that data parallel algorithms amounted to very regular calculations in simulation and search. Our current view of the applicability of data parallelism is somewhat broader. That is, we are beginning to suspect that this is an appropriate style wherever the amount of data to be operated upon is very large. Perhaps, in retrospect, this is a trivial observation in the sense that, if the number of lines of code is fixed and the amount of data is allowed to grow arbitrarily, then the ratio of code to data will necessarily approach zero. The parallelism to be gained by concurrently operating on multiple data elements will therefore be greater than the parallelism to be gained by concurrently executing lines of code.

One potentially productive line of research in this area is searching for counterexamples to this rule: that is, computations involving arbitrarily large data sets that can be more efficiently implemented in terms of control parallelism involving multiple streams of control. Several of the examples presented in this article first caught our attention as proposed counterexamples.

It is important to recognize that this question of

programming style is not synonymous with the hardware design issue of MIMD versus SIMD computers. MIMD computers can be well suited for executing data parallel programs: In fact, depending on engineering details like the cost of synchronization versus the cost of duplication, they may be the best means of executing data parallel programs. Similarly, SIMD computers with general communication can execute control-style parallelism by interpretation. Whether such interpretation is practical depends on the details of costs and requirements. While interesting and important in their own right, these questions are largely independent of the data parallel versus control parallel programming styles.

Having one processor per data element changes the way one thinks. We found that our serial intuitions did not always serve us well in parallel contexts. For example, when sorting is fast enough, the order in which things are stored is often unimportant. Then again, if searching is fast, then sorting may be unimportant. In a more general sense, it seems that the selection of one data representation over another is less critical on a highly parallel machine than on a conventional machine since converting all the memory from one representation to another does not take a large amount of time. One case where our serial intuitions misled us was our expectation that parallel machines would dictate the use of binary trees [14]. It turns out that linear linked lists serve almost as well, since they can be easily converted to balanced binary trees whenever necessary and are far more convenient for other purposes.

Our own transition from serial to parallel thinkers is far from complete, and we would be by no means surprised if some of the algorithms described in this article begin to look quite "old-fashioned" in the years to come.

REFERENCES

1. Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs (1977 ACM Turing Award Lecture). *Commun. ACM 21*, 8 (Aug. 1978), 613–641.
2. Batcher, K.E. Sorting networks and their applications. In *Proceedings of the 1968 Spring Joint Computer Conference* (Reston, Va., Apr.) AFIPS, Reston, Va., 1968, pp. 307–314.
3. Batcher, K.E. Design of a massively parallel processor. *IEEE Trans. Comput. C-29*, 9 (Sept. 1980), 836–840.
4. Bawden, A. A programming language for massively parallel computers. Master's thesis, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., Sept. 1984.
5. Bawden, A. Connection graphs. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*. ACM, (Cambridge, Mass., Aug. 4–6). New York, 1986, pp. 258–265.
6. Blelloch, G. AFL-I: A programming language for massively concurrent computers. Master's thesis, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., June 1986.
7. Blelloch, G. Parallel prefix versus concurrent memory access. Tech. Rep., Thinking Machines Corp., Cambridge, Mass., 1986.
8. Bouknight, W.J., Denenberg, S.A., McIntyre, D.E., Randall, J.M., Sameh, A.H., and Slotnick, D.L. The ILLIAC IV system. *Proc. IEEE 60*, 4 (Apr. 1972), 369–388.
9. Christman, D.P. Programming the Connection Machine. Master's thesis, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., Jan. 1983.
10. Christman, D.P. Programming the Connection Machine. Tech. Rep. ISL-84-3, Xerox Palo Alto Research Center, Palo Alto, Calif., Apr. 1984. (Reprint of the author's master's thesis at MIT.)
11. Falkoff, A.D., and Orth, D.L. Development of an APL standard. In *APL 79 Conference Proceedings* (Rochester, N.Y., June). ACM, New York, pp. 409–453. Published as *APL Quote Quad 9*, 4 (June 1979).
12. Flanders, P.M., et al. Efficient high speed computing with the distributed array processor. In *High Speed Computer and Algorithm Organization*, Kuch, Lawrie, and Sameh, Eds. Academic Press, New York, 1977, pp. 113–127.
13. Haynes, L.S., Lau, R.L., Siewiorek, D.P., and Mizell, D.W. A survey of highly parallel computing. *Computer* (Jan. 1982), 9–24.
14. Hillis, W.D. *The Connection Machine*. MIT Press, Cambridge, Mass., 1985.
15. Iverson, K.E. *A Programming Language*. Wiley, New York, 1962.
16. Knuth, D.E. *The Art of Computer Programming*. Vol. 3, *Sorting and Searching*. Addison-Wesley, Reading, Mass., 1973.
17. Knuth, D. E. *The Art of Computer Programming*. Vol. 2, *Seminumerical Algorithms (Second Edition)*. Addison-Wesley, Reading, Mass., 1981.
18. Kung, H.T., and Lieserson, C.E. Algorithms for VLSI processor arrays. In *Introduction to VLSI Systems*, L. Carver and L. Conway, Eds. Addison-Wesley, New York, 1980, pp. 271–292.
19. Lim, W. Fast algorithms for labeling connected components in 2-D arrays. Tech. Rep. 86.22, Thinking Machines Corp., Cambridge, Mass., July 1986.
20. Minsky, M., and Papert, S. *Perceptrons*. 2nd ed. MIT Press, Cambridge, Mass., 1979.
21. Pan, V., and Reif, J. Efficient parallel solution of linear systems. Tech. Rep. TR-02-85, Aiken Computation Laboratory, Harvard Univ., Cambridge, Mass., 1985.
22. Schwartz, J.T. Ultracomputers. *ACM Trans. Program. Lang. Syst. 2*, 4 (Oct. 1980), 484–521.
23. Shaw, D.E. *The NON-VON Supercomputer*. Tech. Rep., Dept. of Computer Science, Columbia Univ., New York, Aug. 1982.
24. Steele, G.L., Jr., and Hillis, W.D. Connection machine Lisp: Fine-grained parallel symbolic processing. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming* (Cambridge, Mass., Aug. 4–6). ACM, New York, 1986, pp. 279–297.
25. Turner, D.A. A new implementation technique for applicative languages. *Softw. Pract. Exper. 9* (1979), 31–49.

CR Categories and Subject Descriptors: B.2.1 [**Arithmetic and Logic Structures**]: Design Styles—*parallel*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*parallel processors*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs—*concurrent programming structures*; E.2 [**Data Storage Representations**]: *linked representations*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism*; G.1.0 [**Numerical Analysis**]: General—*parallel algorithms*

General Terms: Algorithms

Additional Key Words and Phrases: Combinator reduction, combinators, Connection Machine computer system, log-linked lists, parallel prefix, SIMD, sorting, Ultracomputer

Authors' Present Address: W. Daniel Hillis and Guy L. Steele, Jr., Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142-1214.

Efficient Parallel Scan Algorithms for GPUs

Shubhabrata Sengupta
University of California, Davis

Mark Harris Michael Garland
NVIDIA Corporation

Abstract

Scan and segmented scan algorithms are crucial building blocks for a great many data-parallel algorithms. Segmented scan and related primitives also provide the necessary support for the flattening transform, which allows for nested data-parallel programs to be compiled into flat data-parallel languages. In this paper, we describe the design of efficient scan and segmented scan parallel primitives in CUDA for execution on GPUs. Our algorithms are designed using a divide-and-conquer approach that builds all scan primitives on top of a set of primitive intra-warp scan routines. We demonstrate that this design methodology results in routines that are simple, highly efficient, and free of irregular access patterns that lead to memory bank conflicts. These algorithms form the basis for current and upcoming releases of the widely used CUDPP library.

1 Introduction

Parallel scan and segmented scan operations are data-parallel primitives whose broad importance is well known. Sequence compaction, radix sort, quicksort, sparse-matrix vector multiplication, and minimium spanning tree construction are only a few of the many algorithms that can be efficiently implemented in terms of scan operations [1, 2]. These operations are the analogs of parallel prefix circuits [13], which have a long history, and have been widely used in collection-oriented languages dating back to APL [12]. They also form the basis for efficiently mapping nested data-parallel languages such as NESL [3] on to flat data-parallel machines.

Because of their fundamental importance, scan operations have been implemented for many parallel systems. The earliest GPU implementations of scan were built using graphics pixel shaders for "non-uniform stream compaction" [11] and summed-area table generation [9]. Sengupta *et al.* [19] adapted Blelloch's work-efficient algorithm [1] to pixel shaders, and Harris *et al.* [8] built on this work to produce the first CUDA-based implementation of scan. Sengupta *et al.* [18] developed the first CUDA implementation of segmented scan by extending the reduce and down-sweep phases of their earlier work, and Dotsenko *et al.* [7] recently adapted the algorithm used by Chatterjee *et al.* [5] for the Cray Y-MP to CUDA.

The model of CUDA programs as a hierarchy of threads, thread blocks, and grids of blocks [15, 16] naturally encourages a hierarchical view of scan algorithms. We can think of a parallel scan kernel as performing some amount of (1) sequential work per thread, (2) parallel combination of results across a thread block, and (3) parallel combination of results between blocks. In this paper, we

NVIDIA Technical Report NVR-2008-003, Dec. 2008.

```
template<class OP, class T>
T scan(T *values, unsigned int n)
{
    for(unsigned int i=1; i<n; ++i)
        values[i] = OP::apply(values[i-1] , values[i]);
}
```

Figure 1. Serial implementation of inclusive scan for generic operator `OP` over values of type `T`.

discuss the design of efficient algorithms for the parallel combination of results across a block of threads. Our block-level algorithms simplify earlier methods [8, 18] considerably, avoid all memory bank conflicts, require no artificial padding, and are substantially more efficient. We retain the per-thread and inter-block methods used by Sengupta *et al.* [18], although expanding the amount of serial work performed per thread is another possible avenue for improving performance [7].

The algorithms outlined in this paper form the core of the scan and segmented scan primitives provided in the most recent release of the widely used CUDPP library [6]. For clarity, we present simplified versions of the kernels in the text, but the full kernels used in our performance profiling are available for download in the library.

2 Parallel Scan Operations

Given an input sequence a and an associative[1] binary operator $\oplus$ with identity I, an *inclusive* scan produces an output sequence $b = \text{scan}\langle\textit{inclusive}\rangle(a, \oplus)$ where $b_i = a_0 \oplus \cdots \oplus a_i$. Similarly, an *exclusive* scan produces an output sequence $b = \text{scan}\langle\textit{exclusive}\rangle(a, \oplus)$ where $b_i = I \oplus \cdots \oplus a_{i-1}$. As a concrete example, consider the input sequence:

```
a = [ 3 1  7  0  4  1  6  3 ]
```

Applying an inclusive scan operation to this array with the usual addition operator produces the result

```
scan<inclusive>(a, +) = [ 3 4 11 11 15 16 22 25 ]
```

and the exclusive scan operation produces the result

```
scan<exclusive>(a, +) = [ 0 3  4 11 11 15 16 22 ]
```

As always, the first element in the result produced by the exclusive scan is the identity element, which in this case is 0.

Implementing scan primitives on a serial processor is trivial, as shown in Figure 1. Note that throughout this paper, we use C++ templates to make scan generic over the operator `OP` and the type of values `T`.

Implementing parallel scan primitives requires somewhat more effort than the trivial serial case. Figure 2 shows a simple implementation of a well-known parallel scan algorithm [4, 10]. This code will scan the binary operator `OP` across an array of $n = 2^k$ values using a single thread block of 2^k

[1] In practice, this requirement is often relaxed to include pseudo-associative operations, such as addition of floating point numbers.

```
template<class OP, class T>
__device__ T scan(T *values)
{
    unsigned int i = threadIdx.x;  // ID of this thread
    unsigned int n = blockDim.x;   // number of threads in block

    for(unsigned int offset=1; offset<n; offset *= 2)
    {
        T t;

        if(i>=offset)  t = values[i-offset];
        __syncthreads();

        if(i>=offset)  values[i] = OP::apply(t , values[i]);
        __syncthreads();
    }
}
```

Figure 2. Simple parallel scan of 2^k elements with a single thread block of 2^k threads.

threads. We make these assumptions—that the number of values is a power of 2 and that there is exactly 1 input value per thread—to simplify the presentation of the algorithm.

Analyzing the behavior of this algorithm, we see that it will perform only $\log_2 n$ iterations of the loop, which is optimal. However, this algorithm applies the operator $O(n \log n)$ times, which is asymptotically inefficient compared to the $O(n)$ applications performed by the serial algorithm. It also has practical disadvantages, such as requiring $2 \log_2 n$ barrier synchronizations. In Section 3, we will show how this basic algorithm can be adapted into something that is both asymptotically efficient and very fast in practice.

2.1 Segmented Scan

Segmented scan generalizes the scan primitive by simultaneously performing separate parallel scans on *arbitrary* contiguous partitions ("segments") of the input vector. For example, an inclusive scan of the + operator over a sequence of integer sequences would give the following result:

```
                a = [ [3 1] [7 0  4] [1 6] [3] ]
    segscan(a, +) = [ [3 4] [7 7 11] [1 7] [3] ]
```

Segmented scans provide as much parallelism as unsegmented scans, but operate on data-dependent regions. Consequently, they are extremely helpful in mapping irregular computations such as quicksort and sparse matrix-vector multiplication onto regular execution structures, such as CUDA's thread blocks.

Segmented sequences of this kind are typically represented by a combination of (1) a sequence of values and (2) a *segment descriptor* that encodes how the sequence is divided into segments. Of the many possible encodings of the segment descriptor, we focus on using a *head flags* array which stores a 1 for each element that begins a segment and 0 for all others. This representation is convenient for massively parallel machines and all other representations can naturally be converted to this form. The head flags representation for the example sequence above is:

```
a.values = [ 3 1 7 0 4 1 6 3 ]
 a.flags = [ 1 0 1 0 0 1 0 1 ]
```

For simplicity of presentation, we will treat the head flags array as a sequence of 32-bit integers; however, it may in practice be preferable to represent flags as bits packed in words.

Schwartz demonstrated that segmented scan can be implemented in terms of (unsegmented) scan by a transformation of the given operator [17, 1]. Given the operator $\oplus$ we can construct a new operator $\oplus^s$ that operates on flag-value pairs (f_x, x) as follows:

$$(f_x, x) \oplus^s (f_y, y) = (f_x | f_y, \text{ if } f_y \text{ then } y \text{ else } x \oplus y)$$

Segmented scan can also be implemented directly rather than by operator transformation [5, 18]. In Section 4 we explore both of these implementation strategies.

3 Designing an Efficient Scan

The CUDA programming model requires the programmer to organize parallel kernels into a hierarchy of threads, thread blocks (of at most 512 threads each), and grids of thread blocks. Furthermore, the NVIDIA GPU architecture executes the threads of a block in SIMT (single instruction, multiple thread) groups of 32 called *warps* [14, 15]. While the threads of a warp may follow any execution path, they execute with a shared instruction unit and thus are executing only a single instruction at any instant in time.

This hierarchical grouping of threads naturally encourages a divide-and-conquer approach to designing parallel scan algorithms. To achieve maximum efficiency, we organize our algorithms to match these natural execution granularities. At the lowest level, we design an *intra-warp* primitive to perform a scan across a single warp of threads. We then construct an *intra-block* primitive that composes intra-warp scans together in order to perform a scan across a block of threads. Finally, we combine grids of intra-block scans into a *global* scan of arbitrary length.

In the following sections, we focus primarily on the design of algorithms for intra-block scans. To simplify the discussion, we assume that there is exactly 1 thread per element in the sequence being scanned. Our experience shows that the best efficiency is achieved when each thread initially performs a serial scan of multiple input elements (see Section 6), a finding noted by others as well [7]. The code we present is templated on the input data type and binary operator used, which is assumed to be associative and to possess an identity value. Input arrays to our scan functions (e.g., `ptr` and `hd`) are assumed to be located in fast on-chip shared memory. The code invoking the scan functions is responsible for loading array data from global (off-chip) device memory into (on-chip) shared memory and storing results back.

3.1 Intra-Warp Scan Algorithm

We begin by defining a routine to perform a scan over a warp of 32 threads, shown in Figure 3. It uses precisely the same algorithm as shown in Figure 2, but with a few basic optimizations. First, we take advantage of the synchronous execution of threads in a warp to eliminate the need for barriers. Second, since we know the size of the sequence is fixed at 32, we unroll the loop. We also add the ability to select either an inclusive or exclusive scan via a `ScanKind` template parameter.

For a warp of size w, this algorithm performs $O(w \log w)$ work rather than the optimal $O(w)$ work performed by a work-efficient algorithm [1]. However, since the threads of a warp execute in a SIMT fashion, there is actually no advantage in decreasing work at the expense of increasing

```
template<class OP, ScanKind Kind, class T>
__device__ T scan_warp(volatile T *ptr, const unsigned int idx=threadIdx.x)
{
    const unsigned int lane = idx & 31; // index of thread in warp (0..31)

    if (lane >=  1)  ptr[idx] = OP::apply(ptr[idx -  1] , ptr[idx]);
    if (lane >=  2)  ptr[idx] = OP::apply(ptr[idx -  2] , ptr[idx]);
    if (lane >=  4)  ptr[idx] = OP::apply(ptr[idx -  4] , ptr[idx]);
    if (lane >=  8)  ptr[idx] = OP::apply(ptr[idx -  8] , ptr[idx]);
    if (lane >= 16)  ptr[idx] = OP::apply(ptr[idx - 16] , ptr[idx]);

    if( Kind==inclusive ) return ptr[idx];
    else                  return (lane>0) ? ptr[idx-1] : OP::identity();
}
```

Figure 3. Scan routine for warp of 32 threads with operator `OP` over values of type `T`. The `Kind` parameter is either `inclusive` or `exclusive`.

the number of steps taken. Each instruction executed by the warp has the same cost, whether executed by a single thread or all threads of the warp. Since the work-efficient reduce/downsweep algorithm [1] performs twice as many steps as the algorithm used here, it leads to measurably lower performance in practice.

3.2 Intra-Block Scan Algorithm

We now construct an algorithm to scan across all the threads of a block using this intra-warp primitive. For simplicity, we assume that the maximum block size is at most the square of the warp width, which is true for the GPUs we target. Given this assumption, the intra-block scan algorithm is quite simple.

1. Scan all warps in parallel using inclusive `scan_warp()`.
2. Record the last partial result from each warp i
3. A single warp performs `scan_warp()` on the partial results from Step 2.
4. Each thread of warp i accumulates partial results from Step 3 into its output element from Step 1.

This organization of the algorithm is only possible because of our assumption that the scan operator is associative. The CUDA implementation of this algorithm is shown in Figure 4. The individual steps are labeled and correspond to the algorithm outline.

3.3 Global Scan Algorithm

The `scan_block()` routine performs a scan of fixed size, corresponding to the size of the thread blocks. We use this routine to construct a "global" scan routine for sequences of any length as follows.

```
template<class OP, ScanKind Kind, class T>
__device__ T scan_block(volatile T *ptr, const unsigned int idx=threadIdx.x)
{
    const unsigned int lane   = idx & 31;
    const unsigned int warpid = idx >> 5;

    // Step 1: Intra-warp scan in each warp
    T val = scan_warp<OP,Kind>(ptr, idx);
    __syncthreads();

    // Step 2: Collect per-warp partial results
    if( lane==31 )  ptr[warpid] = ptr[idx];
    __syncthreads();

    // Step 3: Use 1st warp to scan per-warp results
   if( warpid==0 ) scan_warp<OP,inclusive>(ptr, idx);
   __syncthreads();

    // Step 4: Accumulate results from Steps 1 and 3
   if (warpid > 0) val = OP::apply(ptr[warpid-1], val);
    __syncthreads();

    // Step 5: Write and return the final result
    ptr[idx] = val;
    __syncthreads();

    return val;
}
```

Figure 4. Intra-block scan routine composed from `scan_warp()` primitives.

1. Scan all blocks in parallel using `scan_block()`.
2. Store the partial result from each block i to `block_results[`i`]`.
3. Perform a scan of `block_results`.
4. Each thread of block i adds element i from Step 3 to its output element from Step 1.

Because they require global synchronization, Steps 1 & 2, 3, and 4 require 3 separate CUDA kernel invocations. Indeed, Step 3 may require repeated application of the global scan algorithm if the number of blocks in Step 1 is greater than the block size.

Aside from the decomposition into kernels, the structure of this global algorithm is strikingly similar to the intra-block algorithm. Indeed, they are nearly identical except for Step 3, where the fixed width of blocks guarantees that the intra-block routine can scan per-warp partial results using a single warp, while the variable block count necessary in the global scan does not provide an analogous guarantee.

4 Efficient Segmented Scan

To implement efficient segmented scan routines, we follow the same design strategy already outlined in Section 3 for scan. We begin by defining an intra-warp primitive, from which we can build an intra-block primitive, and ultimately a global segmented scan algorithm. The implementations are also quite similar, with the added complications of dealing with arrays of head flags.

4.1 Operator Transformation

As described in Section 2.1, segmented scan can be implemented by transforming the operator $\oplus$ into a segmented operator $\oplus^s$ that operates on flag–value pairs [17]. This leads to a particularly simple strategy of defining a `segmented<>` template such that `scan<segmented<OP>>` applied to an array of flag–value pairs accomplishes the desired segmented scan. Sample code for such a transformer is shown in Figure 5. This trivially converts the inclusive `scan_warp()` and `scan_block()` routines given in Section 3 into segmented scans. Achieving a correct exclusive segmented scan via operator transformation requires additional changes to the inclusive/exclusive logic in these routines.

Although a reasonable approach, one downside of relying purely on operator transformation is that it alters the external interface of the scan routines. It accepts a sequence of flag–value pairs rather than corresponding sequences of values and flags. We can restore our desired interface, and simultaneously accommodate correct handling of exclusive scans, by explicitly expanding `scan_warp<segmented<OP>>()` into the `segscan_warp()` routine shown in Figure 6. The structure of this procedure is exactly the same as the one shown in Figure 3 except that (1) it does roughly twice as many operations and (2) requires slightly different logic for determining the final inclusive/exclusive result.

4.2 Direct Intra-Warp Segmented Scan

We have also explored an alternative technique for adapting our basic `scan_warp` procedure into an intra-warp segmented scan. This routine, which is shown in Figure 7, operates by augmenting the conditionals used in the indexing of the successive steps of the algorithm. Each thread computes the index of the head of its segment, or 0 if the head is not within its warp. This is the *minimum*

```
template<class OP>
struct segmented
{
    template<class T>
    static __host__ __device__
    inline T apply(const T a, const T b)
    {
        T c;
        c.flag  = b.flag | a.flag;
        c.value = b.flag ? b.value : OP::apply(a.value, b.value);
        return c;
    }
};
```

Figure 5. Code for transforming operator `OP` on values of type `T` into an operator `segmented<OP>` on flag–value pairs.

index of the segment, and is recorded in the variable `mindex`. We compute `mindex` by writing the index of each segment head to the `hd` array and propagating it within the warp to other elements of its segment via a max-scan operation. We take advantage of the unpacked format of the head flags and use them as temporary scratch space.

We use the minimum segment indices to guarantee that elements from different segments are never accumulated. The unsegmented routine shown in Figure 3 is essentially just the special case where `mindex=0`. An example of the resulting data movement for a warp of size 8 is illustrated in Figure 8.

4.3 Block and Global Segmented Scan Algorithms

Either of the intra-warp segmented scan routines we have just outlined can be used to build an intra-block segmented scan. The methodology is essentially identical to our construction of the intra-block scan routine in Section 3.2, with two additional complications. First, when writing the partial result produced by the last thread of each warp in Step 2, we also write an aggregate segment flag for the entire warp. This flag indicates whether there is a segment boundary within the warp, and is simply the (implicit) or-reduction of the flags of the warp. Second, we accumulate the per-warp offsets in Step 4 only to elements of the first segment of each warp. This process is illustrated in Figure 9.

The full CUDA implementation of this algorithm is shown in Figure 10. We first record if the warp starts with a new segment, because the flags array is converted to `mindex`-form by the first `segscan_warp()` call. Step 2b determines whether any flag in a warp was set. This step also determines if the thread belongs to the first segment in its warp by checking if (1) the first element of the warp is not the first element of a segment (the warp is *open*) and (2) the index of the head of the segment is 0. The remaining steps compute warp offsets using a segmented scan of per-warp partial results and accumulates them to the per-thread results computed in Step 1.

The global segmented scan algorithm can be built in the same way. We observe that another way to check if a thread belongs to the first segment of a warp (or block) is to do a min-reduction of `hd`. This gives the index of the first element of the second segment in each warp. Each thread

```
template<class OP, ScanKind Kind, class T>
__device__ T segscan_warp(volatile T *ptr, volatile flag_type *hd,
                          const unsigned int idx = threadIdx.x)
{
   const unsigned int lane = idx & 31;

   if (lane >=  1) {
       ptr[idx] = hd[idx] ? ptr[idx] : OP::apply(ptr[idx - 1] , ptr[idx]);
       hd[idx]  = hd[idx -  1] | hd[idx];
   }
   if (lane >=  2) {
       ptr[idx] = hd[idx] ? ptr[idx] : OP::apply(ptr[idx - 2] , ptr[idx]);
       hd[idx]  = hd[idx -  2] | hd[idx];
   }
   if (lane >=  4) {
       ptr[idx] = hd[idx] ? ptr[idx] : OP::apply(ptr[idx - 4] , ptr[idx]);
       hd[idx]  = hd[idx -  4] | hd[idx];
   }
   if (lane >=  8) {
       ptr[idx] = hd[idx] ? ptr[idx] : OP::apply(ptr[idx - 8] , ptr[idx]);
       hd[idx]  = hd[idx -  8] | hd[idx];
   }
   if (lane >= 16) {
       ptr[idx] = hd[idx] ? ptr[idx] : OP::apply(ptr[idx - 16] , ptr[idx]);
       hd[idx]  = hd[idx - 16] | hd[idx];
   }

   if( Kind==inclusive )  return ptr[idx];
   else  return (lane>0 && !flag) ? ptr[idx-1] : OP::identity();
}
```

Figure 6. Intra-warp segmented scan derived by expanding `scan_warp<segmented<OP>>`.

```
template<class OP, ScanKind Kind, class T>
__device__ T segscan_warp(volatile T *ptr, volatile flag_type *hd,
                          const unsigned int idx = threadIdx.x)
{
    const unsigned int lane = idx & 31;

    // Step 1: Convert head flags to minimum-index form
    if( hd[idx] ) hd[idx] = lane;
    flag_type mindex = scan_warp<op_max, inclusive>(hd);

    // Step 2: Perform segmented scan across warp of size 32
    if( lane >= mindex + 1 )  ptr[idx] = OP::apply(ptr[idx - 1] , ptr[idx]);
    if( lane >= mindex + 2 )  ptr[idx] = OP::apply(ptr[idx - 2] , ptr[idx]);
    if( lane >= mindex + 4 )  ptr[idx] = OP::apply(ptr[idx - 4] , ptr[idx]);
    if( lane >= mindex + 8 )  ptr[idx] = OP::apply(ptr[idx - 8] , ptr[idx]);
    if( lane >= mindex +16 )  ptr[idx] = OP::apply(ptr[idx -16] , ptr[idx]);

    // Step 3: Return correct value for inclusive/exclusive kinds
    if( Kind==inclusive )  return ptr[idx];
    else  return (lane>0 && mindex!=lane) ? ptr[idx-1] : OP::identity();
}
```

Figure 7. Intra-warp segmented scan using conditional indexing.

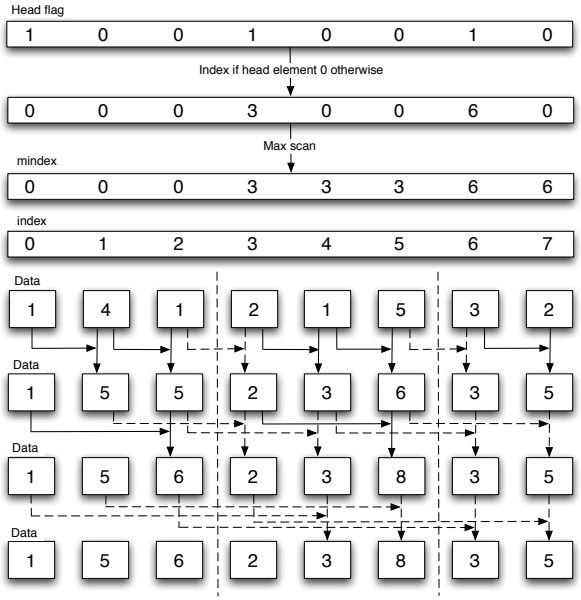


Figure 8. Data movement in intra-warp segmented scan code shown in Figure 7 for threads 0–7 of an 8-thread warp. Data movement in the unsegmented case (dotted arrows) crossing segment boundaries (vertical dashed lines) are not allowed.

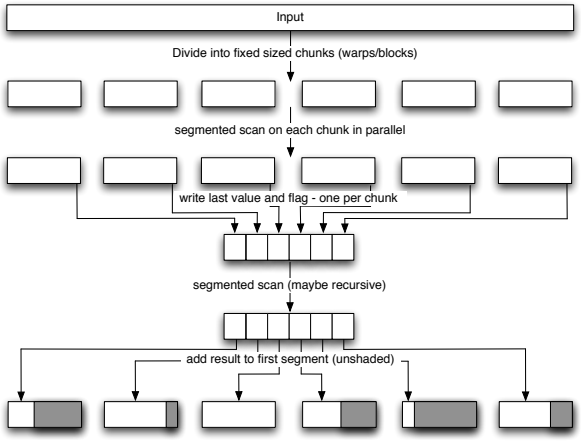


Figure 9. Constructing block/global segmented scans from warp/block segmented scans.

then checks if its thread index is less than this index before adding the result in Step 4. This is a typical time vs. space tradeoff. The method used in Figure 10 must carry one value per thread but no reduction is necessary; this alternative must carry only one value per warp but requires a reduction. We use the former when propagating data between warps because shared memory loads and stores are cheap. However, when we are doing a global segmented scan, Steps 1 and 3 happen in different kernel invocations, so data must be written to global memory which is much slower. Therefore for global scans we do a min-reduction and write only one index per block instead of one per thread.

5 Algorithmic Complexity

Given the hierarchy of blocks and warps described above, we can calculate the algorithmic complexity to scan n elements. Let B and w represent the block and warp size in threads, respectively. We assume that each block scans $O(B)$ elements (i.e., $O(1)$ elements per thread). To scan n elements we use n/B blocks. Let S and W denote step and work complexity, respectively. The *work complexity* of an algorithm is defined as the total number of operations the algorithm performs. Multiple operations may be performed in parallel in each step. The *step complexity* of an algorithm is the number of steps required to complete the algorithm given an infinite number of threads/processors; this is essentially equivalent to the critical path length through the data dependency graph of the computation. A subscript of n, b or w indicates complexities of for the whole array, a block, or a warp, respectively.

As already discussed, our `scan_warp()` routine has step complexity $S_w = O(\log_2 w)$ and work complexity $W_w = O(w \log_2 w)$. For a block containing B/w warps, we arrive at the following step and work complexity for a block scan.

$$S_b = O(S_w \log_w B) = O\left((\log_2 w)\left(\frac{\log_2 B}{\log_2 w}\right)\right) = O(\log_2 B) \tag{1}$$

```
template<class OP, ScanKind Kind, class T>
__device__ T segscan_block(volatile T *ptr, volatile flag_type *hd,
                           const unsigned int idx = threadIdx.x)
{
    unsigned int warpid     = idx >> 5;
    unsigned int warp_first = warpid << 5;
    unsigned int warp_last  = warp_first + 31;

    // Step 1a:
    // Before overwriting the input head flags, record whether
    // this warp begins with an "open" segment.
    bool warp_is_open = (hd[warp_first] == 0);
    __syncthreads();

    // Step 1b:
    // Intra-warp segmented scan in each warp.
    T val = segscan_warp<OP,Kind>(ptr, hd, idx);

    // Step 2a:
    // Since ptr[] contains *inclusive* results, irrespective of Kind,
    // the last value is the correct partial result.
    T warp_total = ptr[warp_last];

    // Step 2b:
    // warp_flag is the OR-reduction of the flags in a warp and is
    // computed indirectly from the mindex values in hd[].
    // will_accumulate indicates that a thread will only accumulate a
    // partial result in Step 4 if there is no segment boundary to its left.
    flag_type warp_flag = hd[warp_last]!=0 || !warp_is_open;
    bool will_accumulate = warp_is_open && hd[idx]==0;

    __syncthreads();

    // Step 2c: The last thread in each warp writes partial results
    if( idx == warp_last )
    {
        ptr[warpid] = warp_total;
        hd[warpid]  = warp_flag;
    }
    __syncthreads();

    // Step 3: One warp scans the per-warp results
    if( warpid == 0 )
        segscan_warp<OP,inclusive>(ptr, hd, idx);

    __syncthreads();

    // Step 4: Accumulate results from Step 3, as appropriate.
    if( warpid != 0 && will_accumulate )
        val = OP::apply(ptr[warpid-1], val);
    __syncthreads();

    ptr[idx] = val;
    __syncthreads();

    return val;
}
```

Figure 10. Intra-block segmented scan routine built using intra-warp segmented scans.

$$W_b = O\left(\sum_{i=1}^{\log_w B} \left\lceil \frac{B}{w^i} \right\rceil W_w\right) = w \log_2 w \left(\sum_{i=1}^{\log_w B} \left\lceil \frac{B}{w^i} \right\rceil\right) = O\left(B \log_2 w\right) \tag{2}$$

The same pattern extends to the array (global) level. The step and work complexity for an array comprising an arbitrary number of blocks, n/B is given by the following expressions.

$$S_n = O(S_b \log_B n) = O((\log_2 w)(\log_B n)) = O(\log_2 n) \tag{3}$$

$$W_n = O\left(\sum_{i=1}^{\log_B n} \left\lceil \frac{n}{B^i} \right\rceil W_b\right) = w \log_2 w \left(\sum_{i=1}^{\log_B n} \left\lceil \frac{n}{B^i} \right\rceil \sum_{j=1}^{\log_w B} \left\lceil \frac{B}{w^j} \right\rceil\right) = O\left(n \log_2 w\right) \tag{4}$$

For any given machine, it is safe to assume that the warp size w is in fact some constant number. Under this assumption, the step complexity of our algorithm is $S_n = O(\log n)$ and the work complexity is $W_n = O(n)$, both of which are asymptotically optimal.

6 Optimizations in CUDPP

The code given in Sections 3 and 4 illustrate the core parallel scan algorithms we use. However, to achieve peak performance for scan kernels, CUDPP combines these basic algorithms with an orthogonal set of further optimizations.

The biggest efficiency gain comes from optimizing the amount of work performed by each thread. We find that processing one element per thread does not generate enough computation to hide the I/O latency to off-chip memory, and so we assign eight input elements to each thread. In our implementations this is handled when data is loaded from global device memory into shared memory. Each thread reads two groups of four elements from global memory and scans both groups of four sequentially. The rightmost result in each group of four (i.e., the reduction of the four inputs) is fed as input to a routine very similar to the block level routines shown in Figure 4 and Figure 10. The key difference is that the block level routines are slightly modified to handle two inputs per thread instead of one as shown here. When the block level routine terminates, the result is accumulated back to the groups of four elements which were scanned serially.

The next most important set of optimizations focuses on minimizing the number of registers used. The GPU architecture relies on multithreading to hide memory access latency, and the number of threads that can be co-resident at one time is often limited by their register requirements. Therefore, it is important to maximize the number of available co-resident threads by minimizing register usage. Optmizing register usage is particularly important for segmented scan since many more registers are needed to store and manipulate head flags. The CUDPP code uses a number of low-level code optimizations designed to limit register requirements, including packing multiple head flags into the bits of registers after they are loaded from off-chip memory.

7 Performance Analysis

In this section, we analyze the performance of the scan and segmented scan routines that we have described and compare them to some alternative approaches. All running times were collected on an NVIDIA GeForce GTX 280 GPU with 30 Streaming Multiprocessors (SMs). These measurements do not include any transfers of data between CPU and GPU memory, under the assumption that scan and segmented scan will be used as building blocks in more complicated applications on GPUs.

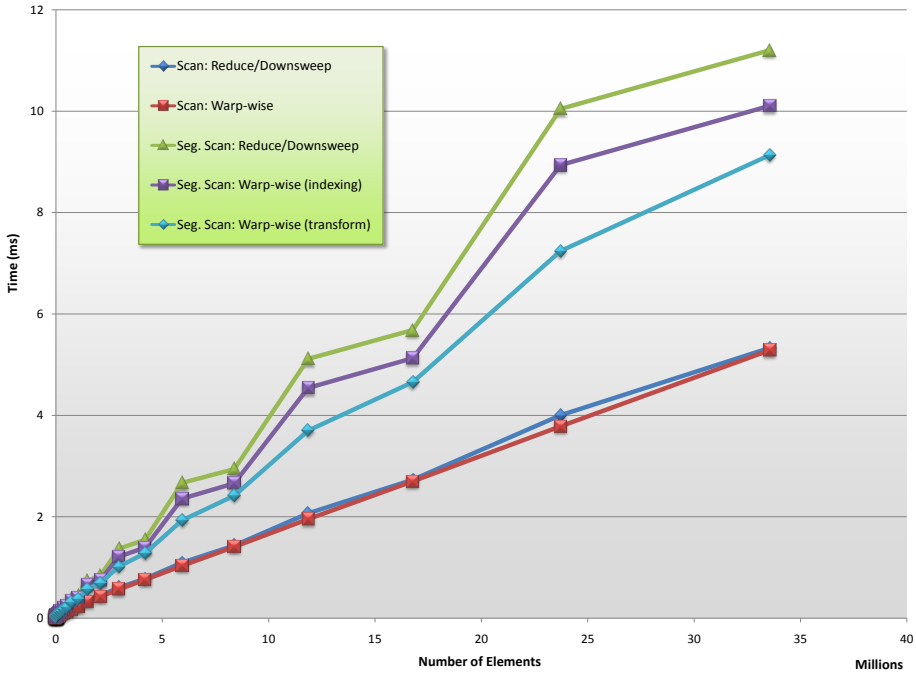


Figure 11. Scan and Segmented Scan performance.

Our first test consists of running both scan and segmented scan routines over sequences of varying length using the CUDPP test apparatus. These tests scan the addition operator over sequences of floating point values.

Figure 11 compares two scan implementations: our warp-wise scan and the reduce/downsweep algorithm used by Sengupta *et al.* [18]. Our warp-wise approach is 5 to 20% faster, improving most on scans of non-power-of-two arrays. Figure 11 compares the reduce/downsweep segmented scan [18] with both of our warp-wise segmented scan kernels: one based on conditional indexing (Fig. 7) and one based on operator transformation (Fig. 6). We see the same trend as in the unsegmented case. Our direct warp-based algorithm using conditional indexing is up to 29% faster than the reduce/downsweep algorithm. The kernel derived from operator transformation improves on it by a further 6 to 10%. Compared to the results reported by Sengupta *et al.* [18] for sequences of 1,048,576 elements running on an older NVIDIA GeForce 8800 GTX GPU, our scan implementation is 2.8 times faster and our segmented scan is 4.2 times faster on the same hardware.

Multiple factors contribute to the performance increase in scan and segmented scan. First, using warp-wise execution minimizes the need for barrier synchronization, because most thread communication is within warps. In contrast, the reduce/downsweep algorithm requires barrier synchronizations between each step in both the reduce and downsweep stages. Second, we halve the number of parallel steps required, as compared to the reduce/downsweep algorithm. This comes at the "expense" of additional work, which is actually not a cost at all due to the SIMT execution of warps. The intra-warp step complexity is in fact optimal. Third, our segmented scan based on operator transformation interleaves scans of flags and data. Like software pipelining, this increases the distance between dependent instructions. Consequently, the performance of this kernel is higher than the indexing kernel, which performs the scan of the flags before the scan of the values. Finally,

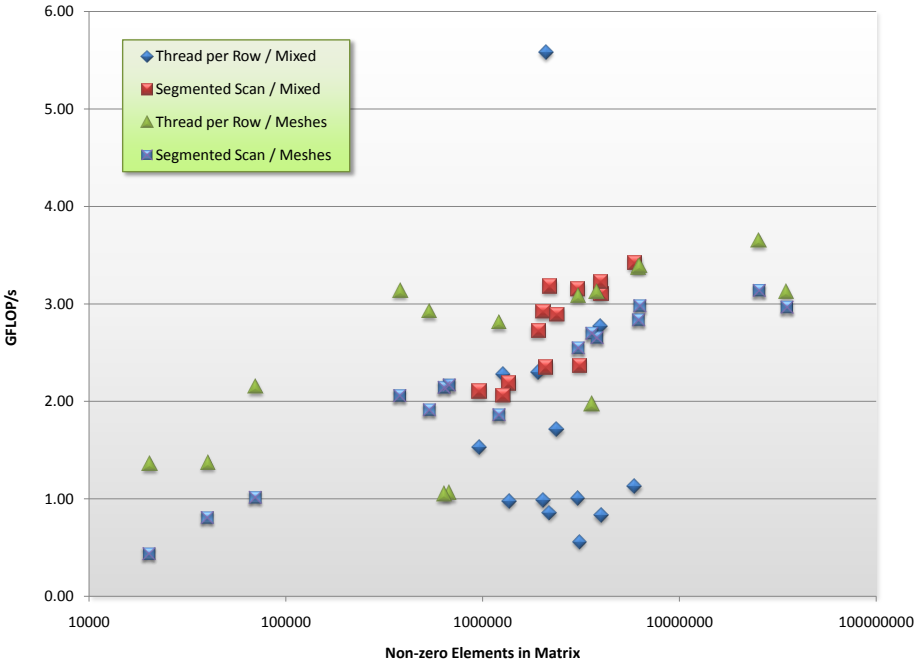


Figure 12. Comparative performance for sparse matrix-vector product kernels using segmented scan vs. assigning 1 thread per row.

while our reduce/downsweep implementations expend much effort to avoid shared memory bank conflicts [8], the intra-warp scan algorithm is inherently conflict-free.

The main efficiency advantage of segmented scan is that its performance is largely invariant to the way in which the sequence is decomposed into subsequences. Thus, it implicitly load-balances work across potentially quite unbalanced subsequences. An example case where this phenomenon is important is in sparse-matrix vector multiplication, where the sparse matrix is represented in a CSR (compressed sparse row) format. One strategy for implementing this computation in CUDA is to assign 1 thread to process each row, examples of which are given by Nickolls *et al.* [15]. Alternatively, sparse-matrix vector multiplication can be implemented with a combined gather and segmented scan [3]. Figure 12 demonstrates the performance of two such kernels on both a corpus of mixed matrices and triangle mesh Laplacian matrices.

The performance of the kernel which simply assigns 1 thread per row has a much higher variance (1.39) as opposed to the kernel using segmented scan (0.55). Furthermore, the segmented scan achieves a higher median performance of 2.59 GFLOP/s as compared to a median of 2.06 GFLOP/s for the thread-per-row kernel. For the mesh matrices, which are all reasonably uniform, the performance rates of the segmented scan are tightly clustered—with the exception of 3 small matrices where overhead is a large cost. Even on the more varied mixed corpus, the performance rates are similar. In contrast, while the thread-per-row kernel performs quite well on certain matrices with uniformly short rows, it suffers considerably on matrices with many very long rows or imbalanced row lengths.

Finally, Figure 13 illustrates the scaling of our segmented scan algorithm on sequences of various sizes. We use the running time on 3 SMs of a GeForce GTX 280 as a base line and show the

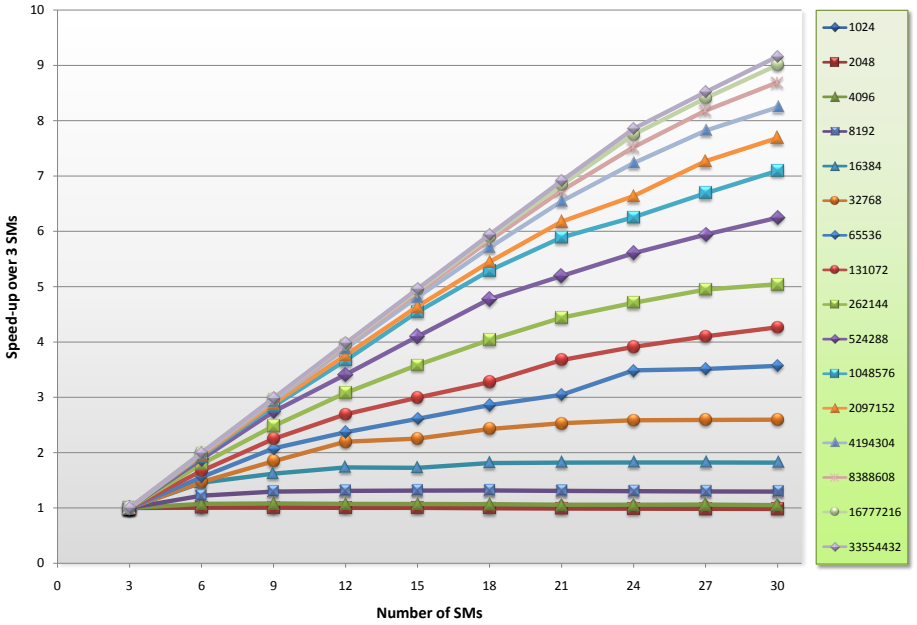


Figure 13. Parallel scaling of segmented scan on sequences of varying length.

speed-up achieved over this base line for 6–30 SMs. Small sequences show relatively little scaling, as they are small enough to be processed efficiently by a small number of SMs. For sequence sizes above 512K elements, we see strong linear scaling. Scaling results for the scan algorithm are similar. This demonstrates the scalability of both the GPU architecture itself and our algorithmic design.

8 Conclusions

The modern manycore GPU is a massively parallel processor and the CUDA programming model provides a straightforward way of writing scalable parallel programs to execute on the GPU. Because of its deeply multithreaded design, a program must expose substantial amounts of fine-grained parallelism to efficiently utilize the GPU. Data-parallel techniques provide a convenient way of expressing such parallelism. Furthermore, the GPU is designed to deliver maximum performance for regular execution paths—via its SIMT architecture—and regular data access patterns—via memory coalescing—and data-parallel algorithms generally fit these expectations quite well.

We have described the design of efficient scan and segmented scan routines, which are essential primitives in a broad range of data-parallel algorithms. By tailoring our algorithms to the natural granularities of the machine and minimizing synchronization, we have produced one of the fastest scan and segmented scan algorithms yet designed for the GPU. The scan and segmented scan routines based on the algorithms described in this paper are available as part of the CUDA Data Parallel Primitives (CUDPP) library, which can be obtained from `http://www.gpgpu.org/developer/cudpp/`.

References

[1] G. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, 1990.

[2] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. Comput.*, 38(11):1526–1538, 1989.
[3] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, Apr. 1994.
[4] W. J. Bouknight, S. A. Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick. The Illiac IV system. *Proceedings of the IEEE*, 60(4):369–388, April 1972.
[5] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Supercomputing '90: Proceedings of the 1990 Conference on Supercomputing*, pages 666–675, 1990.
[6] CUDPP: CUDA data parallel primitives library. `http://www.gpgpu.org/developer/cudpp/`, 2008.
[7] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli. Fast scan algorithms on graphics processors. In *Proc. 22nd Annual International Conference on Supercomputing*, pages 205–213. ACM, June 2008.
[8] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, Aug. 2007.
[9] J. Hensley, T. Scheuermann, G. Coombe, M. Singh, and A. Lastra. Fast summed-area table generation and its applications. *Computer Graphics Forum*, 24(3):547–555, Sept. 2005.
[10] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.
[11] D. Horn. Stream reduction operations for GPGPU applications. In M. Pharr, editor, *GPU Gems 2*, chapter 36, pages 573–589. Addison Wesley, Mar. 2005.
[12] K. E. Iverson. *A Programming Language*. Wiley, New York, 1962.
[13] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, 1980.
[14] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, Mar/Apr 2008.
[15] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, Mar/Apr 2008.
[16] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, June 2008. Version 2.0.
[17] J. T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–521, Oct. 1980.
[18] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Graphics Hardware 2007*, pages 97–106. ACM, Aug. 2007.
[19] S. Sengupta, A. E. Lefohn, and J. D. Owens. A work-efficient step-efficient prefix sum algorithm. In *Proceedings of the Workshop on Edge Computing Using New Commodity Architectures*, pages D–26–27, May 2006.

Brook for GPUs: Stream Computing on Graphics Hardware

Ian Buck Theresa Foley Daniel Horn Jeremy Sugerman Kayvon Fatahalian Mike Houston Pat Hanrahan

Stanford University

Abstract

In this paper, we present Brook for GPUs, a system for general-purpose computation on programmable graphics hardware. Brook extends C to include simple data-parallel constructs, enabling the use of the GPU as a streaming coprocessor. We present a compiler and runtime system that abstracts and virtualizes many aspects of graphics hardware. In addition, we present an analysis of the effectiveness of the GPU as a compute engine compared to the CPU, to determine when the GPU can outperform the CPU for a particular algorithm. We evaluate our system with five applications, the SAXPY and SGEMV BLAS operators, image segmentation, FFT, and ray tracing. For these applications, we demonstrate that our Brook implementations perform comparably to hand-written GPU code and up to seven times faster than their CPU counterparts.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors D.3.2 [Programming Languages]: Language Classifications—Parallel Languages

Keywords: Programmable Graphics Hardware, Data Parallel Computing, Stream Computing, GPU Computing, Brook

1 Introduction

In recent years, commodity graphics hardware has rapidly evolved from being a fixed-function pipeline into having programmable vertex and fragment processors. While this new programmability was introduced for real-time shading, it has been observed that these processors feature instruction sets general enough to perform computation beyond the domain of rendering. Applications such as linear algebra [Krüger and Westermann 2003], physical simulation, [Harris et al. 2003], and a complete ray tracer [Purcell et al. 2002; Carr et al. 2002] have been demonstrated to run on GPUs.

Originally, GPUs could only be programmed using assembly languages. Microsoft's HLSL, NVIDIA's Cg, and OpenGL's GLslang allow shaders to be written in a high level, C-like programming language [Microsoft 2003; Mark et al. 2003; Kessenich et al. 2003]. However, these languages do not assist the programmer in controlling other aspects of the graphics pipeline, such as allocating texture memory, loading shader programs, or constructing graphics primitives. As a result, the implementation of applications requires extensive knowledge of the latest graphics APIs as well as an understanding of the features and limitations of modern hardware. In addition, the user is forced to express their algorithm in terms of graphics primitives, such as textures and triangles. As a result, general-purpose GPU computing is limited to only the most advanced graphics developers.

This paper presents *Brook*, a programming environment that provides developers with a view of the GPU as a streaming coprocessor. The main contributions of this paper are:

- The presentation of the Brook stream programming model for general-purpose GPU computing. Through the use of streams, kernels and reduction operators, Brook abstracts the GPU as a streaming processor.
- The demonstration of how various GPU hardware limitations can be virtualized or extended using our compiler and runtime system; specifically, the GPU memory system, the number of supported shader outputs, and support for user-defined data structures.
- The presentation of a cost model for comparing GPU vs. CPU performance tradeoffs to better understand under what circumstances the GPU outperforms the CPU.

2 Background

2.1 Evolution of Streaming Hardware

Programmable graphics hardware dates back to the original programmable framebuffer architectures [England 1986]. One of the most influential programmable graphics systems was the UNC PixelPlanes series [Fuchs et al. 1989] culminating in the PixelFlow machine [Molnar et al. 1992]. These systems embedded pixel processors, running as a SIMD processor, on the same chip as framebuffer memory. Peercy et al. [2000] demonstrated how the OpenGL architecture [Woo et al. 1999] can be abstracted as a SIMD processor. Each rendering pass implements a SIMD instruction that performs a basic arithmetic operation and updates the framebuffer atomically. Using this abstraction, they were able to compile RenderMan to OpenGL 1.2 with imaging extensions. Thompson et al. [2002] explored the use of GPUs as a general-purpose vector processor by implementing a software layer on top of the graphics library that performed arithmetic computation on arrays of floating point numbers.

SIMD and vector processing operators involve a read, an execution of a single instruction, and a write to off-chip memory [Russell 1978; Kozyrakis 1999]. This results in significant memory bandwidth use. Today's graphics hardware executes small programs where instructions load and store data to local temporary registers rather than to memory. This is a major difference between the vector and stream processor abstraction [Khailany et al. 2001].

The stream programming model captures computational locality not present in the SIMD or vector models through the use of streams and kernels. A *stream* is a collection of records requiring similar computation while *kernels* are

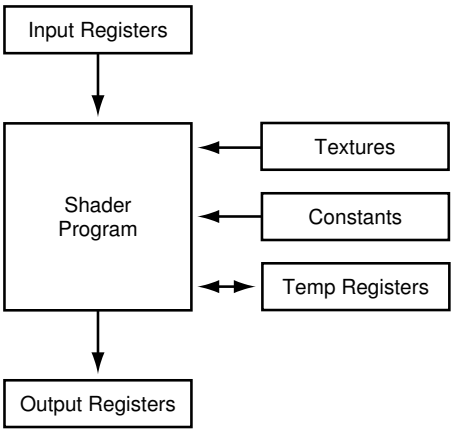


Figure 1: Programming model for current programmable graphics hardware. A shader program operates on a single input element (vertex or fragment) stored in the input registers and writes the execution result into the output registers.

functions applied to each element of a stream. A streaming processor executes a kernel over all elements of an input stream, placing the results into an output stream. Dally et al. [2003] explain how stream programming encourages the creation of applications with high *arithmetic intensity*, the ratio of arithmetic operations to memory bandwidth. This paper defines a similar property called *computational intensity* to compare CPU and GPU performance.

Stream architectures are a topic of great interest in computer architecture [Bove and Watlington 1995; Gokhale and Gomersall 1997]. For example, the Imagine stream processor [Kapasi et al. 2002] demonstrated the effectiveness of streaming for a wide range of media applications, including graphics and imaging [Owens et al. 2000]. The StreamC/KernelC programming environment provides an abstraction which allows programmers to map applications to the Imagine processor [Mattson 2002]. Labonte et al. [2004] studied the effectiveness of GPUs as stream processors by evaluating the performance of a streaming virtual machine mapped onto graphics hardware. The programming model presented in this paper could easily be compiled to their virtual machine.

2.2 Programming Graphics Hardware

Modern programmable graphics accelerators such as the ATI X800XT and the NVIDIA GeForce 6800 [ATI 2004b; NVIDIA 2004] feature programmable vertex and fragment processors. Each processor executes a user-specified assembly-level shader program consisting of 4-way SIMD instructions [Lindholm et al. 2001]. These instructions include standard math operations, such as 3- or 4-component dot products, texture-fetch instructions, and a few special-purpose instructions.

The basic execution model of a GPU is shown in figure 1. For every vertex or fragment to be processed, the graphics hardware places a graphics primitive in the read-only input registers. The shader is then executed and the results written to the output registers. During execution, the shader has access to a number of temporary registers as well as constants set by the host application.

Purcell et al. [2002] describe how the GPU can be considered a streaming processor that executes kernels, written as fragment or vertex shaders, on streams of data stored in geometry and textures. Kernels can be written using a variety of high-level, C-like languages such as Cg, HLSL, and GLslang. However, even with these languages, applications must still execute explicit graphics API calls to organize data into streams and invoke kernels. For example, stream management is performed by the programmer, requiring data to be manually packed into textures and transferred to and from the hardware. Kernel invocation requires the loading and binding of shader programs and the rendering of geometry. As a result, computation is not expressed as a set of kernels acting upon streams, but rather as a sequence of shading operations on graphics primitives. Even for those proficient in graphics programming, expressing algorithms in this way can be an arduous task.

These languages also fail to virtualize constraints of the underlying hardware. For example, stream elements are limited to natively-supported `float`, `float2`, `float3`, and `float4` types, rather than allowing more complex user-defined structures. In addition, programmers must always be aware of hardware limitations such as shader instruction count, number of shader outputs, and texture sizes. There has been some work in shading languages to alleviate some of these constraints. Chan et al. [2002] present an algorithm to subdivide large shaders automatically into smaller shaders to circumvent shader length and input constraints, but do not explore multiple shader outputs. McCool et al. [2002; 2004] have developed Sh, a system that allows shaders to be defined and executed using a metaprogramming language built on top of C++. Sh is intended primarily as a shading system, though it has been shown to perform other types of computation. However, it does not provide some of the basic operations common in general purpose computing, such as gathers and reductions.

In general, code written today to perform computation on GPUs is developed in a highly graphics-centric environment, posing difficulties for those attempting to map other applications onto graphics hardware.

3 Brook Stream Programming Model

Brook was a developed as a language for streaming processors such as Stanford's Merrimac streaming supercomputer [Dally et al. 2003], the Imagine processor [Kapasi et al. 2002], the UT Austin TRIPS processor [Sankaralingam et al. 2003], and the MIT Raw processor [Taylor et al. 2002]. We have adapted Brook to the capabilities of graphics hardware, and will only discuss Brook in the context of GPU architectures in this paper. The design goals of the language include:

- **Data Parallelism and Arithmetic Intensity**

  By providing native support for streams, Brook allows programmers to express the data parallelism that exists in their applications. Arithmetic intensity is improved by performing computations in kernels.

- **Portability and Performance**

  In addition to GPUs, the Brook language maps to a variety of streaming architectures. Therefore the language is free of any explicit graphics constructs. We have created Brook implementations for both NVIDIA and ATI hardware, using both DirectX and OpenGL, as well as a CPU reference implementation. Despite the need to maintain portability, Brook programs execute efficiently on the underlying hardware.

In comparison with existing high-level languages used for GPU programming, Brook provides the following abstractions.

- Memory is managed via streams: named, typed, and "shaped" data objects consisting of collections of records.
- Data-parallel operations executed on the GPU are specified as calls to parallel functions called kernels.
- Many-to-one reductions on stream elements are performed in parallel by reduction functions.

Important features of the Brook language are discussed in the following sections.

3.1 Streams

A stream is a collection of data which can be operated on in parallel. Streams are declared with angle-bracket syntax similar to arrays, i.e. `float s<10,5>` which denotes a 2-dimensional stream of `float`s. Each stream is made up of *elements*. In this example, `s` is a stream consisting of 50 elements of type float. The *shape* of the stream refers to its dimensionality. In this example, `s` is a stream of shape 10 by 5. Streams are similar to C arrays, however, access to stream data is restricted to kernels (described below) and the `streamRead` and `streamWrite` operators, that transfer data between memory and streams.

Streams may contain elements of type `float`, Cg vector types such as `float2`, `float3`, and `float4`, and structures composed of these native types. For example, a stream of rays can be defined as:

```
typedef struct ray_t {
   float3 o;
   float3 d;
   float tmax;
} Ray;
Ray r<100>;
```

Support for user-defined memory types, though common in general-purpose languages, is a feature not found in today's graphics APIs. Brook provides the user with the convenience of complex data structures and compile-time type checking.

3.2 Kernels

Brook kernels are special functions, specified by the `kernel` keyword, which operate on streams. Calling a kernel on a stream performs an implicit loop over the elements of the stream, invoking the body of the kernel for each element. An example kernel is shown below.

```
kernel void saxpy (float a, float4 x<>, float4 y<>,
                   out float4 result<>) {
  result = a*x + y;
}

void main (void) {
  float a;
  float4  X[100], Y[100], Result[100];
  float4  x<100>, y<100>, result<100>;
  ... initialize a, X, Y ...
  streamRead(x, X);             // copy data from mem to stream
  streamRead(y, Y);
  saxpy(a, x, y, result);       // execute kernel on all elements
  streamWrite(result, Result);  // copy data from stream to mem
}
```

Kernels accept several types of arguments:

- Input streams that contain read-only data for kernel processing.
- Output streams, specified by the `out` keyword, that store the result of the kernel computation. Brook imposes no limit to the number of output streams a kernel may have.
- Gather streams, specified by the C array syntax (`array[]`): Gather streams permit arbitrary indexing to retrieve stream elements. In a kernel, elements are fetched, or "gathered", via the array index operator, i.e. `array[i]`. Like regular input streams, gather streams are read-only.
- All non-stream arguments are read-only constants.

If a kernel is called with input and output streams of differing shape, Brook implicitly resizes each input stream to match the shape of the output. This is done by either repeating (`123` to `111222333`) or striding (`123456789` to `13579`) elements in each dimension.

Certain restrictions are placed on kernels to allow data-parallel execution. Memory access is limited to reads from gather streams, similar to a texture fetch. Operations that may introduce side-effects between stream elements, such as writing static or global variables, are not allowed in kernels. Streams are allowed to be both input and output arguments to the same kernel (in-place computation) provided they are not also used as gather streams in the kernel.

A sample kernel which computes a ray-triangle intersection is shown below.

```
kernel void krnIntersectTriangle(Ray ray<>, Triangle tris[],
                                 RayState oldraystate<>,
                                 GridTrilist trilist[],
                                 out Hit candidatehit<>) {
  float idx, det, inv_det;
  float3 edge1, edge2, pvec, tvec, qvec;
  if(oldraystate.state.y > 0) {
    idx = trilist[oldraystate.state.w].trinum;
    edge1 = tris[idx].v1 - tris[idx].v0;
    edge2 = tris[idx].v2 - tris[idx].v0;
    pvec = cross(ray.d, edge2);
    det = dot(edge1, pvec);
    inv_det = 1.0f/det;
    tvec = ray.o - tris[idx].v0;
    candidatehit.data.y = dot( tvec, pvec ) * inv_det;
    qvec = cross( tvec, edge1 );
    candidatehit.data.z = dot( ray.d, qvec ) * inv_det;
    candidatehit.data.x = dot( edge2, qvec ) * inv_det;
    candidatehit.data.w = idx;
  } else {
    candidatehit.data = float4(0,0,0,-1);
  }
}
```

Brook forces the programmer to distinguish between data streamed to a kernel as an input stream and that which is gathered by the kernel using array access. This distinction permits the system to manage these streams differently. Input stream elements are accessed in a regular pattern but are never reused, since each kernel body invocation operates on a different stream element. Gather streams may be accessed randomly, and elements may be reused. As Purcell et al. [2002] observed, today's graphics hardware makes no distinction between these two memory-access types. As a result, input stream data can pollute a traditional cache and penalize locality in gather operations.

The use of kernels differentiates stream programming from vector programming. Kernels perform arbitrary function

evaluation whereas vector operators consist of simple math operations. Vector operations always require temporaries to be read and written to a large vector register file. In contrast, kernels capture additional locality by storing temporaries in local register storage. By reducing bandwidth to main memory, arithmetic intensity is increased since only the final result of the kernel computation is written back to memory.

3.3 Reductions

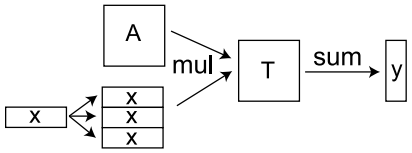
While kernels provide a mechanism for applying a function to a set of data, reductions provide a data-parallel method for calculating a single value from a set of records. Examples of reduction operations include arithmetic sum, computing a maximum, and matrix product. In order to perform the reduction in parallel, we require the reduction operation to be associative: $(a \circ b) \circ c = a \circ (b \circ c)$. This allows the system to evaluate the reduction in whichever order is best suited for the underlying architecture.

Reductions accept a single input stream and produce as output either a smaller stream of the same type, or a single-element value. Outputs for reductions are specified with the `reduce` keyword. Both reading and writing to the reduce parameter are allowed when computing the reduction of the two values.

If the output argument to a reduction is a single element, it will receive the reduced value of all of the input stream's elements. If the argument is a stream, the shape of the input and output streams is used to determine how many neighboring elements of the input are reduced to produce each element of the output.

The example below demonstrates how stream-to-stream reductions can be used to perform the matrix-vector multiplication $y = Ax$.

```
kernel void mul (float a<>, float b<>, out float c<>) {
  c = a * b;
}
reduce void sum (float a<>, reduce float r<>) {
  r += a;
}

float A<50,50>;
float x<1,50>;
float T<50,50>;
float y<50,1>;
...
mul(A,x,T);
sum(T,y);
```



In this example, we first multiply `A` by `x` with the `mul` kernel. Since `x` is smaller than `T` in the first dimension, the elements of `x` are repeated in that dimension to create a matrix of equal size of `T`. The `sum` reduction then reduces rows of `T` because of the difference in size of the second dimension of `T` and `y`.

3.4 Additional language features

In this section, we present additional Brook language features which should be mentioned but will not be discussed further in this paper. Readers who are interested in more details are encouraged to read [Buck 2004].

- The `indexof` operator may be called on an input or output stream inside a kernel to obtain the position of the current element within the stream.
- *Iterator streams* are streams containing pre-initialized sequential values specified by the user. Iterators are useful for generating streams of sequences of numbers.
- The Brook language specification also provides a collection of high-level stream operators useful for manipulating and reorganizing stream data, such as grouping elements into new streams and extracting subregions of streams and explicit operators to stride, repeat, and wrap streams. These operators can be implemented on the GPU through the use of iterator streams and gather operations. Their use is important on streaming platforms which do not support gather operations inside kernels.
- The Brook language provides parallel indirect read-modify-write operators called *ScatterOp* and *GatherOp* which are useful for building and manipulating data structures contained within streams. However, due to GPU hardware limitations, we currently perform these operations on the CPU.

4 Implementation on Graphics Hardware

The Brook compilation and runtime system maps the Brook language onto existing programmable GPU APIs. The system consists of two components: `brcc`, a source-to-source compiler, and the Brook Runtime (BRT), a library that provides runtime support for kernel execution. The compiler is based on cTool [Flisakowski 2004], an open-source C parser, which was modified to support Brook language primitives. The compiler maps Brook kernels into Cg shaders which are translated by vendor-provided shader compilers into GPU assembly. Additionally, `brcc` emits C++ code which uses the BRT to invoke the kernels. Appendix A provides a before-and-after example of a compiled kernel.

BRT is an architecture-independent software layer which provides a common interface for each of the backends supported by the compiler. Brook currently supports three backends; an OpenGL and DirectX backend and a reference CPU implementation. Creating a cross-platform implementation provides three main benefits. First, we demonstrate the portability of the language by allowing the user to choose the best backend for the hardware. Secondly, we can compare the performance of the different graphics APIs for GPU computing. Finally, we can optimize for API-specific features, such as OpenGL's support of 0 to n texture addressing and DirectX's direct render-to-texture functionality.

The following sections describe how Brook maps the stream, kernel, and reduction language primitives onto the GPU.

4.1 Streams

Brook represents streams as floating point textures on the graphics hardware. With this representation, the `streamRead` and `streamWrite` operators upload and download texture data, gather operations are expressed as dependent texture reads, and the implicit repeat and stride operators are achieved with texture sampling. Current graphics APIs, however, only provide `float`, `float2`, `float3` and `float4` texture formats. To support streams of user-defined structures, BRT stores each member of a structure in a different hardware texture.

Many application writers may wish to visualize the result of a Brook computation. The BRT provides a C++ interface which allows the user to bind Brook streams as native

graphics API textures which can be interactively rendered in a traditional graphics application. This option requires that Brook make streams available in a fixed, documented texture layout. By default, streams are stored as a texture with the same dimensions as the stream shape.

A greater challenge is posed by the hardware limitations on texture size and shape. Floating-point textures are limited to two dimensions, and a maximum size of 4096 by 4096 on NVIDIA and 2048 by 2048 on ATI hardware. If we directly map stream shape to texture shape, then Brook programs can not create streams of more than two dimensions or 1D streams of more than 2048 or 4096 elements.

To address this limitation, `brcc` provides a compiler option to wrap the stream data across multiple rows of a texture. This permits arbitrary-sized streams assuming the total number of elements fits within a single texture. In order to access an element by its location in the stream, `brcc` inserts code to convert between the stream location and the corresponding texture coordinates. The Cg code shown below is used for stream-to-texture address translation and allows for streams of up to four dimensions containing as many elements as texels in a maximum sized 2D texture.

```
float2 __calculatetexpos( float4 streamIndex,
    float4 linearizeConst, float2 reshapeConst ) {
  float linearIndex = dot( streamIndex, linearizeConst );
  float texX = frac( linearIndex );
  float texY = linearIndex - texX;
  return float2( texX, texY ) * reshapeConst;
}
```

Our address-translation implementation is limited by the precision available in the graphics hardware. In calculating a texture coordinate from a stream position, we convert the position to a scaled integer index. If the unscaled index exceeds the largest representable sequential integer in the graphics card's floating-point format (16,777,216 for NVIDIA's s23e8 format, 131,072 for ATI's 24-bit s16e7 format) then there is not sufficient precision to uniquely address the correct stream element. For example, our implementation effectively increases the maximum 1D stream size for a portable Brook program from 2048 to 131072 elements on ATI hardware. Ultimately, these limitations in texture addressing point to the need for a more general memory addressing model in future GPUs.

4.2 Kernels

With stream data stored in textures, Brook uses the GPU's fragment processor to execute a kernel function over the stream elements. `brcc` compiles the body of a kernel into a Cg shader. Stream arguments are initialized from textures, gather operations are replaced with texture fetches, and non-stream arguments are passed via constant registers. The NVIDIA or Microsoft shader compiler is then applied to the resulting Cg code to produce GPU assembly.

To execute a kernel, the BRT issues a single quad containing the same number of fragments as elements in the output stream. The kernel outputs are rendered into the current render targets. The DirectX backend renders directly into the textures containing output stream data. OpenGL, however, does not provide a lightweight mechanism for binding textures as render targets. OpenGL Pbuffers provide this functionality, however, as Bolz et al.[2003] discovered, switching between render targets with Pbuffers can have significant performance penalties. Therefore, our OpenGL backend renders to a single floating-point Pbuffer and copies the results to the output stream's texture. The proposed Superbuffer specification [Percy 2003], which permits direct render-to-texture functionality under OpenGL, should alleviate this restriction.

Program	Instructions texld	arith	MFLOPS	Slowdown
Mat4Mult4	8	16	3611	
Mat4Mult1	20	16	1683	53%
Cloth4	6	54	5086	
Cloth1	12	102	2666	47%

Table 1: This table demonstrates the performance cost of splitting kernels which contain more outputs than supported by the hardware. Included are the instruction counts and observed performance of the matrix multiply and cloth kernels executing on both 4-output hardware and 1-output hardware using the NVIDIA DirectX backend. The slowdown is the relative drop in performance of the non-multiple output implementation.

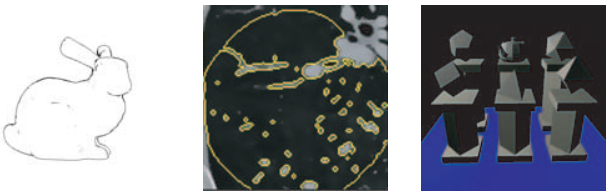
The task of mapping kernels to fragment shaders is complicated by the limited number of shader outputs available in today's hardware. When a kernel uses more output streams than are supported by the hardware (or uses an output stream of structure type), `brcc` splits the kernel into multiple passes in order to compute all of the outputs. For each pass, the compiler produces a complete copy of the kernel code, but only assigns a subset of the kernel outputs to shader outputs. We take advantage of the aggressive dead-code elimination performed by today's shader compilers to remove any computation that does not contribute to the outputs written in that pass.

To test the effectiveness of our pass-splitting technique, we applied it to two kernels: Mat4Mult, which multiplies two streams of 4x4 matrices, producing a single 4x4 matrix (4 `float4`s) output stream; and Cloth, which simulates particle-based cloth with spring constraints, producing updated particle positions and velocities. We tested two versions of each kernel. Mat4Mult4 and Cloth4 were compiled with hardware support for 4 `float4` outputs, requiring only a single pass to complete. The Mat4Mult1 and Cloth1 were compiled for hardware with only a single output, forcing the runtime to generate separate shaders for each output.

As shown in Table 1, the effectiveness of this technique depends on the amount of shared computation between kernel outputs. For the Mat4Mult kernel, the computation can be cleanly separated for each output, and the shader compiler correctly identified that each row of the output matrix can be computed independently. Therefore, the total number of arithmetic operations required to compute the result does not differ between the 4-output and 1-output versions. However, the total number of texture loads does increase since each pass must load all 16 elements of one of the input matrices. For the Cloth kernel, the position and velocity outputs share much of the kernel code (a force calculation) which must be repeated if the outputs are to be computed in separate shaders. Thus, there are nearly twice as many instructions in the 1-output version as in the 4-output version. Both applications perform better with multiple-output support, demonstrating that our system efficiently utilizes multiple-output hardware, while transparently scaling to systems with only single-output support.

4.3 Reductions

Current graphics hardware does not have native support for reductions. BRT implements reduction via a multipass



FFT
Edge Detect

Segment
Lung CT

Ray Tracer
Glassner

Figure 2: These images were created using the Brook applications FFT, Segment, and Ray Tracer

method similar to Kruger and Westermann [2003]. The reduction is performed in $O(\log n)$ passes, where n is the ratio of the sizes of the input and output streams. For each pass, the reduce operation reads up to 8 adjacent stream elements, and outputs their reduced values. Since each pass produces between 2 and 8 fewer values, Brook reductions are a linear-time computation. The specific size of each reduction pass is a function of the size of the stream and reduction kernel.

We have benchmarked computing the sum of 2^{20} `float4` elements as taking 2.4 and .79 milliseconds, respectively, on our NVIDIA and ATI DirectX backends and 4.1 and 1.3 milliseconds on the OpenGL backends. An optimized CPU implementation performed this reduction in 14.6 milliseconds. The performance difference between the DirectX and OpenGL implementations is largely due to the cost of copying results from the output Pbuffer to a texture, as described above.

With our multipass implementation of reduction, the GPU must access significantly more memory than an optimized CPU implementation to reduce a stream. If graphics hardware provided a persistent register that could accumulate results across multiple fragments, we could reduce a stream to a single value in one pass. We simulated the performance of graphics hardware with this theoretical capability by measuring the time it takes to execute a kernel that reads a single stream element, adds it to a constant and issues a fragment kill to prevent any write operations. Benchmarking this kernel with DirectX on the same stream as above yields theoretical reduction times of .41 and .18 milliseconds on NVIDIA and ATI hardware respectively.

5 Evaluation and Applications

We now examine the performance of several scientific applications on GPUs using Brook. For each test, we evaluated Brook using the OpenGL and DirectX backends on both an ATI Radeon X800 XT Platinum running version 4.4 drivers and a pre-release[1] NVIDIA GeForce 6800 running version 60.80 drivers, both running Windows XP. For our CPU comparisons, we used a 3 GHz Intel Pentium 4 processor with an Intel 875P chipset running Windows XP, unless otherwise noted.

5.1 Applications

We implemented an assortment of algorithms in Brook. The following applications were chosen for three reasons: they are representative of different types of algorithms performed in numerical applications; they are important algorithms used widely both in computer graphics and general scientific computing; optimized CPU- or GPU-based implementations are available to make performance comparisons with our implementations in Brook.

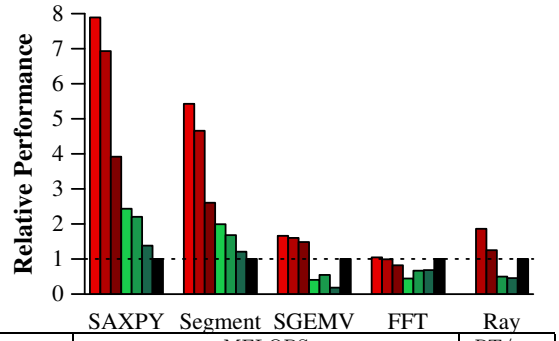
BLAS **SAXPY** and **SGEMV** routines: The BLAS (Basic Linear Algebra Subprograms) library is a collection of low-level linear algebra subroutines [Lawson et al. 1979]. SAXPY performs the vector scale and sum operation, $y = ax + y$, where x and y are vectors and a is a scalar. SGEMV is a single-precision dense matrix-vector product followed by a scaled vector add, $y = \alpha Ax + \beta y$, where x, y are vectors, A is a matrix and α, β are scalars. Matrix-vector operations are critical in many numerical applications, and the double-precision variant of SAXPY is a core computation kernel employed by the LINPACK Top500 benchmark [2004] used to rank the top supercomputers in the world. We compare our performance against that of the optimized commercial Intel Math Kernel Library[Intel 2004] for SAXPY and the ATLAS BLAS library[Whaley et al. 2001] for SGEMV, which were the fastest public CPU implementations we were able to locate. For a reference GPU comparison, we implemented a hand-optimized DirectX version of SAXPY and an optimized OpenGL SGEMV implementation. For these tests, we use vectors or matrices of size 1024^2.

Segment performs a 2D version of the Perona and Malik [1990] nonlinear, diffusion-based, seeded, region-growing algorithm, as presented in Sherbondy et al. [2003], on a 2048 by 2048 image. Segmentation is widely used for medical image processing and digital compositing. We compare our Brook implementation against hand-coded OpenGL and CPU implementations executed on our test systems. Each iteration of the segmentation evolution kernel requires 32 floating point operations, reads 10 floats as input and writes 2 floats as output. The optimized CPU implementation is specifically tuned to perform a maximally cache-friendly computation on the Pentium 4.

FFT: Our Fourier transform application performs a 2D Cooley-Tukey fast Fourier transform (FFT) [1965] on a 4 channel 1024 by 1024 complex signal. The fast Fourier transform algorithm is important in many graphical applications, such as post-processing of images in the framebuffer, as well as scientific applications such as the SETI@home project [Sullivan et al. 1997]. Our implementation uses three kernels: a horizontal and vertical 1D FFT, each called 10 times, and a bit reversal kernel called once. The horizontal and vertical FFT kernels each perform 5 floating-point operations per output value. The total floating point operations performed, based on the benchFFT [Frigo and Johnson 2003] project, is equal to $5 \cdot w \cdot h \cdot channels \cdot \log_2(w \cdot h)$. To benchmark Brook against a competitive GPU algorithm, we compare our results with the custom OpenGL implementation available from ATI at [ATI 2004a]. To compare against the CPU, we benchmark the heavily optimized FFTW-3 software library compiled with the Intel C++ compiler [INTEL 2003].

Ray is a simplified version of the GPU ray tracer presented in Purcell et al. [2002]. This application consists of three kernels, ray setup, ray-triangle intersection (shown in section 3), and shading. For a CPU comparison, we compare against the published results of Wald's [2004] hand-optimized assembly which can achieve up to 100M rays per second on a Pentium 4 3.0GHz processor.

[1] Running 350MHz core and 500Mhz memory



	MFLOPS				RT/sec
ATI					
Reference	4923	14171	2335	1278	-
BrookDX	4324	12163	2251	1212	186
BrookGL	2444	6800	2086	1003	125
NVIDIA					
Reference	1518	5200	567	541	-
BrookDX	1374	4387	765	814	50
BrookGL	861	3152	255	897	45
CPU	624	2616	1407	1224	100

Figure 3: Comparing the relative performance of our test applications between a reference GPU version, a Brook DirectX and OpenGL version, and an optimized CPU version. Results for ATI are shown in red, NVIDIA are shown in green. The bar graph is normalized by the CPU performance as shown by the dotted line. The table lists the observed MFLOPS for each application. For the ray tracer, we list the ray-triangle test rate.

Figure 3 provides a breakdown of the performance of our various test applications. We show the performance of each application running on ATI (shown in red), NVIDIA (green), and the CPU (black). For each GPU platform, the three bars show the performance of the reference native GPU implementation and the Brook version executing with the DirectX and OpenGL backends. The results are normalized by the CPU performance. The table provides the effective MFLOPS observed based on the floating point operations as specified in the original source. For the ray tracing code, we report ray-triangle tests per second. In all of these results, we do not include the `streamRead` and `streamWrite` costs.

We observe that the GPU implementations perform well against their CPU counterparts. The Brook DirectX ATI versions of SAXPY and Segment performed roughly 7 and 4.7 times faster than the equivalent CPU implementations. SAXPY illustrates that even a kernel executing only a single MAD instruction is able to out-perform the CPU due to the additional internal bandwidth available on the GPU. FFT was our poorest performing application relative to the CPU. The Brook implementation is only .99 the speed of the CPU version. FFTW blocks the memory accesses to make very efficient use of the processor cache. (Without this optimization, the effective CPU MFLOPS drops to from 1224 to 204.) Despite this blocking, Brook is able to roughly match the performance of the CPU.

We can also compare the relative performance of the DirectX and the OpenGL backends. DirectX is within 80% of the performance of the hand-coded GPU implementations. The OpenGL backend however is much less efficient compared to the reference implementations. This was largely due to the need to copy the output data from the OpenGL pbuffer into a texture (refer to 4.2). This is particularly evident with SGEMV test which must perform a multipass reduction operation. The hand coded versions use application specific knowledge to avoid this copy.

We observe that, for these applications, ATI generally performs better than NVIDIA. We believe this may be due to higher floating point texture bandwidth on ATI. We observe 1.2 Gfloats/sec of floating point texture bandwidth on NVIDIA compared to ATI's 4.5 Gfloats/sec, while the peak, observable compute performance favors NVIDIA with 40 billion multiplies per second versus ATI's 33 billion.

In some cases, our Brook implementations outperform the reference GPU implementations. For the NVIDIA FFT results, Brook performs better than the reference OpenGL FFT code provided by ATI. We also outperform Moreland and Angel's NVIDIA specific implementation [2003] by the same margin. A similar trend is shown with SGEMV, where the DirectX Brook implementation outperforms hand-coded OpenGL. We assume these differences are due to the relative performance of the DirectX and OpenGL drivers.

These applications provide perspective on the performance of general-purpose computing on the GPU using Brook. The performance numbers do not, however, include the cost of `streamRead` and `streamWrite` operations to transfer the initial and final data to and from the GPU which can significantly affect the total performance of an application. The following section explores how this overhead affects performance and investigates the conditions under which the overall performance using the GPU exceeds that of the CPU.

5.2 Modeling Performance

The general structure of many Brook applications consists of copying data to the GPU with `streamRead`, performing a sequence of kernel calls, and copying the result back to the CPU with `streamWrite`. Executing the same computation on the CPU does not require these extra data transfer operations. Considering the cost of the transfer can affect whether the GPU will outperform the CPU for a particular algorithm.

To study this effect, we consider a program which downloads n records to the GPU, executes a kernel on all n records, and reads back the results. The time taken to perform this operation on the GPU and CPU is:

$$\begin{aligned} T_{gpu} &= n(T_r + K_{gpu}) \\ T_{cpu} &= nK_{cpu} \end{aligned}$$

where T_{gpu} and T_{cpu} are the running times on the GPU and CPU respectively, T_r is the transfer time associated with downloading and reading back a single record, and K_{gpu} and K_{cpu} are the times required to execute a given kernel on a single record. This simple execution time model assumes that at peak, kernel execution time and data transfer speed are linear in the total number of elements processed / transferred. The GPU will outperform the CPU when $T_{gpu} < T_{cpu}$. Using this relationship, we can show that:

$$T_r < K_{cpu} - K_{gpu}$$

As shown by this relation, the performance benefit of executing the kernel on the GPU ($K_{cpu} - K_{gpu}$) must be sufficient to hide the data transfer cost (T_r).

From this analysis, we can make a few basic conclusions about the types of algorithms which will benefit from executing on the GPU. First, the relative performance of the two platforms is clearly significant. The *speedup* is defined as time to execute a kernel on the CPU relative to the GPU, $s \equiv K_{cpu}/K_{gpu}$. The greater the speedup for a given kernel, the more likely it will perform better on the GPU. Secondly,

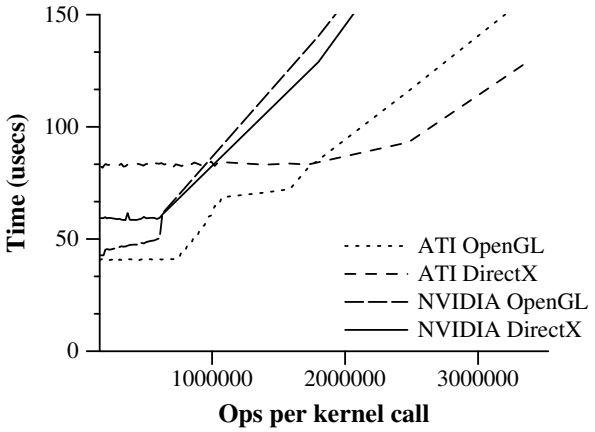


Figure 4: The average cost of a kernel call for various stream lengths with our synthetic kernel. At small sizes, the fixed CPU cost to issue the kernel dominates total execution time. The stair-stepping is assumed to be an artifact of the rasterizer.

an algorithm which performs a significant amount of computation relative to the time spent transferring data is likely to be dominated by the computation time. This relationship is the *computational intensity*, $\gamma \equiv K_{gpu}/T_r$, of the algorithm. The higher the computational intensity of an algorithm, the better suited it is for computing on the GPU. By substituting into the above relation, we can derive the relationship between speedup and computational intensity.

$$\gamma \quad > \quad \frac{1}{s-1}$$

The idea of computational intensity is similar to arithmetic intensity, defined by Dally et al. [2003] to be the number of floating point operations per word read in a kernel. Computational intensity differs in that it considers the entire cost of executing an algorithm on a device versus the cost of transferring the data set to and from the device. Computational intensity is quite relevant to the GPU which generally does not operate in the same address space as the host processor.

For our cost model, we assume that the parameters K_{gpu} and T_r are independent of the number of stream elements n. In reality, we find this generally not to be the case for short streams. GPUs are more efficient at transferring data in mid to large sized amounts. More importantly, there is overhead associated with issuing a kernel. Every kernel invocation incurs a certain fixed amount of CPU time to setup and issue the kernel on the GPU. With multiple back-to-back kernel calls, this setup cost on the CPU can overlap with kernel execution on the GPU. For kernels operating on large streams, the GPU will be the limiting factor. However, for kernels which operate on short streams, the CPU may not be able to issue kernels fast enough to keep the GPU busy. Figure 4 shows the average execution time of 1,000 iterations of a synthetic kernel with the respective runtimes. As expected, both runtimes show a clear *knee* where issuing and running a kernel transitions from being limited by CPU setup to being limited by the GPU kernel execution. For our synthetic application which executes 43 MAD instructions, the ATI runtime crosses above the knee when executing over 750K and 2M floating point operations and NVIDIA crosses around 650K floating point operations for both OpenGL and DirectX.

Our analysis shows that there are two key application properties necessary for effective utilization of the GPU. First, in order to outperform the CPU, the amount of work performed must overcome the transfer costs which is a function of the computational intensity of the algorithm and the speedup of the hardware. Second, the amount of work done per kernel call should be large enough to hide the setup cost required to issue the kernel. We anticipate that while the specific numbers may vary with newer hardware, the computational intensity, speedup, and kernel overhead will continue to dictate effective GPU utilization.

6 Discussion

Our computational intensity analysis demonstrated that read/write bandwidth is important for establishing the types of applications that perform well on the GPU. Ideally, future GPUs will perform the read and write operations asynchronously with the computation. This solution changes the GPU execution time to be max of T_r and K_{gpu}, a much more favorable expression. It is also possible that future streaming hardware will share the same memory as the CPU, eliminating the need for data transfer altogether.

Virtualization of hardware constraints can also bring the GPU closer to a streaming processor. Brook virtualizes two aspects which are critical to stream computing, the number of kernel outputs and stream dimensions and size. Multiple output compilation could be improved by searching the space of possible ways to divide up the kernel computation to produce the desired outputs, similar to a generalization of RDS algorithm proposed by Chan et al.[2002]. This same algorithm would virtualize the number of input arguments as well as total instruction count. We have begun incorporating such an algorithm into `brcc` with promising results.

In addition, several features of Brook should be considered for future streaming GPU hardware. Variable outputs allow a kernel to conditionally output zero or more data for each input. Variable outputs are useful for applications that exhibit data amplification, e.g. tessellation, as well as applications which operate on selected portions of input data. We are currently studying these applications and adding this capability into Brook through a multipass algorithm. It is conceivable that future hardware could be extended to include this functionality thus enabling entirely new classes of streaming applications. Secondly, stream computing on GPUs will benefit greatly from the recent addition of vertex textures and floating point blending operations. With these capabilities, we can implement Brook's parallel indirect read-modify-write operators, ScatterOp and GatherOp, which are useful for working with and building data structures stored in streams. One feature which GPUs support that we would like to expose in Brook is the ability to predicate kernel computation. For example, Purcell et al. [2002] is able to accelerate computation by using the GPU's depth test to prevent the execution of some kernel operations.

In summary, the Brook programming environment provides a simple but effective tool for computing on GPUs. Brook for GPUs has been released as an open-source project [Brook 2004] and our hope is that this effort will make it easier for application developers to capture the performance benefits of stream computing on the GPU for the graphics community and beyond. By providing easy access to the computational power within consumer graphics hardware, stream computing has the potential to redefine the GPU as not just a rendering engine, but the principle compute engine for the PC.

7 Acknowledgments

We would like to thank ATI and NVIDIA for providing access to their hardware, specifically Mark Segal and Nick Triantos. Tim Purcell provided our Brook ray tracing implementation. Bill Mark provided invaluable feedback on our submission. We would also like to thank the following people for their help in the design of the Brook language: Bill Dally, Mattan Erez, Tim Knight, Jayanth Gummaraju, Francois Labonte, Eric Darve, and Massimiliano Fatica from Stanford; Tim Barth and Alan Wray from NASA; Peter Mattson, Ken Mackenzie, Eric Schweitz, Charlie Garrett, Vass Litvinov, and Richard Lethin from Reservoir Labs.

The GPU implementation of Brook is supported by DARPA. Additional support has been provided by ATI, IBM, NVIDIA and SONY. The Brook programming language has been developed with support from Department of Energy, NNSA, under the ASCI Alliances program (contract LLL-B341491), the DARPA Smart Memories Project (contract MDA904-98-R-S855), and the DARPA Polymorphous Computing Architectures Project (contract F29601-00-2-0085). Additional support is provided by the NVIDIA fellowship, Rambus Stanford Graduate fellowship, and Stanford School of Engineering fellowship programs.

A BRCC Code Generation

The following code illustrates the compiler before and after for the SAXPY Brook kernel. The `__fetch_float` and `_stype` macros are unique to each backend. `brcc` also inserts some argument information in the end of the compiled Cg code for use by the runtime. The DirectX assembly and CPU implementations are not shown.

Original Brook code:

```
kernel void saxpy(float alpha, float4 x<>, float4 y<>,
                  out float4 result<>) {
  result = (alpha * x) + y;
}
```

Intermediate Cg code:

```
void  saxpy (float  alpha, float4  x, float4 y, out float4 result) {
  result = alpha * x + y;
}
void main (uniform float  alpha : register (c1),
           uniform _stype _tex_x : register (s0),
           float2 _tex_x_pos : TEXCOORD0,
           uniform _stype _tex_y : register (s1),
           float2 _tex_y_pos : TEXCOORD1,
           out float4 __output_0 : COLOR0) {
  float4  x; float4  y; float4  result;
  x = __fetch_float4(_tex_x, _tex_x_pos );
  y = __fetch_float4(_tex_y, _tex_y_pos );
  saxpy(alpha, x, y, result );
  __output_0 = result;
}
```

Final C++ code:

```
static const char* __saxpy_fp30[] = {
"!!FP1.0\n"
"DECLARE alpha;\n"
"TEX R0, f[TEX0].xyxx, TEX0, RECT;\n"
"TEX R1, f[TEX1].xyxx, TEX1, RECT;\n"
"MADR o[COLR], alpha.x, R0, R1;\n"
"END \n"
"##!!BRCC\n"
"##narg:4\n"
"##c:1:alpha\n"
"##s:4:x\n"
"##s:4:y\n"
"##o:4:result\n"
"##workspace:1024\n"
"##!!multipleOutputInfo:0:1:\n"
"",NULL};
void  saxpy (const float  alpha,
             const ::brook::stream& x,
             const ::brook::stream& y,
             ::brook::stream& result) {
  static const void *__saxpy_fp[] = {
     "fp30", __saxpy_fp30, "ps20", __saxpy_ps20,
     "cpu", (void *) __saxpy_cpu, NULL, NULL };
  static __BRTKernel k(__saxpy_fp);
  k->PushConstant(alpha);
  k->PushStream(x);
  k->PushStream(y);
  k->PushOutput(result);
  k->Map();
}
```

References

ATI, 2004. Hardware image processing using ARB_fragment_program. http://www.ati.com/developer/sdk/RadeonSDK/Html/Samples/OpenGL/HW_Image_Processing.html.

ATI, 2004. Radeon X800 product site. http://www.ati.com/products/radeonx800.

Bolz, J., Farmer, I., Grinspun, E., and Schröder, P. 2003. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph. 22*, 3, 917–924.

Bove, V., and Watlington, J. 1995. Cheops: A reconfigurable data-flow system for video processing. *IEEE Trans. on Circuits and Systems for Video Technology* (April), 140–149.

Brook, 2004. Brook project web page. http://brook.sourceforge.net.

Buck, I. 2004. Brook specification v.0.2. Tech. Rep. CSTR 2003-04 10/31/03 12/5/03, Stanford University.

Carr, N. A., Hall, J. D., and Hart, J. C. 2002. The Ray Engine. In *Proceedings of Graphics hardware*, Eurographics Association, 37–46.

Chan, E., Ng, R., Sen, P., Proudfoot, K., and Hanrahan, P. 2002. Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware. In *Proceedings of Graphics hardware*, Eurographics Association, 69–78.

Cooley, J. W., and Tukey, J. W. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation 19* (April), 297–301.

Dally, W. J., Hanrahan, P., Erez, M., Knight, T. J., Labont, F., Ahn, J.-H., Jayasena, N., Kapasi, U. J., Das, A., Gummaraju, J., and Buck, I. 2003. Merrimac: Supercomputing with Streams. In *Proceedings of SC2003*, ACM Press.

Dongarra, J. 2004. Performance of various computers using standard linear equations software. Tech. Rep. CS-89-85, University of Tennessee, Knoxville TN.

England, N. 1986. A graphics system architecture for interactive application-specific display functions. In *IEEE CGA*, 60–70.

Flisakowski, S., 2004. cTool library. http://ctool.sourceforge.net.

Frigo, M., and Johnson, S. G., 2003. benchFFT home page. http://www.fftw.org/benchfft.

FUCHS, H., POULTON, J., EYLES, J., GREER, T., GOLDFEATHER, J., ELLSWORTH, D., MOLNAR, S., TURK, G., TEBBS, B., AND ISRAEL, L. 1989. Pixel-Planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. In *Computer Graphics (Proceedings of ACM SIGGRAPH 89)*, ACM Press, 79–88.

GOKHALE, M., AND GOMERSALL, E. 1997. High level compilation for fine grained fpgas. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 165–173.

HARRIS, M. J., BAXTER, W. V., SCHEUERMANN, T., AND LASTRA, A. 2003. Simulation of cloud dynamics on graphics hardware. In *Proceedings of Graphics hardware*, Eurographics Association, 92–101.

INTEL, 2003. Intel software development products. http://www.intel.com/software/products/compilers.

INTEL, 2004. Intel math kernel library. http://www.intel.com/software/products/mkl.

KAPASI, U., DALLY, W. J., RIXNER, S., OWENS, J. D., AND KHAILANY, B. 2002. The Imagine Stream Processor. *Proceedings of International Conference on Computer Design* (September).

KESSENICH, J., BALDWIN, D., AND ROST, R., 2003. The OpenGL Shading Language. http://www.opengl.org/documentation/oglsl.html.

KHAILANY, B., DALLY, W. J., RIXNER, S., KAPASI, U. J., MATTSON, P., NAMKOONG, J., OWENS, J. D., TOWLES, B., AND CHAN, A. 2001. IMAGINE: Media processing with streams. In *IEEE Micro*. IEEE Computer Society Press.

KOZYRAKIS, C. 1999. A media-enhance vector architecture for embedded memory systems. Tech. Rep. UCB/CSD-99-1059, Univ. of California at Berkeley.

KRÜGER, J., AND WESTERMANN, R. 2003. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. Graph. 22*, 3, 908–916.

LABONTE, F., HOROWITZ, M., AND BUCK, I., 2004. An evaluation of graphics processors as stream co-processors. Unpublished.

LAWSON, C. L., HANSON, R. J., KINCAID, D. R., AND KROGH, F. T. 1979. Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. on Mathematical Software 5*, 3 (Sept.), 308–323.

LINDHOLM, E., KLIGARD, M. J., AND MORETON, H. 2001. A user-programmable vertex engine. In *Proceedings of SIGGRAPH 2001*, ACM Press/Addison-Wesley Publishing Co., 149–158.

MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. 2003. Cg: A system for programming graphics hardware in a C-like language. *ACM Trans. Graph. 22*, 3, 896–907.

MATTSON, P. 2002. *A Programming System for the Imagine Media Processor*. PhD thesis, Stanford University.

MCCOOL, M. D., QIN, Z., AND POPA, T. S. 2002. Shader metaprogramming. In *Proceedings of Graphics hardware*, Eurographics Association, 57–68.

MCCOOL, M., DU TOIT, S., POPA, T., CHAN, B., AND MOULE, K. 2004. Shader algebra. *ACM Trans. Graph.*.

MICROSOFT, 2003. High-level shader language. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/ directx9_c/directx/graphics/reference/Shaders/ HighLevelShaderLanguage.asp.

MOLNAR, S., EYLES, J., AND POULTON, J. 1992. PixelFlow: High-speed rendering using image composition. In *Proceedings of ACM SIGGRAPH 92*, ACM Press, 231–240.

MORELAND, K., AND ANGEL, E. 2003. The FFT on a GPU. In *Proceedings of Graphics hardware*, Eurographics Association, 112–119.

NVIDIA, 2004. GeForce 6800: Product overview. http://nvidia.com/page/geforce_6800.html.

OWENS, J. D., DALLY, W. J., KAPASI, U. J., RIXNER, S., MATTSON, P., AND MOWERY, B. 2000. Polygon rendering on a stream architecture. In *Proceedings of Graphics hardware*, ACM Press, 23–32.

PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. 2000. Interactive multi-pass programmable shading. In *Proceedings of ACM SIGGRAPH 2000*, ACM Press/Addison-Wesley Publishing Co., 425–432.

PERCY, J., 2003. OpenGL Extensions. http://mirror.ati.com/developer/SIGGRAPH03/ Percy_OpenGL_Extensions_SIG03.pdf.

PERONA, P., AND MALIK, J. 1990. Scale-space And Edge Detection Using Anisotropic Diffusion. *IEEE Trans. on Pattern Analysis and Machine Intelligence 12*, 7 (June), 629–639.

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 703–712.

RUSSELL, R. 1978. The Cray-1 computer system. In *Comm. ACM*, 63–72.

SANKARALINGAM, K., NAGARAJAN, R., LIU, H., HUH, J., KIM, C., D.BURGER, KECKLER, S., AND MOORE, C. 2003. Exploiting ILP, TLP, and DLP using polymorphism in the TRIPS architecture. In *30th Annual International Symposium on Computer Architecture (ISCA)*, 422–433.

SHERBONDY, A., HOUSTON, M., AND NAPEL, S. 2003. Fast volume segmentation with simultaneous visualization using programmable graphics hardware. *IEEE Visualization*.

SULLIVAN, W., WERTHIMER, D., BOWYER, S., COBB, J., GEDYE, D., AND ANDERSON, D. 1997. A new major SETI project based on Project Serendip data and 100,000 personal computers. In *Astronomical and Biochemical Origins and the Search for Life in the Universe, Proceedings of the Fifth International Conference on Bioastronomy*, Editrice Compositori, C. Cosmovici, S. Bowyer, and D. Wertheimer, Eds.

TAYLOR, M. B., KIM, J., MILLER, J., WENTZLAFF, D., GHODRAT, F., GREENWALD, B., HOFFMANN, H., JOHNSON, P., LEE, J.-W., LEE, W., MA, A., SARAF, A., SENESKI, M., SHNIDMAN, N., STRUMPEN, V., FRANK, M., AMARASINGHE, S., AND AGARWAL, A. 2002. The raw microprocessor: A computational fabric for software circuits and general purpose programs. In *IEEE Micro*.

THOMPSON, C. J., HAHN, S., AND OSKIN, M. 2002. Using modern graphics architectures for general-purpose computing: A framework and analysis. *International Symposium on Microarchitecture*.

WALD, I. 2004. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University.

WHALEY, R. C., PETITET, A., AND DONGARRA, J. J. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing 27*, 1–2, 3–35.

WOO, M., NEIDER, J., DAVIS, T., SHREINER, D., AND OPENGL ARCHITECTURE REVIEW BOARD, 1999. OpenGL programming guide.

2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

Virtual Thread: Maximizing Thread-Level Parallelism beyond GPU Scheduling Limit

Myung Kuk Yoon*, Keunsoo Kim*, Sangpil Lee*, Won Woo Ro*, and Murali Annavaram†
* *School of Electrical and Electronic Engineering, Yonsei University*
{*myungkuk.yoon, keunsoo.kim, madfish, wro*}*@yonsei.ac.kr*
† *Ming Hsieh Department of Electrical Engineering, University of Southern California*
annavara@usc.edu

***Abstract*—Modern GPUs require tens of thousands of concurrent threads to fully utilize the massive amount of processing resources. However, thread concurrency in GPUs can be diminished either due to shortage of thread scheduling structures (scheduling limit), such as available program counters and single instruction multiple thread stacks, or due to shortage of on-chip memory (capacity limit), such as register file and shared memory. Our evaluations show that in practice concurrency in many general purpose applications running on GPUs is curtailed by the scheduling limit rather than the capacity limit. Maximizing the utilization of on-chip memory resources without unduly increasing the scheduling complexity is a key goal of this paper.**

This paper proposes a Virtual Thread (VT) architecture which assigns Cooperative Thread Arrays (CTAs) up to the capacity limit, while ignoring the scheduling limit. However, to reduce the logic complexity of managing more threads concurrently, we propose to place CTAs into active and inactive states, such that the number of active CTAs still respects the scheduling limit. When all the warps in an active CTA hit a long latency stall, the active CTA is context switched out and the next ready CTA takes its place. We exploit the fact that both active and inactive CTAs still fit within the capacity limit which obviates the need to save and restore large amounts of CTA state. Thus VT significantly reduces performance penalties of CTA swapping. By swapping between active and inactive states, VT can exploit higher degree of thread level parallelism without increasing logic complexity. Our simulation results show that VT improves performance by 23.9% on average.

***Keywords*-GPU; GPGPU; Warp Scheduling; Virtual Thread (VT); Capacity Limit; Scheduling Limit;**

I. Introduction

Modern Graphics Processing Units (GPUs) are now widely used for executing general purpose applications. These general purpose applications are ported to be GPU kernels to create massive Thread Level Parallelism (TLP), which is then exploited by GPUs to concurrently execute thousands of threads. Each GPU has dozens of Streaming Multiprocessor cores (SM), where each SM can execute dozens of threads concurrently using Single Instruction Multiple Thread (SIMT) execution model. GPU kernels are divided into large Cooperative Thread Arrays (CTAs) or thread blocks. CTAs, in turn, are divided into smaller groups of threads called *warps* or *wavefronts*. GPUs support fast context switching between warps to allow execution of other warps when one warp stalls, so as to hide instruction latencies [1], [2]. To support fast context switching, GPUs have a large register file to store the architectural context of multiple concurrent warps without the need to save and restore the context.

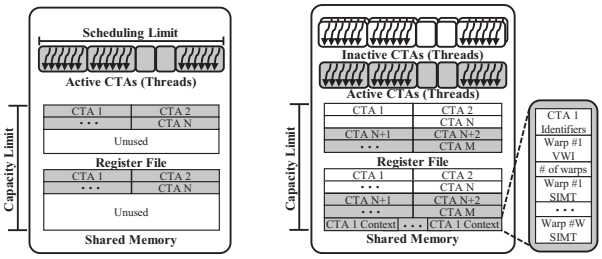
The size of each CTA is usually decided by the programmers [3], [4], however the number of CTAs that can be concurrently executed on an SM is determined by various hardware resource constraints: the number of issuable CTAs, the number of issuable threads, the size of the register file, and the size of shared memory [5], [6]. In this paper, we divide the constraints into two groups: *scheduling limit* and *capacity limit*. Scheduling limit constrains the maximum number of issuable CTAs and threads due to the limited thread concurrency that the hardware can support. On the other hand, capacity limit constrains the maximum number of CTAs due to limited register file and limited amount of shared memory for storing concurrent thread contexts. If the cumulative resource demand of all the assigned CTAs reaches any one of these limits, no further CTAs can be dispatched to SMs even though the other resources may still be available.

To understand the severity of both these limits, scheduling and capacity limits, we measured the number of CTAs assigned to an SM across a wide range of GPU applications (details in Section III). We observed that 18 applications out of 35 applications that we analyzed have at least one kernel that reaches the scheduling limit first when the applications are launched on the Fermi architecture [7], [8]. Among these 18 applications, 10 applications reach thread count limit, and the other 8 applications reach CTA count limit. In these benchmarks, on average, 49.8% of the register file, and 84.6% of the shared memory are unused on Fermi. The resource underutilization problem becomes even more acute in newer architecture such as Kepler [9]. Kepler increased the maximum thread count to 2,048 (compared to 1,536 in Fermi) and the maximum CTA count was increased to 16 (compared to 8 in Fermi). Kepler also increased the total number of registers to 64K $\times$ 32-bit registers, compared to 32K $\times$ 32-bit registers in Fermi [7], [8], [9]. Even though the scheduling limit was increased the register file grew larger. As a result, in the case of the Kepler architecture, 27 applications out of 35 applications reach the scheduling limit. Among these 27 applications, 22 applications cannot dispatch more CTAs onto SMs due to the thread count limit, and 5 applications cannot dispatch more CTAs due to the CTA count limit. In summary, the scheduling limit, especially the thread count limit, is a major TLP limiting factor on GPUs.

Figure 1(a) shows a conceptual view of resource underutilization due to the scheduling limit on GPUs. The thread counts and CTA counts that are allowed in each GPU were fully occupied. But the register file and shared memory structures were underutilized. In fact, the underutilization of register file has also been observed in many prior studies [10], [11], [12], [13], [14], [15]. Register file and shared memory are valuable resources that can be used to further improve parallelism but unfortunately scheduling limits prevent these resources from being fully utilized.

Prior research studies [11], [16], [17], [18], [19] have

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.59



(a) Baseline GPU (b) VT Architecture

Figure 1. Conceptual View of Resource Management on Baseline GPU and VT Architecture

shown increasing TLP is beneficial for many GPU applications. Our evaluations (shown in Section V) also confirm that additional threads can improve performance. Dispatching more CTAs can improve performance and resource utilization, however the scheduling limit cannot be scaled easily due to several reasons. First, it requires additional scheduling support such as larger instruction buffer, scoreboard, and SIMT stack. These structures cause area and power overheads on GPUs [20], [21], [22], [23], [24]. Second, implementing the warp scheduler for a large number of warps is a challenging design problem [25], [26]. Every cycle, the scheduler must access the scoreboard to verify ready warps and a subset of ready warps must be selected for issue. Therefore, scaling scheduling limit increases the scheduling complexity and may increase scheduling latencies.

As a cost-effective alternative, we propose a Virtual Thread (VT) architecture which relaxes the scheduling limit and forces the capacity limit to be the only concurrency limiter. To restrict the corresponding increase in scheduler complexity, we propose to keep the physical scheduling limit unchanged. The scheduler still handles the same number of CTAs it is provisioned to handle in the baseline architecture without VT. Figure 1(b) shows the conceptual view of resource management on VT. In VT, all the CTAs are placed either in active or inactive state. The scheduler only deals with active CTAs which are still confined by the baseline scheduler limit. When all the warps in an active CTA hit a long latency operation, such as waiting for a response from the global memory, the active CTA is switched to inactive state. After the state change, another ready CTA is swapped into the active state. As we will describe in detail shortly, an inactive CTA will not be part of any scheduling decisions and hence the scheduling complexity is not increased.

Given that the total CTAs (active and inactive) are still bounded by the capacity limit, there is no need to save and restore the entire architectural state while switching CTAs between active and inactive states. In particular, the large register file and shared memory state of the swapped CTAs [27] stay unperturbed, and only a small amount of per-CTA state data is saved when a CTA is moved from active to inactive state, and the small amount state is restored when an inactive CTA is switched back to active state.

While increasing TLP generally helps performance, increasing TLP unchecked, particularly in the presence of global memory stalls may increase memory contention [5], [28]. To overcome this problem, VT uses a memory request prioritization technique which is inspired by prior works [29], [30]. By exploiting the increased TLP and managing the memory contention, VT increases average performance by 23.9%. VT's performance is only 3% lower than a hardware-intensive traditional approach, called MAX in our evaluations, that increases the scheduling limit through more scheduling hardware.

The remainder of this paper is organized as follows. Section II presents the GPU programming model and the baseline GPU architecture. In Section III, we present motivational data for this work. In Section IV, we propose the VT architecture. The experimental results of VT are presented in Section V. In Section VI, we present related work. Finally, we conclude in Section VII.

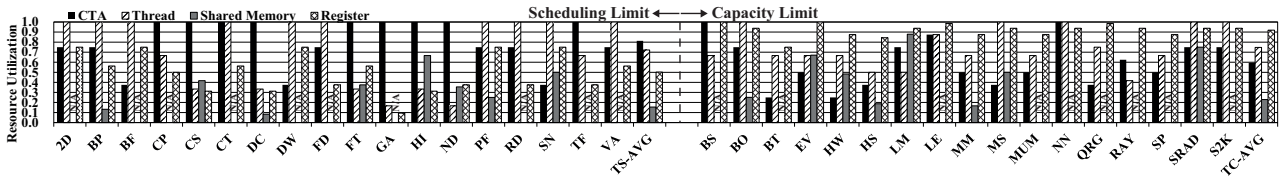
II. Background

For consistency, we use NVIDIA terminology to refer to various hardware and software components throughout this paper [7], [8], [9], [31]. Many general purpose GPU applications typically have multiple compute kernels which are the primary computational blocks that can be executed on GPUs. These kernels are comprised of tens of thousands of threads which are organized as CTAs [31]. Each CTA is further sub-divided into a collection of threads called a warp, which is the minimum scheduled unit of work in GPUs [31]. The threads in each CTA can synchronize via hardware barriers. GPUs are provisioned with a range of specialized memories. The largest memory is global memory which is accessible to all threads in a kernel and accessing the global memory space takes hundreds of clock cycles [26], [32]. Then there is an on-chip shared memory which is available for inter-thread communication within a CTA, but different CTAs cannot access each other's shared memory resources [33]. Lastly, each thread can access private local memory.

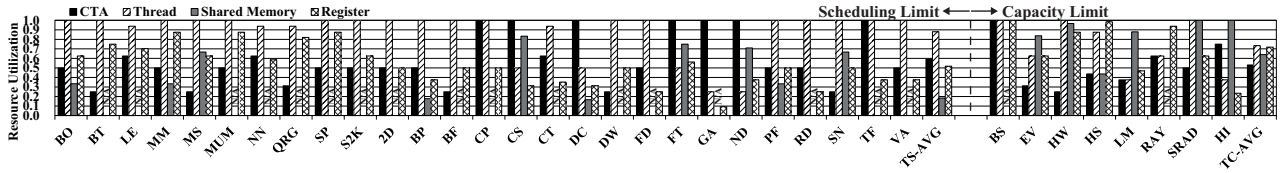
GPUs consist of thousands of processing units to execute a massive number of threads concurrently. For example, NVIDIA Kepler GPU includes 2,048 processing units in each Streaming Multiprocessor (SM or SMX) to execute up to 30,720 (15 SMXs $\times$ 2,048) threads concurrently [9]. The CTA scheduler dispatches CTAs to SMs, and the number of CTAs assigned to an SM is constrained by various factors: the number of issuable CTAs, the number of issuable threads, the size of the register file, and the size of the shared memory [5], [6]. We classify TLP limiting constraints into two categories: *scheduling limit* and *capacity limit*.

The scheduling limit is due to the logic complexity of managing a massive number of threads and CTAs. For instance, one scheduling limit that curtails the number of issuable warps is the number of program counters since GPUs have to provide a separate program counter per each concurrent warp. The capacity limit is due to the cumulative storage demand from all the assigned CTAs. For instance, the number of architected registers used in a warp may determine the total number of warps that can be assigned per SM since the cumulative register usage cannot exceed the available register file size. The amount of shared memory used by each warp may also limit the total number of warps assigned as the total memory usage across all warps must fit the available shared memory. As can be seen from these examples, TLP can be curtailed due to a number of competing design constraints.

Due to varying application demands, some resource limits may be reached first while there are plenty of other available resources. If the total resource usage of current in-flight CTAs reaches the limit of any one of various resource constraints, no more CTAs can be scheduled even though



(a) Resource Utilization on Fermi Architecture



(b) Resource Utilization on Kepler Architecture

Figure 2. Resource Utilization on Various GPU Architectures

all other resources are still available. For example, NVIDIA Kepler GPU has 64K × 32-bit register file and 48KB shared memory per SMX, categorized as capacity limit in this work. In addition, each SMX allows up to 2,048 threads or up to 16 CTAs (whichever limit reaches first) to be issued concurrently on each SMX [9], categorized as scheduling limit in this work. Let us assume that an example application which consists of hundreds of CTAs is launched onto the Kepler GPU. If each CTA contains 256 threads and each thread only needs 10 registers then the number of issuable CTAs is limited to 8 due to the the maximum thread count (scheduling limit), even though there are plenty of available registers. A well designed system should be able to maximize all the available resources to achieve the highest throughput.

While resource underutilization may appear to be a design imbalance that may be fixed by either reducing the resources or increasing the scheduling structure sizes, we should point out that GPU designs are tuned for highly parallel applications, such as graphics computing [34], [35], [36]. The use of GPUs for general purpose computing creates the imbalances [37], [38]. In spite of these concerns, GPUs are rapidly being deployed for general purpose computing. For general purpose computing with diverse application demands, it is much more challenging to tune the size of the structures at design time. Rather we believe that existing GPU designs can be enhanced with features such as VT to improve their execution efficiency even for general purpose computing.

III. Understanding GPU TLP Limits

To quantify resource utilization in GPUs, we characterize the resource demands of various applications in two different GPU configurations, which are similar to Fermi and Kepler architectures. Detailed simulation environment is presented on Section V. Figure 2 shows the fraction of available SM resources used by each application. We consider four different limitations: the maximum number of CTAs, the maximum number of threads, the total register file usage, and the total shared memory usage [5], [6]. The first two limits are scheduling limits and the last two are capacity limits. In the figure, we divide applications into two types. Type-C applications which are shown to the right of the dotted line in Figure 2(a) are constrained by the capacity limit. Most Type-C applications utilize almost the entire 32K × 32-bit register file in Fermi, except that *LM* is limited by the shared memory capacity. On average, 91.9% of register file and 22.8% of shared memory are occupied for the Type-C applications. Therefore, the Type-C applications require more on-chip memory capacity for storing thread contexts to improve TLP regardless of the other limiting factors. In contrast to the Type-C applications, the scheduling limit is the bottleneck in scaling TLP for the Type-S applications, which are shown to the left of the dotted line in the figure. For these applications the number of allowable threads or CTAs is the TLP limiting factor. As these applications do not fully utilize the on-chip memory, a large fraction of register file and shared memory are wasted. As shown in Figure 2(a), only 50.2% of register file and 15.4% of shared memory space are used in these applications.

The resource underutilization problem becomes even more magnified when the available resources are scaled in future GPUs. From Fermi to Kepler generation [8], [9], [31], per-SM CTA limit is increased from 8 to 16, thread limit is increased from 1,536 to 2,048, and the register file size is doubled. Figure 2(b) shows the resource utilization with the Kepler architecture for the same set of benchmarks. Compared to Fermi, 27 applications are now classified as Type-S; 22 of these applications reach the thread count limit and 5 applications reach the CTA count limit. Consequently, 48.2% of register file and 81.6% of shared memory space are underutilized in Kepler. From these results, we surmise that as resources are scaled in future for many general purpose applications running on GPUs the TLP limiting factor is the scheduling limit (especially thread count limit) rather than the capacity limit. Therefore, relaxing the scheduling limit constraint can extract more TLP and improve the utilization of the on-chip memory structures.

To investigate the extent of additional scheduling resources needed to reach the capacity limit, either register file capacity or shared memory capacity, we calculate the maximum number of threads that can be assigned in the two generations of GPU architecture: Fermi, and Kepler. Figure 3 shows the fraction of additional threads required to reach the capacity limit (either the register file or shared memory whichever saturates first) for the applications classified as Type-S in Figure 2. In case of Fermi, on average 38.2%

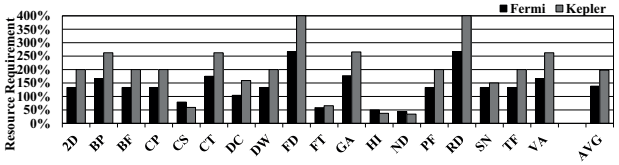


Figure 3. Thread Counts to Saturate Register File/Shared Memory for Type-S Applications

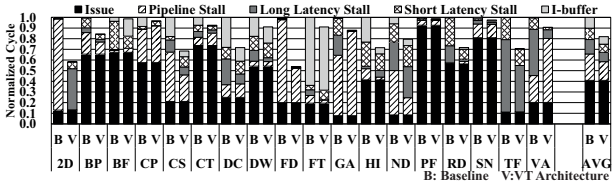


Figure 4. Cycle Distribution for Type-S Applications

more threads are required to reach the capacity limit. In Kepler, on average 97.7% more threads are needed to fully utilize on-chip storage, and 300.0% more threads are needed in the worst case.

However, implementing GPUs which can support 3× more threads is a non trivial challenge due to the need for providing additional hardware resources such as instruction buffers, scoreboards, and SIMT stack. Based on the previous studies, these hardware logic blocks increase power and area overheads [21], [22], [23], [24], [20]. In addition to the hardware overheads, implementing a warp scheduler for the large number of threads is also a difficult problem. In modern GPUs, the scheduler accesses the scoreboard to find the ready warps and selects the highest priority warp among the ready warps every cycle [25], [26]. Therefore, if the number of concurrently running threads is increased significantly, the scheduler complexity is increased and the searching time for an issuable warp can be increased.

A. Advantages of Higher TLP

GPUs exploit high TLP to hide processing latency of individual warps [1], [28]. If a warp is stalled by a long latency memory access or data dependency, then warp schedulers issue another ready warp from the warp pool so that execution of warps are interleaved. The effectiveness of stall hiding depends on the number of available warps in the warp pool, which is the key reason why GPUs require a large number of concurrent threads [5], [11], [19], [39].

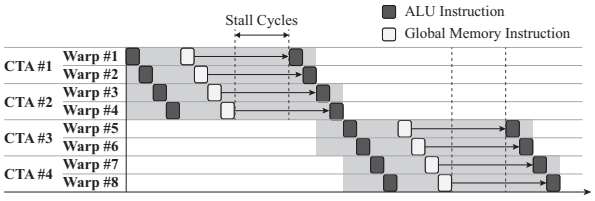
To quantify whether the existing level of TLP is already sufficient to hide long latency memory stalls, we classified the scheduling cycles into several categories. The first bar in each group in Figure 4 (B for the baseline architecture) shows the breakdown of scheduling cycles for the Type-S applications on the Fermi architecture (other bar in each group will be described in Section V). We divide the cycles into two different types: issue and stall. The bottom category shows the fraction of cycles where the scheduler is able to find a warp which is ready for issue. All the other categories are stall categories where the scheduler cannot issue an instruction. The warp schedulers cannot issue instructions during 59.1% of total execution cycles. Hence, more than half of the available issue bandwidth is unused.

To better understand the reasons for these stalls, we further divide the stalls into four different types: pipeline, long latency, short latency, and I-buffer stalls. Pipeline stall (the second component from the bottom), which takes 24.8% of total cycles, indicates that the schedulers have at least one ready warp, however these ready warps could not be issued since the functional units, such as load/store units, do not accept more instructions. In particular, we observed that the majority of pipeline stalls are caused by the memory subsystem saturation, which occurs when load/store units are not able to handle more memory requests [29]. Long latency stall accounts for 13.6% of missed scheduling opportunities, which occurs when the warps in the scheduler pool are all waiting for memory responses. Thus 38.4% of the total scheduling cycles are wasted due to memory related stalls (pipeline and long latency stalls). Recall that GPUs fundamentally rely on TLP to hide memory latency related stalls. Hence, 38.4% stalls could have been potentially avoided with increasing TLP. Finally, short latency stall takes 10.7% of total cycles, which happens when no more instructions are issued due to preceding non-memory dependent instructions. The other reasons including instruction fetch buffer stall (I-buffer), which indicates next instructions of all warps are not yet fetched, are only 10.0%.

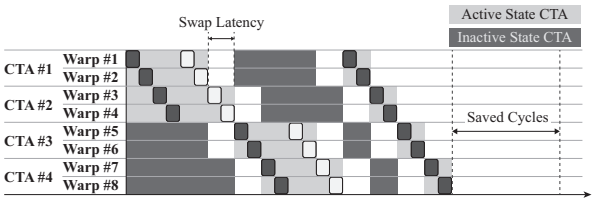
IV. Virtual Thread Architecture

As seen from Figure 4 improving TLP still has a significant potential for higher performance with better latency hiding. The data shown in Figure 2(b) shows that 48.2% of register file and 81.6% of shared memory space are underutilized in Kepler for a majority of benchmark applications. Thus the option we explore to increase TLP is to allow more threads into the GPU to fill up the available register file or shared memory. At the same time we do not increase the scheduling limit to deal with the increased thread count. Scaling the scheduling limit using more program counters, larger scheduling windows, and deeper fetch and scheduling queues is non-trivial due to power and area limitations as shown in prior works [22], [25], [26]. As a cost-effective alternative, we propose the Virtual Thread (VT) architecture. The proposed architecture takes advantage of the fact that on-chip memory capacity is not a limiting factor for most applications. With VT, CTAs are allocated on each SM up to the capacity limit regardless of the scheduling limit. Thus the number of CTAs allocated to each SM (and consequently the number of threads) will either cumulatively occupy the entire register file or the entire shared memory, whichever limit reaches first. Based on the data in Figure 2, the register file limit is reached earlier than the shared memory for the majority of the applications.

To keep the scheduling limit unchanged, the VT architecture maintains only a fraction of CTAs in active state which will be managed within the current scheduling limit, and the other CTAs are left in inactive state. When all the warps from an active CTA stall due to a long latency memory operation, VT swaps out the stalled CTA into the inactive state and brings in another ready CTA. We exploit the fact that even though the swapped out CTA's architected state is in the register file and shared memory, the incoming CTA does not need to use any of those registers or shared memory. In fact all the allocated CTAs' architected state fit within the capacity limit, irrespective of whether a CTA is active or inactive. Thus rather than naively saving and restoring CTA's register or memory state on a context switch, each CTA needs to save a much smaller per-CTA state information. As we will describe shortly, the per-CTA state information



(a) Execution Order on Baseline GPU



(b) Execution Order using VT Architecture

Figure 5. Execution Order with and without VT

is composed of warp identifiers, CTA identifiers, and SIMT stack including the program counter.

Given the limited amount of per-CTA state information that needs to be saved and restored, it is feasible to save the per-CTA state information in the shared memory itself, which is often even more underutilized than the register file, as can be seen from Figure 2. Shared memory access latency is significantly shorter than global memory [16], [40], [41], and therefore CTA swapping latency is negligible compared to the global memory access latency stalls that are suffered by a stalled CTA. This dynamic context switching of CTAs increases TLP allowing VT to improve performance. If there is not enough shared memory to store the context information of all the CTAs that can be assigned with VT, we can scale back VT to support fewer CTAs.

Figure 5 illustrates an example execution with and without VT. In this example, we assume for illustrative purposes that each SM can issue two CTAs concurrently due to the scheduling limit, however the SM has unused registers and shared memory which can support two additional CTAs. Figure 5(a) shows an execution order of conventional GPU without VT. After issuing a series of instructions from each warp in round-robin order, each warp hits a global memory access stall. Eventually both CTAs are stalled, but no new CTAs can be brought in. Only after a CTA completes execution, the SM releases CTA resources and then assigns new CTA into the SM.

With VT, we reduce the long latency stalls using more CTAs as illustrated in Figure 5(b). At the beginning of execution, all four CTAs are assigned to the SM ignoring the fact that only two CTAs can be actively scheduled. Each of four CTAs gets its own register file and shared memory allocation to improve resource utilization. At the start of execution to honor the existing scheduling limit, only two CTAs are placed in the active state and the scheduler only deals with active CTAs. After detecting Warp #1 and #2 from CTA #1 are all waiting for returning responses from the global memory, CTA #1 is swapped out and CTA #3 is swapped in. For illustration purposes, the swapping latency is entirely hidden by the execution cycles of CTA #2. When CTA #2 is stalled, it is swapped with CTA #4. The warp scheduler issues instructions from the newly activated CTAs (#3 and #4). Eventually warps from the newly activated CTA #3 and #4 issue global memory access instructions and then the VT architecture swaps out CTA #3 and #4 with CTA #1 and #2 since memory requests from CTA #1 and #2 are processed. To summarize, the long latency operations are more effectively hidden and Memory Level Parallelism (MLP) is increased due to additional CTAs.

A. Architectural Support for VT

In this section, we describe the microarchitectural support for VT. The number of microarchitectural changes are quite minimal and are highlighted in Figure 6. In order to manage CTA context swap, we add Virtual Thread Controller (VTC) logic which is shown in Figure 7. To provide the complete operational detail of VT architecture, we describe how each pipeline stage works in baseline and highlight if there are any changes that are needed for the VT architecture.

Fetch and Decode Stages: In our baseline, a Fermi-like architecture, the fetch stage of the pipeline selects one of the warp program counters and fetches an instruction from that warp into an instruction fetch buffer. The instruction fetch buffer is indexed using the warp ID to locate the buffer entries. The VT architecture enables many more CTAs (and hence more warps) to be assigned to the SM up to the capacity limit. Thus indexing the instruction fetch buffer with a larger warp ID is not feasible without unduly increasing the complexity of the fetch buffer design. Hence, whenever a CTA is swapped out (referred to as CTA_{old} for simplicity) and a new CTA (CTA_{new}) is brought in, we re-assign warp ids of CTA_{old} to CTA_{new}. We will use the term Physical Warp ID (PWI) to refer to the fact that PWI is used to physically index the instruction fetch buffers. Thus warps from CTA_{new} essentially use the same PWIs as CTA_{old}.

When a CTA swaps out, the PC of each warp in the the CTA_{old}, which is usually stored within the SIMT stack, is stored in a dedicated context storage area within the shared memory. The size and content of each CTA context will be described in detail shortly. After saving the CTA_{old} context, PCs of all the warps in the CTA_{new} are now restored into the PC array by reading the new context from the shared memory. Then the fetch buffers for CTA_{old} are flushed. After flushing the buffers, warps from CTA_{new} fetch instructions from their own PCs. Since these warps reuse the PWIs, they simply bring instructions into the same entries as warps from the CTA_{old}.

Issue Stage: Our baseline architecture uses a scoreboard to keep track of instruction dependencies. The scoreboard can be implemented in multiple ways. Without loss of generality in our baseline, the scoreboard is implemented as an array of bit-vectors, where each bit-vector in the array tracks the dependencies of one warp. Each bit-vector in the scoreboard array is thus accessed by the warp id. As soon as a warp instruction is issued, the destination register number of that warp is marked as busy in the corresponding bit-vector of that warp. Any younger instruction that needs a register marked as busy in the scoreboard is stalled from issuing. When the instruction completes its execution and writes back its results to the register file, it then resets the busy bit in the scoreboard. The scoreboard is accessed every cycle to decide which warp is ready for issue [25].

Rather than increasing the scoreboard size to accommodate more warps, VT reuses the scoreboard entries from

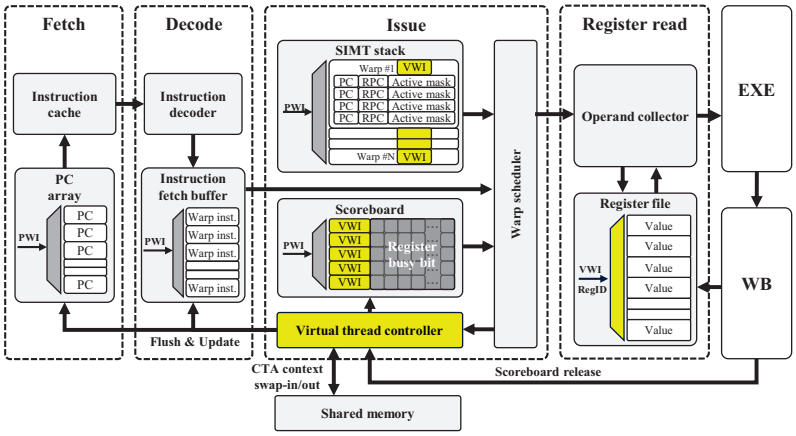


Figure 6. GPU Architecture with VT

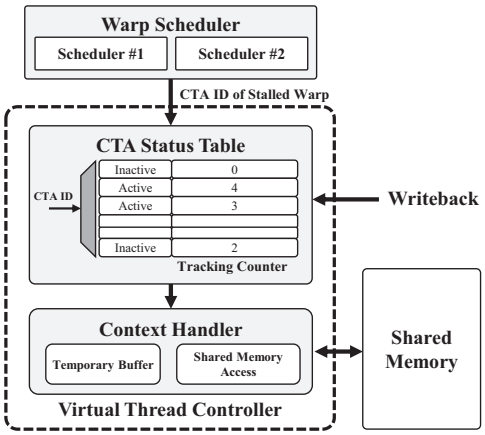


Figure 7. Virtual Thread Controller

CTA_{old} for CTA_{new}. After CTA_{old} is switched out, each bit-vector associated with the warps in CTA_{old} are all reset to zero. At that time, VT can reuse the scoreboard bit-vectors for warps from CTA_{new}. VT uses the PWI to index the scoreboard bit-vector. Since VT reuses the PWIs for CTA_{new}, the newly initiated warps reuse the the same bit-vectors.

Note that some operations from CTA_{old} that are still in the pipeline may be completed after CTA_{old} is swapped out. Generally when an instruction completes it would update the scoreboard. However, in VT any warps from the swapped out CTA_{old} do not update the scoreboard. Instead they simply write to the register file associated with CTA_{old}. Hence, in VT any swapped out CTA_{old} operations are not allowed to access or clear the scoreboard bits since the scoreboard vectors are reassigned to CTA_{new}. This process is described in more detail in the writeback stage discussion.

Register Read Stage: Once an instruction is issued the input operands are read from the register file and stored in the operand collector buffer. In the baseline architecture, each warp's register file base location is computed using the PWI. With VT, the PWIs are reused and hence warps from CTA_{new} cannot use them to access their own local registers. We introduce Virtual Warp ID (VWI) to each warp that is assigned to an SM. VWI is a unique warp identifier across all the assigned warps to the SM, including both active and inactive CTAs. Only when a CTA finishes execution its VWIs are released and reused for another CTA. Hence, context switching a CTA does not release the VWI. Initially when VT assigns the maximum number of CTAs that can fit within the capacity limit, VWIs are assigned sequentially starting from the first CTA. For instance, if there are five CTAs with eight warps per CTA, then VWIs 0-7 are assigned to the eight warps in CTA#0, VWIs 8-15 are assigned to the eight warps in CTA#1 and so on.

The VT architecture uses VWIs to access the register file, instead of PWIs. Since the baseline architecture allows only 48 physical warps they can be encoded in 6 bits. However, the VT architecture requires more bits to encode the VWI. As shown in our motivation results earlier, most benchmarks reach the capacity limit well below 256 warps (5× the baseline) in total. Thus we limit the total number of virtual warps to 256 which can be encoded using 8 bits. Note that the limitation of 256 warps is an empirically-driven choice and the primary impact of using a larger number of VWIs is that the VWI encoding requires additional bits.

The VT architecture simply fills up the underutilized registers to pack more warps. Hence using the VWIs, instead of the PWIs, does not perturb the register allocation process or the number of register accesses of the baseline architecture. In fact the total number of register accesses remains the same over the entire kernel execution with or without VT.

Writeback Stage: After an instruction is completed in the functional unit, VWI is also used to determine the location in register file where the new value of destination register should be written. Typically the destination register is also released from the scoreboard at that time. As noted earlier, VT resets the scoreboard bit-vector from CTA_{old} and reassigns it to CTA_{new} on initiating a context switch. Hence, when any pending operations from CTA_{old} complete, they should not be allowed to clear the scoreboard bits since they are reassigned to CTA_{new}. To handle this scenario correctly, we store VWI for each warp in the scoreboard. We use PWIs to index the scoreboard array. But during writeback we compare the VWI field of the scoreboard entry with VWI of the completing instruction to make sure that the entry belongs to the same warp. If two VWIs mismatch, this means the current warp belongs to an inactive CTA, therefore the scoreboard is not updated by CTA_{old}. While CTA_{old} may not update the scoreboard it is still necessary to keep track of the progress of CTA_{old}'s long latency operations so as to recognize when a stalled CTA is ready for execution again. For this purpose, we use a CTA status table incorporated within the virtual thread controller. The controller structure and operation will be described in detail shortly.

Branch Divergence Handling: The baseline architecture uses SIMT stack to store next PC and reconvergence PC to handle branch divergence [21], [42], [43]. As discussed in prior work [21], a portion of the SIMT stack per each warp may be cached on-chip for fast branch handling and any entries that do not fit the SIMT stack are stored in global memory. The cached SIMT stack usually holds a fixed number of entries per each warp. In our implementation, we assume that each warp has a 4-entry SIMT stack that is

cached and any warp that requires more than four entries will spill to the global memory. In practice across a broad range of applications we studied, the 4-entry SIMT stack never overflowed. With VT, only the cached portion of the SIMT stack is saved/restored in shared memory as part of the CTA context. The global memory portion of the SIMT stack, if any, is left untouched. We simply need to use VWI, instead of PWIs, to access the global memory portion of the SIMT stack to uniquely identify each CTA's complete SIMT stack.

CTA Swap Operation: To manage CTA context swap, the virtual thread controller (VTC), which is a schedule monitoring logic is inserted in the VT architecture. VTC keeps track of the state of all CTAs in order to determine which CTA can be brought back to active from inactive state or vice versa. VTC employs a table called CTA status table, as shown in Figure 7. The table is indexed by CTA ID, and each entry of the table has two fields. One-bit status field indicates the corresponding CTA is whether active or inactive. The second field is called tracking counter, and the counter is used for two different purposes depending on the CTA state. If the CTA is active, it is used for indicating the number of stalled warps that belong to that CTA. The counter is incremented when the warp scheduler detects a warp is stalled by a long latency memory operation and decremented when the warp is released from the stall. The warp schedulers send associated CTA ID of the warp to VTC for this operation. If the counter value reaches the number of warps in the CTA, the CTA is marked for swap out. Note that the number of warps per CTA is precomputed at kernel launch and is already available in the SM for tracking CTA progress.

When a CTA is marked for swap out, the necessary context for restarting its execution is saved in the shared memory through the context handler. The per-CTA state information includes VWI, CTA ID, and SIMT stack (including PC). All these elements are fixed size and hence the per-CTA context size is also fixed. Furthermore, the context information is saved in consecutive memory location, the shared memory stores generated due to context saving are all coalesced memory accesses. The SIMT stacks entries are first moved into a temporary buffer and are then scheduled for storing into the shared memory. The temporary buffer allows the context handler to save the SIMT stack outside of the CTA switching critical path. Finally the SIMT stack, instruction buffer, program counter array, and scoreboard entries of all warps belonging to the swapped out CTA are invalidated so the new CTA can reuse these structures. More details of the context saving overhead will be discussed in Section IV-C.

Before invalidating the scoreboard, the tracking counter associated with the swapped-out CTA is initialized with the number of write-pending registers that belong to the CTA in order to handle the pending operations from the swapped-out CTA. The number of pending register writes can be obtained by counting the busy bits in the scoreboard bit-vector entries of all the warps that belong to the swapped-out CTA. Using the counter, VTC is able to collectively track whether all current in-flight instructions of the swapped-out CTA have completed. When an instruction belonging to a swapped-out CTA is completed at the writeback stage, the tracking counter is decremented. An inactive CTA becomes a candidate for scheduling once the counter value is zero. All such ready but pending CTAs are candidates for execution once a currently active CTA is marked for swap out.

In our current implementation, we use a first-ready-first-serve policy to select the CTA for swapping. Once the CTA is selected then VTC reads its context from the context space in the shared memory. VTC does not require additional access ports for the shared memory. VTC shares access port with load/store units and requests are scheduled so that demand requests have always a higher priority than the CTA swap operation. We observed that such additional accesses have only negligible performance impact on the original demand requests.

B. Shared Memory Management

Figure 1(b) shows how the shared memory space is managed with VT. We partition the shared memory into two regions: data and context. The data region is used for data which is allocated for each CTA, and swapped-out CTA states are stored in the context region. Note that the memory management is simplified by allocating data region from low to high shared memory address and context region from high to low address.

In VT, the context size of each CTA is fixed which simplifies shared memory management, at the expense of some wasted space. At the beginning of a kernel launch, the GPU runtime calculates the number of CTAs needed to reach the capacity limit, as well as the number of warps in each CTA. Once the total number of virtual warps are calculated then the runtime reserves shared memory which is equal to the number of virtual warps $\times$ the context size of each warp. As stated earlier, the context of each warp is primarily the 4-entry SIMT stack, VWI, and CTA identifiers.

C. Cost of Context Swapping

In this section, we discuss the size and timing overheads for the CTA swap operation. The details of CTA-specific data structures in existing GPUs are not disclosed in any publicly accessible literature, and we believe they are implementation dependent. Therefore, the following discussion is based on our baseline design assumptions. However, we have evaluated performance impact with various cost scenarios in Section V, and conclude that the performance impact of context saving/restoring of SIMT stack is minimal.

We first calculate the total size of state data for a CTA. The following is the list of CTA state data that we store/restore on a CTA switch in our implementation.

Virtual Warp Identifiers: For each swapped out CTA, we need to store W VWIs, where W is number of warps in each CTA. In VT, instead of saving all VWIs, we save the first VWI and the number of warps in each CTA since VWIs are sequentially assigned. Assuming the first VWI requires N bits, the total size for warp identifiers is $N + log_2 W$ per CTA. In our particular baseline, the VT architecture has a maximum of 256 virtual warps and each CTA can have up to 32 warps $(1,024/32)$ [9]. Hence, an 8-bit VWI of the first warp in the CTA, and a 5-bit for the number of warps in CTA must be stored; a total of 13 bits.

CTA Identifiers: Each CTA has a unique identifier based on three different dimensional indices, blockIdx.x, blockIdx.y, and blockIdx.z [31], [44]. Based on PTX specification [33], we assume that each index requires 32 bits. Therefore, the total size of CTA identifiers is 96 bits per CTA.

SIMT Stack: SIMT stack maintains thread divergence information of a warp [42], [43]. We assume a stack entry

consists of thread active mask and two program counter fields [43]. Each stack entry requires 32 bits for a 32-lane active mask, 64 bits to save the Program Counter (PC), and another 64 bits to save the Reconvergence PC (RPC). In VT, the SIMT stack size is fixed at D entries (D=4 in our base machine). Then, total size of stack is $160 \times D$ bits. Given W warps in a CTA, the total size of stack for the CTA is about $160 \times D \times W$ bits.

In summary, the state data size for a CTA is as follows:

$$State(bits) = (N + log_2 W) + 96 + (160 \times D \times W) \quad (1)$$

From Equation 1, we estimate CTA swapping latency. Based on the previous studies [5], [31], we assume the shared memory has total 32 banks and each bank can read 32-bit data per one or two clock cycles. Therefore, the total bandwidth of shared memory (B) is 1,024 bits or 512 bit per clock cycle. Therefore, the total CTA swapping latency can be computed using a parametrized model as follows:

$$Cycles = \frac{(N + log_2 W) + 96 + (160 \times D \times W)}{B} \quad (2)$$

D. Resolving Memory Contention

One side effect of increasing TLP is the increase in MLP. Typically the number of memory instructions issued per unit time also increase as more warps are concurrently executed. Note that the total number of memory requests over the kernel execution time does not change. It is the rate of memory requests that changes. The increase in MLP can be beneficial or detrimental, depending on the memory system capability, such as available bandwidth. Previous studies [5], [28] have shown that increasing MLP may hurt performance if they increase memory system contention. To alleviate any potential contention, we propose memory request reordering technique built on previous studies [29], [30].

Mascar [29] technique places a reordering queue between the load/store unit and L1 cache. Once a load instruction is issued, it checks the cache tags to find the hit/miss status. If the load is a hit, data is provided. If the load is a miss, then it may need additional resources to handle the miss such as Miss Status Handling Registers (MSHRs). If the miss cannot be properly handled due to structural hazards, or failure to reserve a cache entry for replacement (reservation failures), then that load instruction is moved to a re-execution queue for later replay, instead of stalling the entire memory system.

Inspired by this prior work [29], we create two load queues: one queue to manage loads from active CTAs and another queue to manage loads from inactive CTAs that are swapped out after the loads were issued. Every load enters the active queue first and if corresponding CTA is swapped out then that load is moved to inactive queue. The loads from the active queue are always given higher priority than the loads from the inactive queue. By giving higher priority to the active queue, any load requests from a swapped-out CTA are handled only after servicing active CTA requests. If no new memory requests are generated from the active CTAs, the load requests from the swapped-out CTAs are handled. Therefore, the VT architecture is able to detect load requests from swapped out CTAs and gives them lower priority to resolve memory contention.

Based on the previous study [29], each entry for the load queue has 301 bits to store information. In VT, each entry is 309 bits because of extra VWI (8bits). In our evaluation, total 60 (30 for active queue and 30 for inactive queue) entries are used, therefore the total size of queues is approximately 2,317 bytes. The space overhead of queues is about 3.5% compared to the L1 cache/shared memory which is about 64 KB [7], [9].

Table I
GPU MICROARCHITECTURAL PARAMETERS

Parameters	Value
Number of SMs	15
Core Clock	1.4 GHz
SIMD Pipeline Width	16
Warp Size	32-threads
Max Number of CTAs / Core	8
Max Number of Threads / Core	1,536
Max Number of Warps / Core	48
Number of Warp Schedulers / Core	2
Number of Scoreboard Counters / Core	256
Number of Registers	32,768
Size of Active & Inactive Queue	30 & 30
L1 Cache Size / Core	16 KB
Shared Memory / Core	48 KB
Shared Memory Bandwidth / Cycle	512 bits
MSHRs/Core	64
Warp Scheduler Policy	LRR & GTO & TW
CTA Scheduler Policy	Round Robin

Table II
LIST OF BENCHMARK APPLICATIONS

Type-S Applications		Type-C Applications	
Application Name	**Abbr.**	**Application Name**	**Abbr.**
2D Convolution [45]	2D	BlackScholes [46]	BS
Back Propagation [47]	BP	B+ Tree [47]	BT
Breadth First Search [48]	BF	Heart wall [47]	HW
Coulombic Potential [5]	CP	EigenValues [46]	EV
Convolution Separable [46]	CS	Binomial Options [46]	BO
Convolution Texture [46]	CT	HotSpot [47]	HS
Discrete Cosine Transform [46]	DC	LavaMD [47]	LM
Discrete Haar Wavelet Decomposition [46]	DW	Single Asian OptionP [46]	SP
2D finite different time domain [45]	FD	Matrix Multiplication [46]	MM
Fast Fourier Transform [48]	FT	Speckle Reducing Anisotropic Diffusion [47]	SRAD
Gaussian Elimination [47]	GA	Neural Network Digit Recognition [5]	NN
Histogram [46]	HI	MUMmerGPU [47]	MUM
Needleman Wunsch [47]	ND	QuasiRandom Generator [46]	QRG
Path Finder [47]	PF	Ray Tracing [5]	RAY
Reduction [46]	RD	Leukocyte [47]	LE
Sorting Networks [46]	SN	Merge Sort [46]	MS
Thread Fence Reduction [46]	TF	Symmetric Rank-2K Operations [45]	S2K
Vector Addition [46]	VA		

V. EVALUATION

A. Methodology

Our baseline configuration is similar to an NVIDIA Fermi-like GPU architecture. The baseline architecture is able to execute a maximum of 8 CTAs and 1,536 threads concurrently, and each SM has 32K $\times$ 32-bit registers with 64 KB reconfigurable on-chip memory [31]. We configure the size of shared memory to be 48 KB. The detailed microarchitecture configuration that we simulated is presented in Table I. We have chosen 18 applications from NVIDIA CUDA SDK [46], Rodinia benchmark suite [47], GPGPU-Sim [5], scalable heterogeneous computing benchmark suite [48], and polybench [45] that are Type-S applications which are bottlenecked by the scheduling limit. Clearly, Type-C applications show no better or worse performance compared to the baseline. They have reached the capacity limit and no

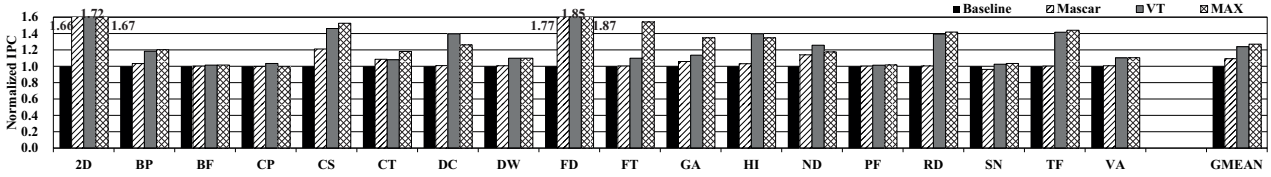


Figure 8. Performance Improvement of VT Architecture

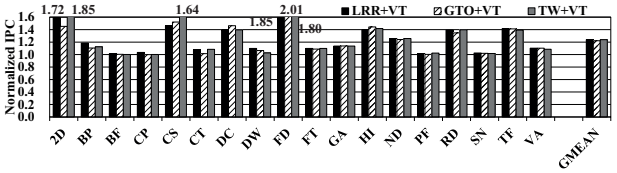


Figure 9. Impact of Various Warp Schedulers

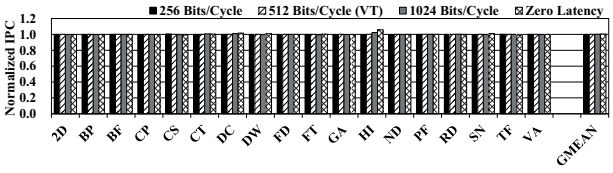


Figure 10. Impact of Various Swapping Delays

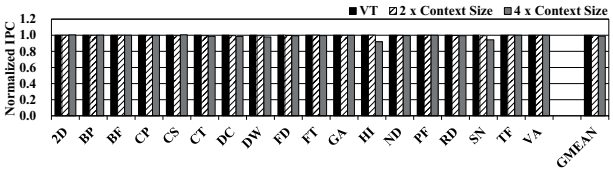


Figure 11. Impact of Various Context Sizes

more virtual CTAs are created, hence SMs will schedule the same number of threads to the baseline. In this case, the VT architecture does not improve the performance for the Type-C applications since CTA swapping will not happen. Therefore, there is no need to apply VT when applications are not bottlenecked by the scheduling limit.

The applications from Type-S and Type-C are listed in Table II. All of our applications run from beginning to 1 billion instructions using GPGPU-sim [5]. We also use the largest available input set for each benchmark since some benchmarks dispatch only a small number of CTAs with the small input set which results in an artificially high resource underutilization. For the evaluation, we use Loose Round Robin (LRR) warp scheduler since the LRR warp scheduler is commonly implemented on GPUs [32], [49], [50]. However, to verify the impact of various warp schedulers with VT, we also evaluate VT using Greedy-Then-Oldest (GTO) [51] and TWo-level (TW) [32] warp schedulers and compare the results to GTO and TW warp schedulers, respectively.

B. Performance of VT Architecture

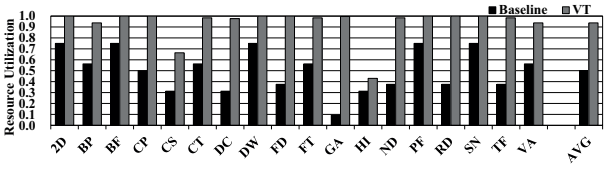
Figure 8 shows the normalized IPC of four different GPU configurations: baseline (Baseline), Mascar [29], VT architecture, and unlimited scheduling resources (as many program counters as needed to fill the capacity, as many SIMT stacks as needed etc.) with Mascar (MAX). Our results show that an average performance improvement for VT is 23.9% compared to baseline. The performance improves universally across almost all the benchmarks. However, *BF*, *PF*, and *SN* will see no benefits from VT. *SN* and *PF* do not suffer from any stalls to begin with and even in the baseline instructions are issued for more than 80% of total execution cycles as shown in Figure 4. In addition, *BF* has low ratio of memory related stalls. Therefore, VT can only hide a small portion of stall cycles in these applications resulting in small performance improvement.

We compare VT with Mascar which is a recently proposed memory reordering technique [29] since VT has the memory reorder scheme which is inspired from previous studies [29], [30]. Based on our evaluation, Mascar improves an average 9.3% performance while VT improves an average 23.9% compared to the baseline. With Mascar, applications *2D*, *CS*, *FD*, and *ND* show high performance improvement compared to the baseline. These applications suffer from relatively high pipeline stalls and Mascar is well suited for reducing the pipeline stalls. With VT, the performance improves universally across almost all the benchmarks because VT is designed to increase TLP to hide memory related stalls such as long latency stalls and pipeline stalls. Overall, VT outperforms Mascar by 13.4%.

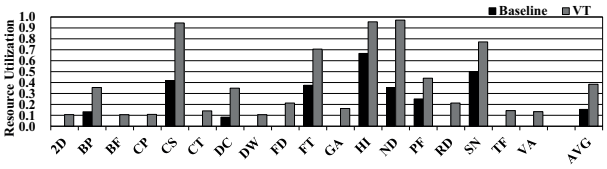
We also compare VT with a hardware-intensive traditional approach, shown as MAX in our evaluations. MAX increases the scheduling limit to fill up all the available registers or shared memory (whichever comes first) assuming scheduling hardware can be scaled at zero cost. Hence, MAX assumes no latency penalties for supporting larger scoreboards, bigger instruction buffers, and larger comparison logic to pick the ready warps. MAX also includes Mascar on top of unlimited scheduling resources. Based on the results, the performance of VT is nearly equal to the performance of MAX. On average, VT improves the performance by about 23.9% and MAX improves the performance about 26.9%. In *2D*, *CP*, *DC*, *HI*, and *ND*, VT yields higher performance than MAX. The reason behind this is due to the small changes in warp scheduling in VT due to the memory request prioritization.

We also measured the detailed cycle breakdown with VT. The second bar in each group in Figure 4 shows the cycle distributions for VT. The stall cycle distribution shows that VT reduces 18.1% of stall cycles compared to baseline. Especially, the pipeline stalls and long latency stalls are significantly reduced. VT exploits higher TLP to hide the long latency stalls more effectively.

1) Impact of Warp Scheduler: In this section, we evaluate the performance impact of VT using various warp schedulers: LRR [32], [49], [50], GTO [51], and TW warp schedulers [32]. Figure 9 shows the performance improvement of VT using various warp schedulers. As mentioned in the previous section, using the LRR warp scheduler, VT improves the performance by about 23.9% compared to the baseline LRR configuration. The performance of GTO+VT

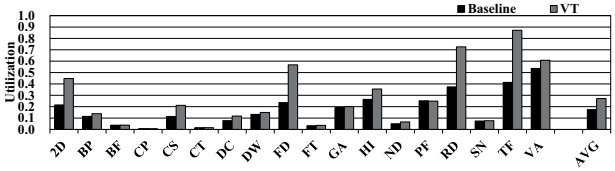


(a) Register File Utilization

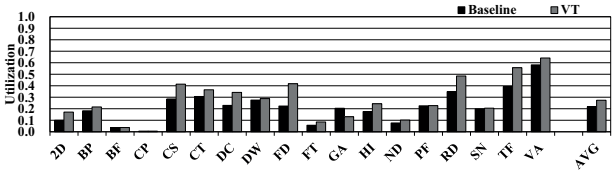


(b) Shared Memory Utilization

Figure 12. Resource Utilization with VT



(a) L1 MSHR Utilization



(b) DRAM Bandwidth Utilization

Figure 13. Impact on Memory System

is about 22.2% compared to the baseline GTO configuration. In addition, TW+VT improves the performance by about 23.9% compared to the baseline TW configuration. As can be seen from the results, VT can be combined with any warp scheduling policies and it provides fairly consistent performance improvements.

2) Impact of Swapping Delay and Context Size: We evaluate the performance impact of VT using various swapping delays. In the prior section, we assumed that the context can be swapped to shared memory using 512-bits/cycle bandwidth. In this section, we evaluate VT using 256 bits, 512 bits, and 1,024 bits of shared memory bandwidth. In addition, we have evaluated VT using no swapping overhead. Figure 10 shows the evaluation results. Based on the results, the performance gaps are very small. VT without the swapping overhead is 0.4% faster than the baseline that uses 512-bits/cycle bandwidth. Also, VT with 256 bits of shared memory bandwidth shows no performance degradation. The only time swapping overhead has an impact on performance is when there are no CTAs in active state or an occasional conflict in using the shared memory to save/restore CTA context and a demand request.

We also evaluate the performance impact of the size of the CTA context data that is needed to be saved and restored with VT. We increase the context size to be twice as large and four times as large. Large context is typically due to increasing the number of entries in the SIMT stack. Figure 11 shows the performance results. As the context size doubles, VT performance is only degraded by about 0.2% on average. VT performance improvements are reduced by 1.1% on average when context size is quadrupled. The performance degradation of increasing the context size is not due to increased latency to save/restore context state. Rather it is due to the increased demand for more shared memory where the context is saved. As the size of CTA contexts is increased, the number of CTAs that can be dispatched to SMs is decreased for some applications such as *HI* and *SN* as they hit the capacity limit quicker.

C. Resource Utilization

In this section, we present the resource utilization for the register file and shared memory with VT. Figure 12 shows the resource utilization graph for the register file and the shared memory. Figure 12(a) shows the register file utilization. In Baseline, register file resource utilization is around 50.2%, however VT increases the register utilization to 93.8%. The register file utilization may not reach 100% since the unused register file is too small to assign another CTA. Also, for applications such as *CS* and *HI* more CTAs cannot be assigned as the shared memory capacity limit is reached.

Figure 12(b) shows the result of shared memory utilization on GPUs. On average, shared memory space utilization is about 15.4% in the baseline. However, with VT, an average utilization is increased to 38.5%. Note that most applications reach the register file capacity limit well before reaching the shared memory limit.

D. Impact on Memory System

In this section, we show the impact on memory system when VT is activated. We measured the L1 cache miss rate and L1 MSHR utilization. Our results on L1 cache miss rates showed that VT increases cache miss rates by less than 1.7%, which has negligible performance impact. Most applications only rarely reuse cache lines [52], [53] temporally. When a CTA is swapped out only a small fraction of its data in L1 cache is reused when the CTA is swapped back in. As a result, VT does not degrade cache miss rate. L1 MSHR utilization is increased with VT. Note that even though the overall cache miss rate is not decreased, MLP is increased because more memory requests are initiated within a shorter time interval with VT. However, VT does not generate any useless data requests since most of the data brought into the cache is used before the CTA is swapped out. As a result, Figure 13(a) shows that MSHR utilization is increased from 17.5% to 27.1% on average. However, the performance impact of the increased utilization is negligible.

In addition, we measure DRAM bandwidth utilization. Figure 13(b) shows the bandwidth utilization on Baseline and VT. On average, the DRAM utilization is increased from 21.7% to 27.4% when VT is activated. In *GA*, the utilization is decreased because L2 cache miss rate is slightly decreased. The reason behind the lower miss rate is due to the improved inter-CTA data locality. Overall, the DRAM utilization increase is also for the same reason as MSHR utilization increase. Namely, MLP is improved with VT as additional CTAs generate more requests.

VI. Related Work

To the best of our knowledge, this is the first paper that proposes a simple context switching solution to improve TLP using underutilized register file and shared memory on GPUs. In this section, we describe the closely related work.

Latency Tolerance Techniques on CPUs: Previously, many studies proposed various latency tolerance techniques on CPUs [54], [55], [56]. In these techniques, instructions which depend on a long latency memory operation are removed from the instruction window to allow executing near future instructions. By doing this, instruction level parallelism is successfully increased and the memory latency can be well tolerated. In this paper, we propose a light weight context switching technique which increases TLP to hide the long latency memory stalls more effectively on GPUs.

Application Level Context Switching on GPUs: Application level context switching in current GPUs is analogous to process scheduling in CPUs involving full context swap [8]. According to Fermi whitepaper this switch takes up to 25us, which corresponds to tens of thousands of cycles. However, VT targets to improve scheduling efficiency for applications, and VT does not perform full context swap and switches only a limited CTA context information which takes only tens of cycles.

Resource Underutilization Problem on GPUs: Several power management policies have been proposed for reducing leakage power consumption for unused on-chip resources [10], [57], [58], [59]. For instance, Abdel-Majeed and Annavaram [10] proposed the warped register file. In the proposed technique, the tri-modal register file is inserted into GPUs. The tri-modal register file can change the register state into ON, OFF, and drowsy states. The registers are power-gated if the registers are not needed to execute the kernels (OFF state). In addition, the state of registers is changed to ON state when the registers are being used by the processors, otherwise the state of registers is in drowsy state. In contrast to the technique, VT targets to increase performance by minimizing underutilized resources and improving resource utilization.

In addition, flexible resource management schemes have been proposed to overcome the capacity limit [37], [60], [61]. In this approach, all on-chip storages in an SM, especially register file and shared memory are unified into a single malleable memory. This approach alleviates capacity limit by more flexibly managing storage demands of various applications. However, as the capacity limit is relaxed, more applications will be constrained by the scheduling limit. Therefore, VT complements potential TLP bottleneck with this approach by relaxing the scheduling limit.

Programming and compile time approaches have also been proposed for improving resource utilization [62], [63]. For instance, in shared memory multiplexing [63], allocated shared memory is released as soon as the space is no longer used by CTAs, even though the CTAs are not yet completed. These approaches better utilize on-chip storage space by activating more threads, however in common such techniques still capped to both thread and CTA limits. VT allows to issue more threads even beyond the scheduling limit, therefore further improves TLP and enforces capacity limit to be the only TLP limiting factor.

Thread Scheduling on GPUs: Within the scheduling limit, a solution for squeezing more TLP is relaxing the restriction of CTA-granularity resource allocation. Xiang *et al.* proposed the warp level resource management technique [11] called WarpMan. In WarpMan, the CTA scheduler allocates the threads on SMs at a warp granularity instead of a CTA granularity. By doing that, the proposed technique can exploit higher degree of TLP using additionally issued threads. The number of thread that can be dispatched on SMs cannot exceed the conventional hardware limit, however VT can dispatch more threads on SMs.

GPUs exploit TLP for hiding long processing delay, which is the primary objective of warp schedulers [26], [32], [64]. Narasiman *et al.* proposed the two-level warp scheduler for maximizing latency hiding effect [32]. The scheduler divides the warps into multiple fetch groups and the long latency operation stalls of each group can be hidden by executing warps from other fetch groups. All these techniques improve performance by rearranging the order of execution within the scheduling limit. Our approach provides more warps and CTAs to be scheduled therefore improves latency hiding effect as can be seen in the previous section.

Some previous studies also have noted that issuing more CTAs is not always beneficial for GPGPU applications [5], [28], [65]. Issuing additional threads can increase memory contention on GPUs, therefore the performance may degrade for the applications. Researchers have explored minimizing memory contention by improving CTA or warp scheduling policy [28], [50], [51]. Kayiran *et al.*, or proposed a dynamic CTA scheduling mechanism [28], which throttles the number of active CTAs based on the degree of memory contention. The OWL scheduler [50] further improves performance by reordering CTAs for maximizing bank level parallelism and row-buffer locality of DRAM. Rogers *et al.* proposed Cache-Conscious Wavefront Scheduling [51], which issues warp instructions based on the intra-wavefront locality, therefore the scheduler reduces the operation stalls by increasing cache efficiency. In this work we show that increasing TLP is still beneficial for many GPGPU applications in terms of the performance, along with memory requests reordering.

Memory Request Scheduling on GPUs: To maximize the benefits of additional TLP and minimize its side effects, we have proposed the memory enhancement technique on VT which is inspired by recently proposed memory reordering approaches [29], [30]. As we discussed previously, the design of VT adopts the concept of the re-execution queue from Mascar [29]. The queue resolves blocking problem of cache requests, and VT applies prioritization techniques for further improving its effect in the context of VT. MRPB [30] prioritizes requests using a similar scheduling buffer, and more focuses on the reordering policy of cache access requests to be cache friendly. Our VT architecture incorporates memory reordering approaches after providing extra CTAs to be scheduled, thereby achieves even better performance compared to applying only a single technique.

VII. Conclusion

In this paper, we observe that the scheduling limit is a major TLP bottleneck on GPUs. To relax the scheduling limit constraint, we propose the VT architecture which can dispatch CTAs to SMs until the on-chip storage capacity limit is reached regardless of the scheduler limit. We use the notion of active and inactive CTAs where the number of active CTAs is still limited by the scheduling logic. VT swaps out a CTA when warps in the active CTA are all stalled by the long latency memory operations. By switching

between active and inactive states, VT can exploit higher degree of TLP without increasing logic complexity. Based on our evaluation, VT improves performance by 23.9% on average.

ACKNOWLEDGMENT

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP)(No. NRF-2015R1A2A2A01008281), by Memory Division, Samsung Electronics Co., Ltd., and by the following grants: DARPA-PERFECT-HR0011-12-2-0020 and NSF-CAREER-0954211, NSF-0834798. W. W. Ro is the corresponding author.

REFERENCES

[1] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke, "APOGEE: Adaptive Prefetching on GPUs for Energy Efficiency," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, 2013.

[2] Y. Zhang and J. Owens, "A Quantitative Performance Analysis Model for GPU Architectures," in *Proceedings of IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, 2011.

[3] C. Wojek, G. Dorkó, A. Schulz, and B. Schiele, "Sliding-windows for Rapid Object Class Localization: A Parallel Technique," in *Pattern Recognition*, Springer, 2008.

[4] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU Concurrency with Elastic Kernels," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, 2013.

[5] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads using a Detailed GPU Simulator," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, ISPASS '09, 2009.

[6] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: A Simulation Framework for CPU-GPU Computing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, 2012.

[7] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU Architecture," *IEEE Micro*, vol. 31, no. 2, 2011.

[8] "Fermi: NVIDIA's Next Generation CUDA Compute Architecture." http://goo.gl/zAtZMY", 2009. Accessed: 24 April 2014.

[9] "NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110." http://goo.gl/DLi9nu. Accessed: 23 Dec 2014.

[10] M. Abdel-Majeed and M. Annavaram, "Warped Register File: A Power Efficient Register File for GPGPUs," in *Proceedings of the 19th International Symposium on High Performance Computer Architecture*, HPCA '13, 2013.

[11] P. Xiang, Y. Yang, and H. Zhou, "Warp-level Divergence in GPUs: Characterization, Impact, and Mitigation," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, HPCA '14, 2014.

[12] H. Jeon and M. Annavaram, "GPGPU Register File Management by Hardware Co-operated Register Reallocation," tech. rep., 2014.

[13] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "GPU Register File Virtualization ," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '15, 2015.

[14] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu, "A Case for Core-assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, 2015.

[15] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram, "Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming," in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016.

[16] C. Li, Y. Yang, H. Dai, S. Yan, F. Mueller, and H. Zhou, "Understanding the Tradeoffs between Software-managed vs. Hardware-managed Caches in GPUs," in *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '14, 2014.

[17] M. Rhu and M. Erez, "The Dual-path Execution Model for Efficient GPU Control Flow," in *Proceedings of IEEE 19th International Symposium on High Performance Computer Architecture*, HPCA '13, 2013.

[18] W. W. L. Fung, "GPU Computing Architecture for Irregular Parallelism," 2015.

[19] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram, "Warped-Preexecution: A GPU Pre-execution Approach for Improving Latency Hiding," in *Proceedings of the 22th International Symposium on High Performance Computer Architecture*, HPCA '16, 2016.

[20] R. Balasubramanian, V. Gangadhar, Z. Guo, C.-H. Ho, C. Joseph, J. Menon, M. P. Drumond, R. Paul, S. Prasad, P. Valathol, and K. Sankaralingam, "Enabling GPGPU Low-Level Hardware Explorations with MIAOW: An Open-Source RTL Implementation of a GPGPU," *ACM Trans. Archit. Code Optim.*, vol. 12, June 2015.

[21] S. Collange, "Stack-less SIMT Reconvergence at Low Cost," tech. rep., Technical Report HAL-00622654, INRIA, 2011.

[22] NVIDIA, "Whitepaper: NVIDIA GeForce GTX 680." http://goo.gl/f58MQ, 2012.

[23] N. Brunie, S. Collange, and G. Diamos, "Simultaneous Branch and Warp Interweaving for Sustained GPU Performance," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, 2012.

[24] B. Coon, P. Mills, S. Oberman, and M. Siu, "Tracking Register Usage during Multithreaded Processing using a Scoreboard having separate memory regions and storing sequential register size indicators," 2008. US Patent 7,434,032.

[25] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *Micro, IEEE*, vol. 28, no. 2, 2008.

[26] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "A Hierarchical Thread Scheduler and Register File for Energy-Efficient Throughput Processors," *ACM Trans. Comput. Syst.*, vol. 30, Apr. 2012.

[27] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling Preemptive Multiprogramming on GPUs," in *Proceedings of the 41st International Symposium on Computer Architecture*, ISCA '14, 2014.

[28] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, 2013.

[29] A. Sethia, D. Jamshidi, and S. Mahlke, "Mascar: Speeding up GPU Warps by Reducing Memory Pitstops," in *Proceedings of the 21st International Symposium on High Performance Computer Architecture*, HPCA '15, 2015.

[30] W. Jia, K. Shaw, and M. Martonosi, "MRPB: Memory Request Prioritization for Massively Parallel Processors," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, HPCA '14, 2014.

[31] C. Nvidia, "CUDA C Programming Guide," *NVIDIA Corporation*, 2014.

[32] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU Performance via Large Warps and Two-level Warp Scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '11, 2011.

[33] N. Compute, "PTX: Parallel Thread Execution ISA Ver-

sion 2.3," *Dostopno na: http://developer. download. nvidia. com/compute/cuda/3*, vol. 1, 2010.

[34] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., 1st ed., 2010.

[35] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, May 2008.

[36] S. Stone, J. Haldar, S. Tsao, W. m.W. Hwu, B. Sutton, and Z.-P. Liang, "Accelerating Advanced MRI Reconstructions on GPUs," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, 2008.

[37] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, "Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor," in *Proceedings of the 45th Annual International Symposium on Microarchitecture*, MICRO '12, 2012.

[38] C.-C. Wu, J.-Y. Ke, H. Lin, and W. chun Feng, "Optimizing Dynamic Programming on Graphics Processing Units via Adaptive Thread-Level Parallelism," in *Proceedings of IEEE 17th International Conference on Parallel and Distributed Systems*, ICPADS '11, 2011.

[39] P. Xiang, Y. Yang, M. Mantor, N. Rubin, and H. Zhou, "Many-thread Aware Instruction-level Parallelism: Architecting Shader Cores for GPU Computing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, 2012.

[40] F. Ji and X. Ma, "Using Shared Memory to Accelerate MapReduce on Graphics Processing Units," in *Proceedings of the 2011 IEEE International Parallel Distributed Processing Symposium*, IPDPS '11, 2011.

[41] P. Micikevicius, "3D Finite Difference Computation on GPUs Using CUDA," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU '09, 2009.

[42] W. Fung and T. Aamodt, "Thread Block Compaction for Efficient SIMT Control Flow," in *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, HPCA '11, 2011.

[43] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware," *ACM Trans. Archit. Code Optim.*, vol. 6, July 2009.

[44] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, no. 2, 2008.

[45] "PolyBench: The Polyhedral Benchmark Suite." http://web.cse.ohio-state.edu/~pouchet/software/polybench/. Accessed: 30 April 2015.

[46] "NVIDIA CUDA SDK Code Samples." http://developer.nvidia.com/cuda-downloads. Accessed: 24 April 2014.

[47] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC '09, 2009.

[48] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The Scalable Heterogeneous Computing (SHOC) Benchmark Suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, 2010.

[49] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated Scheduling and Prefetching for GPGPUs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, 2013.

[50] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, 2013.

[51] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, 2012.

[52] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal, "A Detailed GPU Cache Model based on Reuse Distance Theory," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, HPCA '14, 2014.

[53] Y. Oh, K. Kim, M. K. Yoon, J. H. Park, Y. Park, W. W. Ro, and M. Annavaram, "APRES: Improving Cache Efficiency by Exploiting Load Characteristics on GPUs," in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016.

[54] A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, "A Large, Fast Instruction Window for Tolerating Cache Misses," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, 2002.

[55] M. Pericas, A. Cristal, R. Gonzalez, D. Jimenez, and M. Valero, "A Decoupled KILO-instruction Processor," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, HPCA '06, 2006.

[56] Y. Kora, K. Yamaguchi, and H. Ando, "MLP-aware Dynamic Instruction Window Resizing for Adaptively Exploiting Both ILP and MLP," in *Proceedings of the 46th International Symposium on Microarchitecture*, MICRO '13, 2013.

[57] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, 2013.

[58] M. Abdel-Majeed, D. Wong, and M. Annavaram, "Warped Gates: Gating Aware Scheduling and Power Gating for GPGPUs," in *Proceedings of the 46th International Symposium on Microarchitecture*, MICRO '13, 2013.

[59] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-Compression: Enabling power efficient GPUs through register compression," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, 2015.

[60] A. B. Hayes and E. Z. Zhang, "Unified On-chip Memory Allocation for SIMT Architecture," in *Proceedings of the 28th International Conference on Supercomputing*, ICS '14, 2014.

[61] O. Villa, D. R. Johnson, M. O'Connor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, S. W. Keckler, and W. J. Dally, "Scaling the Power Wall: A Path to Exascale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, 2014.

[62] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, 2008.

[63] Y. Yang, P. Xiang, M. Mantor, N. Rubin, and H. Zhou, "Shared Memory Multiplexing: A Novel Way to Improve GPGPU Throughput," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, 2012.

[64] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, 2011.

[65] X. Chen, L.-W. Chang, C. Rodrigues, J. Lv, Z. Wang, and W. mei Hwu, "Adaptive Cache Management for Energy-Efficient GPU Computing," in *Proceedings of the 47th International Symposium on Microarchitecture*, MICRO '14, 2014.

Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware

WILSON W. L. FUNG, IVAN SHAM, GEORGE YUAN, and TOR M. AAMODT
University of British Columbia

7

Recent advances in graphics processing units (GPUs) have resulted in massively parallel hardware that is easily programmable and widely available in today's desktop and notebook computer systems. GPUs typically use single-instruction, multiple-data (SIMD) pipelines to achieve high performance with minimal overhead for control hardware. Scalar threads running the same computing kernel are grouped together into SIMD batches, sometimes referred to as warps. While SIMD is ideally suited for simple programs, recent GPUs include control flow instructions in the GPU instruction set architecture and programs using these instructions may experience reduced performance due to the way branch execution is supported in hardware. One solution is to add a stack to allow different SIMD processing elements to execute distinct program paths after a branch instruction. The occurrence of diverging branch outcomes for different processing elements significantly degrades performance using this approach. In this article, we propose dynamic warp formation and scheduling, a mechanism for more efficient SIMD branch execution on GPUs. It dynamically regroups threads into new warps on the fly following the occurrence of diverging branch outcomes. We show that a realistic hardware implementation of this mechanism improves performance by 13%, on average, with 256 threads per core, 24% with 512 threads, and 47% with 768 threads for an estimated area increase of 8%.

This work was supported by the Natural Sciences and Engineering Research Council of Canada.
An earlier version of this article appeared in the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'07) [Fung et al. 2007]. The new material in this article consists of: (1) a reevaluation of our proposed mechanism with a baseline configuration providing a better match to existing GPUs to provide a more accurate performance-area analysis; (2) a more detailed description of an area-efficient implementation of our best performing warp scheduling policy (Majority); (3) an extended analysis of the sources of performance loss of our Majority scheduling policy; (4) a data cache bank conflict model is now incorporated into our baseline configuration; (5) sensitivity analysis of our proposed mechanism against various SIMD warp sizes and thread pool sizes; (6) a proposal to further reduce the area consumed by the warp pool while maintaining the ability to handle excessively diverging code via spilling to memory; (7) a discussion on the limitations of dynamic warp formation.
Author's address: Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, B.C., V6T 1Z4, Canada; email: {wwlfung, aamodt}@ece.ubc.ca.

DOI 10.1145/1543753.1543756 http://doi.acm.org/10.1145/1543753.1543756

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—Single-instruction-stream, multiple-data-stream processors (SIMD)

General Terms: Design, Performance

Additional Key Words and Phrases: SIMD, fine-grained multithreading, control flow, GPU

ACM Reference Format:

Fung, W. W. L., Sham, I., Yuan, G., and Aamodt, T. M. 2009. Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. ACM Trans. Architec. Code Optim. 6, 2, Article 7 (June 2009), 37 pages.
DOI = 10.1145/1543753.1543756 http://doi.acm.org/10.1145/1543753.1543756.

1. INTRODUCTION

As semiconductor process technology advances continue and transistor density increases, computation potential continues to grow. However, finding effective ways to leverage process technology scaling for improving the performance of real-world applications has become increasingly challenging. To improve performance, hardware must exploit parallelism. Until recently, the dominant approach has been to extract more instruction level parallelism from a single thread through increasingly complex scheduling logic, larger caches, and sophisticated prediction mechanisms. As diminishing returns to ILP scaling begin to limit performance of single-threaded applications [Agarwal et al. 2000], attention has shifted toward employing additional resources to increase throughput by exploiting explicit thread-level parallelism in software (forcing software developers to share the responsibility for improving performance).

The modern graphics processing unit (GPU) can be viewed as an example of the latter approach [Purcell et al. 2002; Buck et al. 2004]. Earlier generations of GPUs consisted of fixed function 3D-rendering pipelines. New real-time rendering techniques required new hardware, which impeded the adoption of new graphics algorithms and thus motivated the introduction of programmability [Lindholm et al. 2001], long available in traditional offline computer animation [Upstill 1990], into GPU hardware for real-time computer graphics. In modern GPUs, much of the formerly hardwired pipeline is replaced with programmable hardware processors that run a relatively small program, called a shader, on each input vertex or pixel [Lindholm et al. 2001]. Shaders are either written by the application developer or substituted by the graphics driver to implement traditional fixed-function graphics pipeline operations. The compute model provided by modern graphics processors for running nongraphics workloads is closely related to stream processing [Rixner et al. 1998; Dally et al. 2003].

The programmability of shader processors has greatly improved over the past several years, and the shaders of the latest generation GPUs are turing-complete, opening up exciting new opportunities to speedup “general purpose” (i.e., nongraphics) applications. Based on experience gained from pioneering efforts to generalize the usage of GPU hardware [Purcell et al. 2002; Buck et al. 2004], GPU vendors have introduced new programming models and

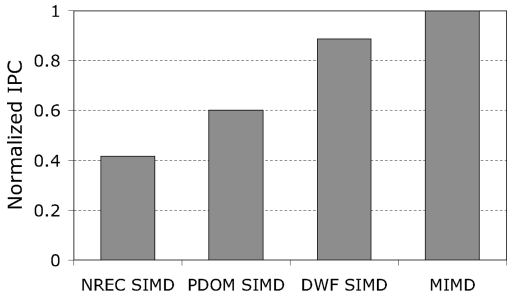


Fig. 1. Performance loss due to branch divergence when executing scalar SPMD threads using SIMD hardware. NREC is a naïve branch handling mechanism; PDOM is our baseline representation of modern GPU branch handling capability; DWF is our proposed approach; MIMD represents an upper bound.[2]

associated hardware support to further broaden the class of applications that may efficiently use GPU hardware [AMD, Inc. 2006; NVIDIA Corp. 2007a].

Even with a general-purpose programming interface, mapping existing applications to the parallel architecture of a GPU is a nontrivial task. Although some applications can achieve speedups of up to 431 times over a modern CPU [Hwu et al. 2007], other applications show less improvement when mapped to a GPU [Buatois et al. 2008]. One major challenge for contemporary GPU architectures is efficiently handling control flow in shaders [Shebanow 2007]. The reason is that, in an effort to improve computation density, modern GPUs typically batch together groups of individual threads running the same shader, and execute them together in lock step on a single-instruction, multiple-data (SIMD) pipeline [Lorie and Strong 1984; Montrym and Moreton 2005; Moy and Lindholm 2005; Luebke and Humphreys 2007; NVIDIA Corp. 2007a]. Such thread batches are referred to as warps[1] by NVIDIA [Lindholm et al. 2008; NVIDIA Corp. 2007a; Shebanow 2007]. This approach has worked well [Levinthal and Porter 1984] for graphics operations such as shadow volume generation [Crow 1977] and bump mapping [Blinn 1978], which historically have not required branch instructions. However, when shaders do include branches, the execution of different threads grouped into a warp to run on the SIMD pipeline may no longer be uniform across SIMD elements. This causes a hazard in the SIMD pipeline [Moy and Lindholm 2005; Woop et al. 2005] known as branch divergence [Lorie and Strong 1984; Shebanow 2007]. We found that naïve handling of *branch divergence* incurs a significant performance penalty on the GPU for control-flow intensive applications relative to an ideal multiple-instruction, multiple-data (MIMD) architecture with the same peak IPC capability (see Figure 1).

This article makes the following contributions:

—It establishes that, for the set of nongraphics applications we studied, reconverging control flow of processing elements at the immediate postdominator

[1]The term "warp" originates from weaving, the first parallel-thread technology in the textile industry [Lindholm et al. 2008]. In its original context, the term "warp" refers to "the threads stretched lengthwise in a loom to be crossed by the weft" [Fowler et al. 1995].

of the divergent branch is nearly optimal with respect to oracle information about the future control flow of each individual processing element for our baseline intrawarp branch reconvergence mechanism.

—It quantifies the performance gap between the immediate postdominator branch reconvergence mechanism and the performance that would be obtained on a MIMD architecture with support for the same peak number of operations per cycle. This performance gap highlights the importance of finding better branch handling mechanisms.

—It proposes and evaluates a novel hardware mechanism, dynamic warp formation, for regrouping processing elements of individual SIMD warps in hardware on a cycle-by-cycle basis to improve the efficiency of branch handling.

—It highlights quantitatively that warp scheduling policy (the order in which the warps are issued from the scheduler) is an integral part to both the performance and area overhead of dynamic warp formation, and proposes an area efficient implementation of a well-performing scheduling policy.

In particular, for a set of data parallel, nongraphics applications ported to our modern GPU-like multithreaded SIMD architecture, we find the speedup obtained by reconverging the diverging threads within a SIMD warp at the immediate postdominator of the diverging branch obtains a speedup of 45% over not reconverging. Dynamically regrouping scalar threads into SIMD warps on a cycle-by-cycle basis obtains an additional speedup of 13% with 256 threads per shader core. This additional speedup increases to 22%, 24%, 37%, and 47% with 384, 512, 640, and 768 threads per shader core, respectively (the 47% speedup with 768 threads per shader core corresponds to 114% speedup versus not reconverging). Using CACTI 4.2, we estimate the hardware required by this regrouping mechanism would add 8% to the total chip area if it were implemented on a contemporary GPU, such as the NVIDIA Geforce 8800GTX (470mm^2) [Lindholm et al. 2008].

The rest of this article is organized as follows: Section 2 gives an overview of the baseline architecture used in this article. Section 3 describes the immediate postdominator control-flow reconvergence mechanism and evaluates the potential of intrawarp stack-based reconvergence mechanisms with a limit study. Section 4 describes our proposed dynamic warp formation mechanism. Section 5 describes the simulation methodology of the proposed GPU microarchitecture. Section 6 describes our experimental results. Section 7 estimates the area overhead to implement our proposed mechanism. Section 8 discusses some limitations of dynamic warp formation. Section 9 describes related work. Section 10 summarizes this article and suggests future work.

2. BASELINE ARCHITECTURE

This section describes the compute model and the baseline SIMD GPU microarchitecture used for the rest of this article.

[2]*NREC SIMD* and *PDOM SIMD* are described in Section 3, while *DWF SIMD* is described in Section 4. Benchmarks and microarchitecture used here are described in Section 5.

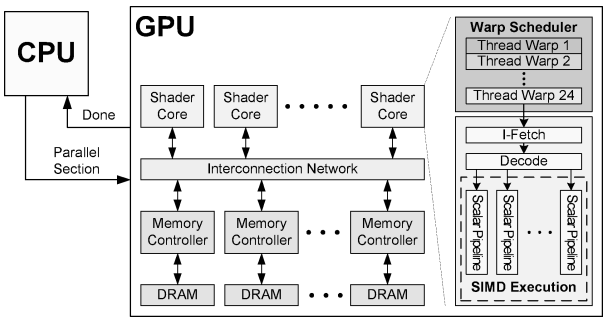


Fig. 2. Baseline GPU microarchitecture. Blocks labeled "scalar pipeline" include register read, execute, memory, and writeback stages.

2.1 Compute Model

In this article, we have adopted a compute model that is similar to NVIDIA's CUDA programming model [NVIDIA Corp. 2007a]. In this compute model, the application starts off as a single thread running on the CPU. At some point during execution, the CPU reaches a kernel call and spawns a parallel section to the GPU to exploit data-level parallelism. At this point, the CPU will then stop its execution and wait for the GPU to finish the parallel section.[3] This sequence can repeat multiple times until the program completes.

Each parallel section consists of a collection of threads executing the same code, which we call a thread program. Similar to many thread programming APIs, a thread program is encapsulated as a function call. In our implementation, at least one of the arguments is dedicated to passing in the thread ID, which each thread uses to help determine its behavior during the parallel section. For example, each thread may use its ID to select which data to operate upon. In this sense, the programming model is essentially the *Single-Program, Multiple-Data* (SPMD) model commonly used in large scale shared memory multiprocessors.

All threads within a parallel section execute in parallel and share the same memory space. Cache coherence and memory consistency are not enforced in our model, as threads running on different shader cores are independent of each other (similar to CUDA applications that do not make use of the "atomic" global memory operations added in CUDA 1.1 [NVIDIA Corp. 2007a]). In the present study, to ensure that threads of the next parallel section will have access to the correct data, data caches are flushed and a memory fence operation is performed at the end of each parallel section.

2.2 SIMD GPU Microarchitecture

Figure 2 illustrates the baseline microarchitecture used in the rest of this article. In this figure, the GPU is composed of several shader cores connected via a interconnection network to several independent DRAM controllers. Each

[3]CUDA version 1.1 and higher allows the CPU to continue execution in parallel with the GPU [NVIDIA Corp. 2007a].

shader core executes multiple parallel threads from the same parallel section, with each thread's instructions executed in order by the hardware.[4] The multiple threads on a given shader core are grouped into SIMD warps by the scheduler. Each warp of threads executes the same instruction simultaneously on different data values in parallel scalar pipelines (i.e., vector lanes). Operands are read from and written to a wide register file in parallel. Memory requests access a highly banked data cache and cache misses are forwarded to memory controllers and/or higher level caches via an interconnection network. Each memory controller processes memory requests by accessing its associated DRAM, possibly in a different order than the requests are received so as to reduce row-activate and precharge overheads [Rixner et al. 2000]. The interconnection network we simulated is a crossbar with a parallel iterative matching allocator [Dally and Towles 2004].

Since our focus in this article is nongraphics applications, graphic-centric details are omitted from Figure 2. However, traditional graphics processing still heavily influences this design: The use of what is essentially SIMD hardware to execute SPMD software (with possible interthread communication) is heavily motivated by the need to balance efficient "general purpose" computing kernel execution with a large quantity of existing and important (to GPU vendors) graphics software that has few, if any, control-flow operations in its shaders [Shebanow 2007]. However, it is important to recognize that thread programs for graphics (i.e., shaders) may very well make increasing use of control flow operations in the future, for example, to achieve more realistic lighting effects.

2.3 Latency Hiding

Because texture workloads in graphics rendering normally exceed the capacity of on-chip caches, cache miss rates can be high even though significant locality exists [Hakura and Gupta 1997], which may severely lower performance if the pipeline had to stall for every cache miss. This is especially true when the latency of memory requests can take several hundreds of cycles due to the combined effects of contention in the interconnection network and row-activate and precharge overheads at the DRAM. While traditional microprocessors can mitigate the effects of cache misses using out-of-order execution, a more compelling approach when software provides the parallelism is to interleave instruction execution from different threads.

With a large number of shader threads multiplexed on the same execution resources, our architecture employs fine-grained multithreading (FGMT), also known as barrel processing, where individual threads are interleaved by the fetch unit to proactively hide the potential latency of stalls before they occur [Thornton 1964; Thistle and Smith 1988]. As illustrated in Figure 3(a), instructions from multiple shader threads are issued fairly in a round-robin queue. When a shader thread is blocked by a memory request, the corresponding shader core simply removes that thread's warp from the pool of "ready" warps and thereby allows other shader threads to proceed while the memory

[4]A shader core in our study is akin to CUDA's notion of a Streaming Multiprocessor [Lindholm et al. 2008].

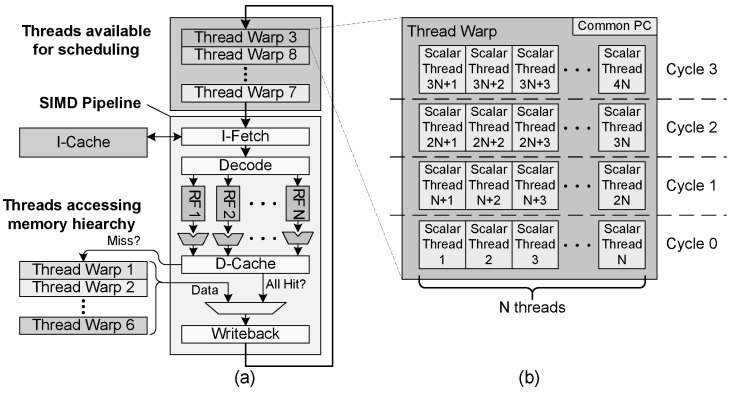


Fig. 3. Detail of a shader core. (a) Using fine-grained multithreading to hide data memory access latency. N is the SIMD width of the pipeline. (b) Grouping 4N scalar threads into a SIMD warp executed over 4 cycles.

system processes its request. With a large number of threads (768 per shader core, 12,288 in total in this article) interleaved on the same pipeline, FGMT effectively hides the latency of most memory operations, since the pipeline is occupied with instructions from other threads while memory operations complete. Barrel processing also hides the pipeline latency so that data bypassing logic can potentially be omitted to save area with minimal impact on performance. In this article, we also simplify the dependency check logic design by restricting each thread to have at most one instruction running in the pipeline at any time.

An alternative to FGMT with a large number of threads is to interleave fewer threads, but provide many registers to each thread [Intel Corporation 2008]. Each thread executes on the pipeline until it encounters a data dependency upon an outstanding long latency memory request, at which time the pipeline will switch execution to another thread. Latency hiding is achieved via software loop unrolling, which generates independent memory access operations, which can be overlapped to achieve the memory level parallelism required for effective use of the graphics memory subsystem.

2.4 SIMD Execution of Scalar Threads

While FGMT can hide memory latency with relatively simple hardware, a modern GPU must also exploit the explicit parallelism provided by the data-parallel programming model [Buck et al. 2004; NVIDIA Corp. 2007a] associated with programmable shader hardware to achieve maximum performance at minimum cost. SIMD hardware [Bouknight et al. 1972] can efficiently support SPMD program execution provided that individual threads follow similar control flow paths. Figure 3(b) illustrates how instructions from multiple ($M = 4N$) shader threads are grouped into a single SIMD warp and scheduled together on multiple (N) scalar pipelines over several ($M/N = 4$) consecutive clock cycles. The multiple scalar pipelines execute in "lock-step" and all control logic may be shared to greatly reduce area relative to a MIMD architecture. A significant

source of area savings for such a SIMD pipeline is the simpler instruction cache support required for a given number of scalar threads.

SIMD can also be used to relax the latency requirement of the scheduler and simplify the scheduler's hardware. With a SIMD warp size wider than the actual SIMD hardware pipeline, the scheduler only needs to issue a single warp every M/N cycles (M=warp size, N=pipeline width) [Lindholm et al. 2008]. The scheduler's hardware is also simplified, as it has fewer warps to manage. Similar to the NVIDIA's GeForce 8 series GPU [Lindholm et al. 2008], the performance evaluations presented in this article assume the use of this technique (see Section 4.4).

3. SIMD CONTROL FLOW SUPPORT

To ensure the hardware can be easily programmed for a wide variety of applications, some recent GPU architectures provide branch instructions that allow individual scalar threads to follow distinct program paths [NVIDIA Corp. 2007a; AMD, Inc. 2006] while executing on SIMD hardware. We note that where it applies, predication [Allen et al. 1983] is a natural way to support fine-grained control flow on a SIMD pipeline. With branches on superword condition codes (BOSCCs) [Shin et al. 2007], a predicated instruction is skipped entirely if all threads in a warp evaluate to false. BOSCCs work effectively for strongly biased branches, such as loops with no early exits and if-branches for error handling where all threads are likely to follow the same control flow. With weakly biased branches, false predicates introduce significant overhead.

To support distinct control flow operation outcomes on distinct processing elements with loops and function calls, several approaches have been proposed: Lorie and Strong describe a mechanism using mask bits along with special compiler-generated priority encoding "else" and "join" instructions [Lorie and Strong 1984]. Lindholm and Moy [2005] describe a mechanism for supporting branching using a serialization mode. Woop et al. [2005] describe the use of a hardware stack and masked execution. Kapasi et al. [2000] propose conditional streams, a technique for transforming a single kernel with conditional code to multiple kernels connected with interkernel buffers. These and other approaches are discussed in Section 9.

The effectiveness of a SIMD pipeline is based on the assumption that all threads running the same thread program expose identical control-flow behavior. While this assumption holds for most existing graphics shaders [Shebanow 2007], many parallel nongraphics applications (and potentially, future graphics shaders) tend to have more diverse control-flow behavior. When a branch leads to different control flow paths for different threads in a warp, branch divergence occurs because a SIMD pipeline cannot execute different instructions in the same cycle. The following sections describe two baseline techniques for handling branch divergence, both of which were implemented in our simulator.

3.1 SIMD Serialization

A naïve solution to handle branch divergence in a SIMD pipeline is to serialize the threads within a warp after their program counters diverge. While this

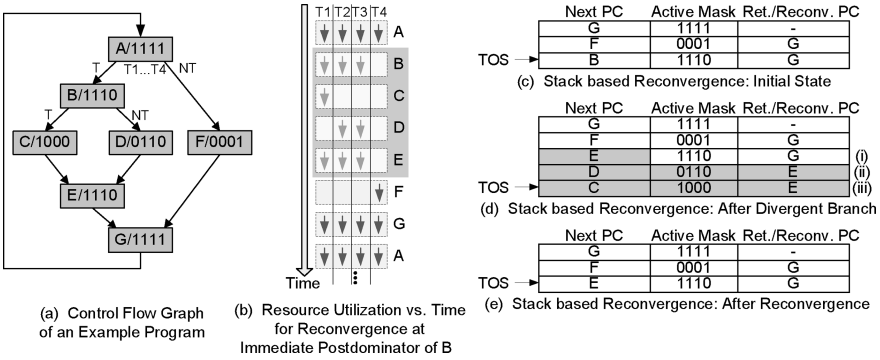


Fig. 4. Implementation of immediate postdominator-based reconvergence.

approach is simple, it achieves poor performance. A single warp with branch divergence is effectively separated into multiple warps, each containing threads taking the same execution path. These warps are then scheduled and executed independently of each other, and they never reconverge back into a single warp. Without branch reconvergence, threads within a warp will continue diverging until each thread is executed in isolation from other threads in the original warp, leading to very low utilization of the parallel functional units, as shown by the first bar in Figure 1.

3.2 SIMD Reconvergence (Baseline Branch Handling Mechanism)

Given the drawback of serialization it is desirable to use a mechanism for reconverging control flow. The opportunity for such reconvergence is illustrated in Figure 4(a). In this example, threads in a warp diverge after reaching the branch at A. The first three threads encounter a taken branch and go to basic block[5] B (denoted by the bit mask 1110 in Figure 4(a)), while the last thread goes to basic block F (denoted by the bit mask 0001 in Figure 4(a)). The three threads executing basic block B further diverge to basic blocks C and D. However, at basic block E the control-flow paths reach a join point [Muchnick 1997]. If the threads that diverged from basic block B to C wait before executing E for the threads that go from basic block B to basic block D, then all three threads can continue execution simultaneously at block E. Similarly, if these three threads wait after executing E for the thread that diverged from A to F, then all four threads can execute basic block G, simultaneously. Figure 4(b) illustrates how this sequence of events would be executed by the SIMD function units. In this part of the figure, solid arrows indicate SIMD units that are active.

The behavior described earlier can be achieved using a stack-based reconvergence mechanism [Woop et al. 2005]. In this article, we use the mechanism shown in Figure 4(c,d,e) as our baseline. Here, we show how the stack is updated as the group of three threads in Figure 4(a) that execute B diverge and

[5]A basic block is a sequence of consecutive statements in which flow of control enters only at the beginning and leaves only at the end without the possibility of branching except at the end [Aho et al. 1986].

then reconverge at E. Before the threads execute the diverging branch at B, the state of the stack is as shown in Figure 4(c). When the branch divergence is detected after executing B, the stack is modified to the state shown in Figure 4(d). The changes that occur are the following:

(1) The original top of stack (TOS) in Figure 4(c), also at (i) in Figure 4(d), has its next PC field modified to the instruction address of the reconvergence point E (this address could be specified through an extra field in the branch instruction);
(2) A new entry (ii) is allocated onto the stack and initialized with the next PC value of the fall through of the branch (D) and a mask encoding, which processing elements evaluated the branch as "not-taken" (0110) along with the reconvergence point address (E);
(3) A new entry (iii) is allocated onto the stack with the target address (C) of the branch, the mask (1000) encoding the processing element that evaluated the branch as "taken", and the reconvergence point address (E).

Whenever the new next PC of the top entry on the stack equals the reconvergence PC of the TOS entry, the top entry is immediately popped (before being used to fetch an instruction), and the value of the next PC field from the next entry in the stack is used to fetch the next instruction. Note that this mechanism easily supports complex "nested" branch hammocks, irreducible control flow [Muchnick 1997], as well as data-dependent loops.

In this article we use the immediate postdominator [Muchnick 1997] of the diverging branch instruction as the reconvergence point.[6] A *postdominator* is defined as follows: A basic block X postdominates basic block Y (written as "X pdom Y") if and only if all paths from Y to the exit node (of a function) go through X. A basic block X, distinct from Y, immediately postdominates basic block Y if and only if X pdom Y and there is no basic block Z distinct from Y and X such that X pdom Z and Z pdom Y. Immediate postdominators are typically found at compile time as part of the control flow analysis necessary for code optimization.

The performance impact of the immediate postdominator reconvergence technique (labeled PDOM throughout this article) depends upon the SIMD warp size. Figure 5 shows the harmonic mean IPC of the benchmarks studied in Section 6 compared to the performance of MIMD hardware for 8, 16, and 32 wide SIMD execution assuming 16 shader cores. Execution unit utilization decreases from 26.3% for MIMD to 21.1% for 8-thread, to 18.7% for 16-thread, and down to 15.8% for 32-thread SIMD warps.[7] This increasing performance loss is due to the higher impact of branch divergence when SIMD warps are wider and each warp contains more threads. Thus, using 32-thread SIMD warps results in a 40% slowdown, so performance can potentially improve by 66% with a better branch handling mechanism.

[6]While Rotenberg et al. [1999] also identified immediate postdominators as control reconvergence points in the context of superscalar processor, to our knowledge, we are the first to propose this scheme for SIMD control flow.
[7]MIMD is < 100% due to the memory bandwidth requirements of our benchmarks.

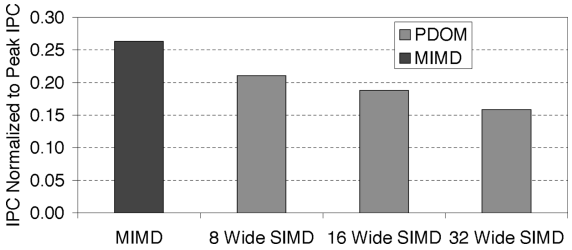


Fig. 5. Performance loss for PDOM versus SIMD warp size (realistic memory system). An 8-wide SIMD pipeline is used for all configurations.

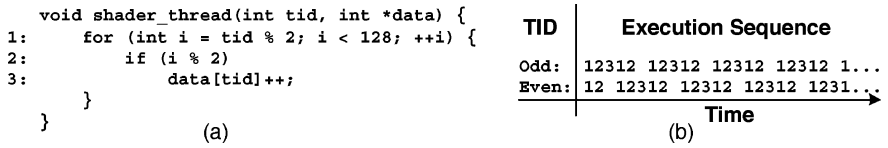


Fig. 6. (a) Contrived example for which reconvergence at points beyond the immediate postdominator yields a significant improvement in performance like that shown in Figure 7. The parameter `tid` is the thread ID. (b) Execution sequence for threads with odd or even `tid` (numbers refer to lines in part(a)).

In the following section, we explore whether immediate postdominators are the "best" reconvergence points, or whether there might be a benefit to dynamically predicting a reconvergence point past the immediate postdominator (we will show a contrived code example where this is beneficial).

3.3 Reconvergence Point Limit Study

While a mechanism for reconverging at the immediate postdominator is able to recover much of the performance lost due to branch divergence compared to not reconverging at all, Figure 6(a) shows an example where this reconvergence mechanism is suboptimal. In this example, threads with even value of `tid` diverge from those with odd values of `tid` each iteration of the loop. If even threads allow the odd threads to "get ahead" by one iteration, all threads can execute in lock-step until individual threads reach the end of the loop. This suggests that reconverging at points beyond the immediate postdominator may yield better performance. To explore this possibility, we conducted a limit study, assessing the impact of always predicting the best reconvergence point assuming oracle knowledge of each thread's future control flow.

For this limit study, dynamic instruction traces are captured from only the first 128 threads. SIMD warps are formed by grouping threads by increasing thread ID, and an optimal alignment for the instruction traces of each thread in a warp is found via repeated applications of the Needleman-Wunsch algorithm [Needleman and Wunsch 1970]. With four threads per warp, the optimal alignment is found by exhaustively searching all possible pair-wise alignments between threads within a warp. The best reconvergence points are then identified from the optimal alignment.

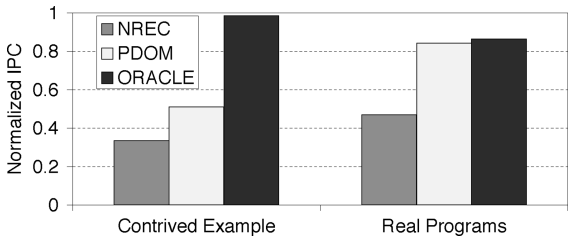


Fig. 7. Impact of predicting optimal SIMD branch reconvergence points (SIMD width= 4). NREC=No Reconvergence. PDOM=reconvergence at immediate postdominators. ORACLE= reconvergence at optimal postdominators.

Figure 7 compares performance of not reconverging (NREC), immediate postdominator reconvergence (PDOM), and reconverging to the points found using the previously described method (ORACLE), assuming a warp size of 4. In this figure, we assume an idealized memory system (all cache accesses hit) and examine both a contrived program with the behavior abstracted in Figure 6 and the seven benchmarks described in Section 5 (depicted by the bars labeled Real Programs). While the contrived example experiences a 92% speedup with ORACLE versus PDOM, the improvement on the real programs we studied is much less (2.6%). Interestingly, one of the benchmarks (bitonic sort) also has similar even/odd thread dependence as our contrived example, but it also has barrier synchronizations forcing loop iterations to run in lock-step. While this limit study indicates that reconverging at the immediate postdominator is a good heuristic, in the next section, we present a mechanism for improving performance significantly relative to PDOM.

It is important to recognize the limitations of this limit study: We explored a limited set of benchmarks and used short SIMD width. It may be valuable to repeat this study on a larger set of applications with wider SIMD width.

4. DYNAMIC WARP FORMATION AND SCHEDULING

While the postdominator reconvergence mechanism can recover performance loss due to diverging branches, it does not fully utilize the SIMD pipeline relative to a MIMD architecture with the same peak IPC capability (resulting in a 40% slowdown relative to MIMD for a warp size of 32). In this section, we describe our proposed hardware mechanism for recovering the lost performance potential of the hardware.

The performance penalty due to branch divergence is hard to avoid with only one thread warp, since the diverged parts of the warp cannot execute simultaneously on the SIMD hardware in a single cycle. Dynamic warp formation attempts to improve upon this by exploiting the FGMT aspect of the GPU microarchitecture: With FGMT employed to hide memory-access latency, there is usually more than one thread warp ready for scheduling in a shader core. Every cycle, the thread scheduler tries to form new warps from a pool of active threads by combining scalar threads whose next program counter (PC) values are the same. As the thread program executes, diverged warps are broken up into scalar threads to be regrouped into new warps according to their branch

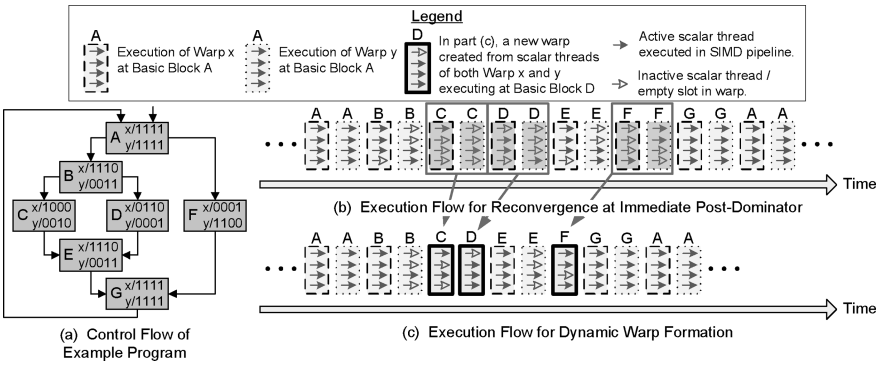


Fig. 8. Dynamic warp formation example.

targets (indicated by the next PC value of each scalar thread). In this way, the SIMD pipeline is fully utilized even if a thread program consists of diverging branches.

Figure 8 illustrates this idea. In this figure, two warps, Warp x and Warp y, are executing the example program shown in Figure 8(a) on the same shader core and both suffer from branch divergence. Figure 8(b) shows the interleaved execution of both warps using the stack-based reconvergence technique discussed in Section 3.2, which results in a SIMD pipeline utilization below 50% when basic blocks C, D, and F are executed. As shown in Figure 8(c), using dynamic warp formation to regroup scalar threads from both warps in Figure 8(b) into a single warp in Figure 8(c) with more active threads for these blocks can significantly increase average pipeline utilization (from 66% to 81%).

Implementing dynamic warp formation requires careful attention to the details of the register file, a consideration we explore in Section 4.1. In addition to forming warps, the thread scheduler also selects one warp to issue to the SIMD pipeline every scheduler cycle depending upon a scheduling policy. We explore the design space of this scheduling policy in Section 4.3. We show that thread scheduler policy is critical to the performance impact of dynamic warp formation in Section 6.1.

4.1 Register File Access

To reduce area and support high-bandwidth access to data in a SIMD pipeline, a well-known approach is to implement a register file with registers for different processing elements (lanes of a SIMD pipeline) packed into a single-wide register. Registers for different elements are accessed by a single decoder, as shown in Figure 9(a). This hardware is a natural fit when threads are grouped into warps "statically" before they begin executing instructions and each thread stays in the same lane until it is completed. For our baseline architecture, each wide register contains a scalar register for each lane. Each scalar thread in a warp is statically assigned to a unique lane and always accesses the corresponding portion of the wide registers associated with that lane. The registers used by each scalar thread within a given lane are then assigned statically at a given offset based on the warp ID.

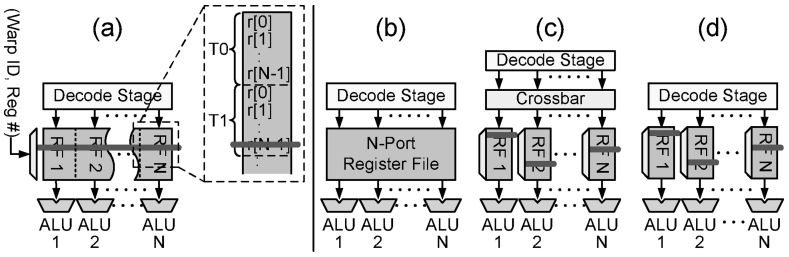


Fig. 9. Register file configuration for (a) static warp formation, (b) ideal dynamic warp formation and MIMD, (c) unconstrained dynamic warp formation with potential for bank conflicts, (d) lane aware dynamic warp formation. The line running across each lane represents whether registers in different lanes are all addressed by a common decoder (continuous line in part (a)) or each lane has its own decoder for independent addressing ((c) and (d)). In part (a), the blown up section illustrates the layout of scalar registers (r[0]...r[N-1]) for individual threads (T0, T1) in a lane.

However, we have not explicitly considered the impact of such static register assignment on dynamic warp formation. As described so far, dynamic warp formation would require each register to be equally accessible from all lanes, as illustrated in Figure 9(b). While grouping threads into warps dynamically, it is preferable to avoid the need to migrate register values with threads as they are regrouped into different warps. To accomplish this, organizations, such as those shown in Figure 9(b,c,d), allow the registers used by each scalar thread to be assigned statically to the lanes in the same way as described previously. If we dynamically form new warps from scalar threads with identical next PC values without consideration of the "home" lane of a scalar thread's registers but wish to avoid a heavily mulitported register file organization, such as in Figure 9(b), we must design a multibanked register file, where each bank contains registers of threads with the same "home" lane, with a crossbar as in Figure 9(c). Warps formed dynamically may then have two or more threads with the same "home" lane, resulting in bank conflicts. These bank conflicts introduce stalls into all lanes of the pipeline and significantly reduce performance, as shown in Section 6.4.

A better solution, which we call *lane aware* dynamic warp formation, ensures that each thread remains within its "home" lane. In particular, lane aware dynamic warp formation assigns a thread to a warp only if that warp does not already contain another thread in the same lane. While the crossbar in Figure 9(c) is unnecessary for lane aware dynamic warp formation, the traditional hardware in Figure 9(a) is insufficient. When a particular set of threads are grouped into a warp "statically," each thread's registers are at the same offset within the corresponding lane, thus requiring only a single decoder. With lane aware dynamic warp formation, the offsets to access a register in a warp will not be the same in each lane.[8] This yields the register file configuration shown in Figure 9(d), which is used for our performance and area estimations in Section 6 and Section 7.

[8]Note that each lane is still executing the same instruction in any given cycle—the varying offsets are a byproduct of supporting fine-grained multithreading to hide memory access latency combined with dynamic warp formation.

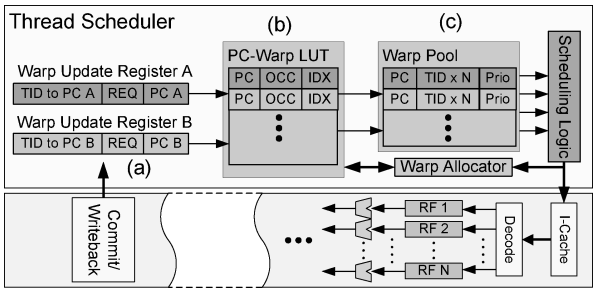


Fig. 10. Implementation of dynamic warp formation and scheduling. In this figure, N is the width of the SIMD warp. See text for a detailed description.

One subtle performance issue affecting the impact of lane aware scheduling for one of our benchmarks (Bitonic) is related to the type of pathological even/odd thread identifier control dependence described in Section 3.3. For example, if all warps have threads in even lanes that evaluate a branch as taken, while threads in odd lanes evaluate the same branch as not-taken, then it is impossible for dynamic warp formation to create warps with more active threads. A simple solution we employ for all applications is to alternately swap the position of even and odd thread home lanes every other warp when threads are first created (an approach we call *thread swizzling*).

4.2 Hardware Implementation

Figure 10 shows a high-level block diagram illustrating how dynamic warp formation can be implemented in hardware. The two warp update registers, labeled (a) in Figure 10, store information for different target PCs of an incoming, possibly diverged warp; the PC-warp LUT (b) provides a level of indirection to guide diverged threads to an existing or newly created warp in the warp pool (c). The warp pool is a staging area holding all the warps ready to be issued to the SIMD pipeline. A detailed implementation of the scheduling logic, assuming the Majority scheduling policy introduced in Section 4.3, is proposed in Section 4.4.

When a warp arrives at the last stage of the SIMD pipeline, its threads' identifiers (TIDs) and next PC(s) are passed to the thread scheduler (Figure 10(a)). For conditional branches, there are at most two different next PC values.[9] For each unique next PC sent to the scheduler from writeback, the scheduler looks for an existing entry in the PC-warp LUT already mapped to the PC and allocates a new entry if none exists[10] (Figure 10(b)).

The PC-warp LUT (Figure 10(b)) provides a level of indirection to reduce the complexity of locating warps in the warp pool (Figure 10(c)). It does this by using the IDX field to point to a warp being formed in the warp pool. This warp is updated with the thread identifiers of committing threads having this

[9]Indirect branches that diverge to more than two PCs can be handled by stalling the pipeline and sending up to two PCs to the thread scheduler every cycle.

[10]In our detailed model, we assume the PC-warp LUT is organized as a small dual-ported 4-way set associative structure. Sets in the PC-warp LUT are indexed by the lower bits of the PC.

next PC value. Each entry in the warp pool contains the next PC value of the warp, N TID entries for hardware supporting N-wide warps, and some policy-specific data (labelled "Prio") for scheduling logic. A design to handle the worst case where each thread diverges to a different execution path would require the warp pool to have enough entries for each thread in a shader core to have its own entry. However, we observe that for the benchmarks we simulated only a small portion of the warp pool is used, and we can shrink the warp pool significantly to reduce area overhead without causing a performance penalty (see Section 6.8). This much smaller warp pool can still support the worst-case where each thread has diverged, by employing the spilling mechanism described in Section 4.5.

To implement the lane aware scheduler mentioned in Section 4.1, each entry in the PC-warp LUT has an occupancy vector (OCC) tracking, which lanes of the current warp are free. This is compared against the request vector (REQ) of the warp update register that indicates which lanes are required by the threads assigned to one portion of a newly diverged warp. If OCC indicates that a required lane is already occupied by a thread in the warp pointed to by IDX, a new warp will be allocated and the TIDs of the threads causing the conflict will be assigned into this new warp. The TIDs of the threads that do not cause any conflict will be assigned to the original warp pointed to by IDX. In the case of such conflicts, the PC-warp LUT IDX field is updated to point to the new warp in the warp pool. The warp with the older PC still resides in the warp pool but will no longer be updated.

Each scheduler cycle a single warp in the warp pool may be issued to the SIMD pipeline according to one of the scheduling policies described in the next section. A forwarding path is added to allow updating of the warp that is issued in the same scheduler cycle to allow concurrently merging warps (after a decision has been made to issue the warp). This is possible because instruction cache access does not require TID information, so the TID can potentially be updated as the warp traverses the fetch stage, reducing the effective pipeline depth. Once issued, the associated warp pool entry is returned to the warp allocator.

4.3 Scheduling Policies

Even though dynamic warp formation has the potential to fully utilize the SIMD pipeline, this will only happen when the set of PC values currently being executed is small relative to the number of scalar threads. If each scalar thread progresses at a substantially different rate, then all threads will eventually map to entirely different PCs. To avoid this, all threads should have a similar rate of progress. We have found that the warp scheduling policy, namely, the order in which warps are issued from the warp pool, has a critical effect on performance due to this effect (as shown in Section 6.1). We evaluated the following policies:

Time Stamp (DTime). Warps are issued in the order they arrive at the scheduler. When a warp triggers a data cache miss, it is taken out of the scheduler until its requested data arrives from memory. This may change the order that warps are issued.

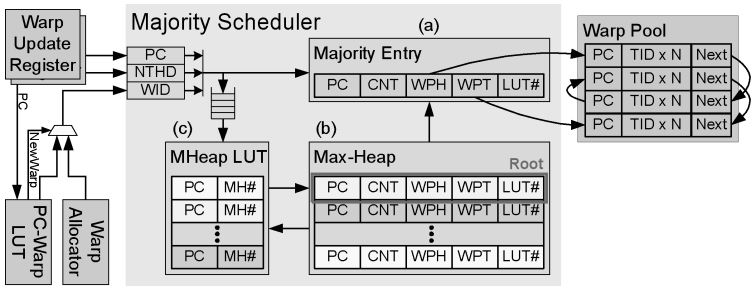


Fig. 11. Implementation of majority scheduling policy with Max-Heap. See text for details.

Program Counter (DPC). In a program sequentially laid out in instruction memory, the program counter value itself may be a good indicator of a thread's progress. By giving higher issue priority to warps with smaller PCs, threads lagging behind are given the opportunity to catch up.

Majority (DMaj). As long as a majority of the threads are progressing at the same rate, the scheduling logic will have a large pool of threads with similar PC values from which it can create new warps every cycle. The majority policy attempts to encourage this behavior by choosing the most common PC among all the existing warps and issuing all warps at this PC before choosing a new PC.

Minority (DMin). If a small minority of threads diverges away from the rest, the majority policy tends to leave these threads behind. In the minority policy, warps with the least frequent PCs are given priority with the hope that, by doing so, the threads in these warps may eventually catch up and converge with other threads.

Postdominator Priority (DPdPri). Threads falling behind after a divergence need to catch up with other threads after the immediate postdominator. If the issue priority is set lower for warps that have gone beyond more postdominators, then the threads that have not yet gone past the postdominator tend to catch up.

In Section 6, we show that Majority scheduling policy achieves the best result over the benchmarks we evaluated. In Section 6.2, we show that DPdPri's remedies certain performance deficiencies of DMaj identified for one of our benchmarks with simple control-flow behavior. While DPdPri is better in some cases, the number of postdominators encountered by a thread does not always accurately represent a thread's progress through a program with nested control flow, resulting in lost opportunities to group threads into warps. The simulation results in Section 6.4 that take into account a realistic implementation with a Max-Heap indicate that, overall, the DMaj scheduling policy offers better performance.

4.4 A Majority Scheduling Policy Implementation

Majority scheduling, the policy we found to work best (as shown in Section 6.1), can be implemented in hardware with a Max-Heap and a look-up table, as shown in Figure 11. The performance and area impact of this hardware implementation is carefully evaluated in Sections 6 and 7, respectively. In Figure 11, the Majority Entry (a) keeps track of the current majority PC value

and a group of warps with this PC value[11]; the Max-Heap (b) is a full binary tree (including a root entry) of PC values (and its group of warps) sorted by number of threads with this PC using the algorithm described in Cormen et al. [2001], and the MHeap LUT (c) provides a level of indirection for incoming warps to update their corresponding entries in the Max-Heap (similar to the function of the PC-warp LUT but for the Max-Heap rather than the Warp Pool). While a simple Max-Heap performing one swap per cycle per updated warp is sufficient for our usage, using a pipelined max-heap structure, such as those explored by Ioannou and Katevenis [2007] and Basu et al. [2007], may provide an even better hardware implementation by further reducing the bandwidth (and hence area) requirements of the Max-Heap.

Each entry in the Max-Heap represents a group of warps with the same PC value, and this group of warps has CNT threads. This group of warps forms a "linked list" in the warp pool with a "Next" entry referring to the next warp in the list. WPH is the index of the head of this list in the Warp Pool and WPT is the index of the tail of the list (warps are inserted at the end of the "list"). LUT# is the index of the corresponding entry in the MHeap LUT and is used to update the corresponding MH# entry in the LUT after a swap (see details later in the text).

4.4.1 *Warp Insertion.* In the implementation of the majority scheduler that we consider, a 32-thread warp is issued over 4 cycles. Once a (possibly diverged) warp leaves the writeback stage of the pipeline and enters the scheduler, it is processed as follows. When a warp arrives at the thread scheduler, its threads are divided among the two warp update registers, as before. Each warp update register then sends a warp update consisting of the following information to the Majority scheduler: the warp's next PC value (labeled PC in Figure 11), the number of threads of this warp (NTHD), and the ID of the warp in the warp pool (WID) (acquired either via the PC-warp LUT or warp allocator, as in Section 4.2). Inside the Majority scheduler, the next PC value of each incoming warp update request is compared against the one stored in the Majority Entry. If they match, the Majority Entry is updated directly; otherwise, the warp update request is pushed into an update queue to the Max-Heap.

Every cycle up to two warp updates are processed in parallel from the update queue. For each update, the MHeap LUT is searched to provide the index (MH#) to an existing Max-Heap entry with a matching next PC value. If such an entry is not found in the MHeap LUT, an index of a new entry in the Max-Heap is obtained. The Max-Heap entry, at the location identified by MH#, is then updated (toward the end of the same cycle): CNT is incremented by NTHD, and if WID is different from WPT, WPT will be updated with WID, and WID will be assigned to the "Next" entry in the warp referred to by the original WPT. (This is essentially attaching the warp referred to by WID to the end of the linked list of this Max-Heap entry). If this is a newly inserted entry in the Max-Heap, the value of WID will be assigned to both WPH and WPT.

[11]This entry is separated because we finish executing all warps at a given PC value before selecting a new PC value.

In subsequent cycles after the entries in the Max-Heap are updated or inserted, each of these modified entries is compared with its parent entry and swapped up the binary tree until it encounters a parent with a larger thread count (CNT) or it becomes the root of the binary tree (which may be popped from the Max-Heap and becomes the next Majority Entry). Whenever a swap occurs, the corresponding entry in MHeap LUT (denoted by LUT# in the swapping Max-Heap entries) is updated. During this operation, no entries in the Max-Heap can be updated. New warp updates generated by incoming warps in the meantime are buffered in the update queue.

4.4.2 *Warp Issue.* In every scheduler cycle,[12] a warp from the warp pool will be issued to the SIMD pipeline. If the Majority Entry has any warps remaining (CNT>0), the warp denoted by WPH will be issued to the pipeline. WPH will then be updated with the value in the "Next" entry of the issued warp, and CNT will be decremented by the number of threads in the issued warp. If the Majority Entry runs out of warps (CNT=0), the root entry of the Max-Heap will be popped and will become the Majority Entry. If the Max-Heap is in the process of rebalancing itself after a warp insertion, no warp will be issued until the Max-Heap is eventually balanced. The Max-Heap balances itself over multiple cycles according to the algorithm described by Cormen et al. [2001] (bandwidth constrained as in Table V). The MHeap LUT is updated in parallel to any swap operation, and again, before the Max-Heap is rebalanced, none of its entries can be updated by warps arriving at the scheduler.

4.4.3 *Complexity.* As the Max-Heap is a tree structure, every warp insertion only requires up to $log_2(N)$ swaps to rebalance for a N-entry max-heap. Experimentally, we find that usually fewer swaps are required. We find in Section 6.3 that a design with both the Max-Heap and the MHeap LUT each having two read ports and two write ports is sufficient to keep the Max-Heap balanced in the common case (handling two warp insertions from each part of the incoming warp every scheduler cycle[12]). This is assuming that both structures are clocked as fast as the SIMD pipeline, so that a total of eight reads and eight writes can be performed in every scheduler cycle.[12]

When the Max-Heap is rebalancing, updates from incoming warps are buffered in a queue and will be performed in subsequent cycles. This does not disrupt the scheduler as long as the updates are done before a new Majority PC is needed. If this queue is full, the scheduler stops accepting warps, which stalls the pipeline for a finite duration until the Max-Heap is rebalanced again and is free to process the next update from the queue. We find that a four-entry queue is enough to avoid stalling the pipeline for all of our benchmarks. If the number of PCs in flight exceeds the Max-Heap capacity, a spilling mechanism as described in Section 4.5 is used. However, we observe in Section 7 that with a 64-entry Max-Heap, the need for spilling never arose for our benchmarks.

[12]A scheduler cycle is equivalent to several pipeline cycles, because each warp can take several cycles (e.g., 4 for a warp size of 32) to execute in the pipeline (see Section 2.4), so the scheduler may operate at a slower frequency.

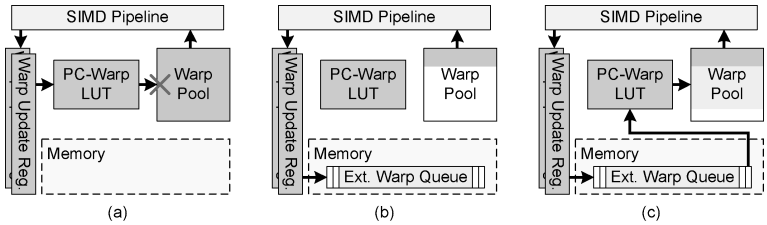


Fig. 12. Warp spilling mechanism to handle excessively diverging code.

4.4.4 *Applicability to Other Scheduling Policies.* The previously described hardware implementation of the Majority scheduling policy can be modified to implement other policies. DMin can be implemented by using a Min-Heap in place of the Max-Heap. The priority criterion can be changed to the PC of the group to implement DPC. On the other hand, with DPdPri, the priority of a warp changes when incoming threads with higher priority are assigned to the warp. This volatility of warp priorities renders the use of the Max-Heap impractical as it will cause the Max-Heap to rebalance frequently. The wide range of priorities also makes it hard to group the warps into a small number of entries in the Max-Heap. Note that DTime can be implemented with a circular FIFO.

4.5 Handling Excessively Diverging Code

While our programming model does allow each thread to diverge to its unique execution path, sizing the warp pool for this unlikely case is wasteful, as most entries would be unused in the common case (as shown in Section 6.8). With a smaller warp pool, dynamic warp formation can still correctly execute code that results in excessively diverging code by spilling entries to memory when the warp pool overflows.

When the warp pool overflows, the scheduler starts writing incoming warps (with threads grouped according to their next PC) directly to an external warp queue in memory, bypassing the PC-warp LUT and scheduling logic (as shown in Figure 12(a) and (b)). Warps already in the warp pool are issued until the warp pool drains. As these warps are drained from the warp pool, it is re-populated by groups of warps loaded from the external queue (as shown in Figure 12(c)). Warps from the external queue arrive at the scheduler can be merged into existing warps inside the warp pool via the PC-warp LUT, using the same mechanism described in Section 4.2. Late arrival of warps due to memory latency does not stall the pipeline, but the warp pool may be drained to empty in the meantime, taking away opportunities for warps to be merged. This creates a FIFO warp update queue with virtually unlimited capacity. The external warp queue can be managed as a circular buffer with three hardware registers: an offset pointer to the circular buffer and two pointers to head and tail of the queue tracking where to spill and fill warps.

We leave a more-detailed exploration and performance evaluation of this spilling mechanism as future work.

Table I. Hardware Configuration

Shader Core Pipeline Frequency	650 MHz
# Shader Cores	16
SIMD Warp Size	32 (issued over 4 clocks)
SIMD Pipeline Width	8
# Threads per Shader Core	768
# Memory Modules	8
DRAM Control Bus Frequency	650 MHz
GDDR3 Memory Timing	t_{CL}=9, t_{RP}=13, t_{RC}=34 t_{RAS}=21, t_{RCD}=12, t_{RRD}=8
Bandwidth per Memory Module	8Byte/Cycle (5.2GB/s)
Memory Controller	out of order
Data Cache Size (per core)	512KB 8-way set assoc.
# Data Cache Banks (per core)	16
Data Cache Hit Latency	10 cycle latency (pipelined 1 access/thread/cycle)
Default Warp Scheduling Policy	Majority
PC-Warp LUT	64 entries, 4-way set assoc.
MHeap LUT	128 entries, 8-way set assoc.
Max-Heap	64 entries

5. METHODOLOGY

While simulators for contemporary GPU architectures exist [Sheaffer et al. 2004; del Barrio et al. 2006], none of them we are aware of currently model the general-purpose GPU computing architecture described in this article. Therefore, we developed a novel simulator, *GPGPU-Sim*, to model various aspects of the massively parallel architecture used in modern GPUs with highly programmable pipelines. GPGPU-Sim was constructed "from the ground up" starting from the instruction set simulator (sim-safe) of SimpleScalar version 3.0d [Burger and Austin 1997]. We developed our cycle-accurate GPU performance simulator, modeling the microarchitecture described in Section 2.2, around the SimpleScalar PISA instruction set architecture and then interfaced it with sim-outorder, which only provides timing for code running on the CPU in Figure 2. For the purpose of this article, the PISA instruction set architecture is a good representation of PTX, the virtual ISA used in the CUDA programming model [NVIDIA Corp. 2007b]. While lacking multipy-add, transcendental, and texture instructions, PISA features control flow instructions similar to PTX. PISA lacks support for predication; however, on our hardware model, we believe PDOM captures much of the benefits of predication. Table I shows our baseline configuration.

The SimpleScalar out-of-order core waits for the GPU when it reaches a parallel section. After GPU simulation of the compute kernel is completed, program control is returned to the out-of-order core. This repeats until the benchmark finishes.

The benchmark applications used for this study were selected from SPEC CPU2006 [Standard Performance Evaluation Corporation], SPLASH2 [Woo et al. 1995], and CUDA [NVIDIA Corp.]. Each benchmark was manually modified to extract and annotate the computing kernels, which is a time-consuming task, limiting the number of benchmarks we could consider. The

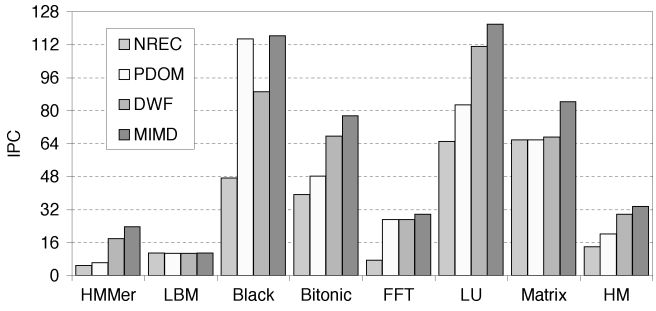


Fig. 13. Performance comparison of NREC, PDOM, and DWF versus MIMD.

programming model we assume is similar to that of CUDA [NVIDIA Corp. 2007a]. In our detailed simulation, a computing kernel is invoked by a spawn instruction, which signals the out-of-order core to launch a predetermined number of threads for parallel execution on the GPU simulator. If the number of threads to be executed exceeds the capacity of the hardware configuration, the software layer is responsible for organizing threads into *blocks* in our model. Threads within a block are assigned to a single shader core, and all of them have to finish before the shader core can begin with a new block.

6. EXPERIMENTAL RESULTS

First, we consider dynamic warp formation and scheduling modeling the implementation described in Section 4.2 using the Majority scheduling policy implemented with the Max-Heap modeled, as described in Section 4.4. The hardware is sized as in Table V and employs the thread swizzling mechanism described in Section 4.1. This implementation uses the lane aware scheduling described in Sections 4.1 and 4.2. We also model bank conflicts at the data cache.

Figure 13 shows the performance of the different branch-handling mechanisms discussed in this article and compares them to a MIMD pipeline with the same peak IPC capability. Here, we use the detailed simulation model described in Section 5 including simulation of memory access latencies. On average (HM in Figure 13), PDOM (reconvergence at the immediate postdominator described in Section 3.2) achieves a speedup of 45% versus not reconverging (NREC). Dynamic warp formation (DWF) achieves a further speedup of 47% using the Majority scheduling policy. The average DWF and MIMD performance differs by only 11%. We believe the 47% speedup of DWF versus PDOM justifies the estimated 8% area cost (Section 7) of incorporating dynamic warp formation and scheduling into future GPUs. We note that this magnitude of speedup could not be obtained by simply spending this additional area on extra shader cores.

For benchmarks with little diverging control flow, such as FFT and Matrix, DWF performs as well as PDOM, while MIMD outperforms both of them with its "free running" nature. The significant slowdown of Black-Scholes (Black) with DWF is a phenomenon exposing a weakness of our default Majority scheduling policy. This will be examined in detail in Section 6.2.

Table II. Memory Bandwidth Utilization

	HMMer	LBM	Black	Bitonic	FFT	LU	Matrix
PDOM	57.47%	94.71%	8.18%	50.94%	75.43%	0.84%	84.02%
DWF	63.61%	95.04%	6.47%	71.02%	74.89%	1.47%	63.71%
MIMD	64.42%	95.07%	8.30%	81.21%	81.49%	1.58%	99.23%

Table III. Cache Miss Rates Pending Hits[13] Classified as a Miss

	HMMer	LBM	Black	Bitonic	FFT	LU	Matrix
PDOM	13.37%	14.15%	1.52%	21.25%	15.52%	0.05%	7.88%
DWF	5.05%	14.56%	1.50%	30.67%	14.68%	0.06%	9.71%
MIMD	3.74%	15.34%	1.17%	30.67%	13.79%	0.05%	7.30%

Table IV. Cache Miss Rates Without Pending Hits[13]

	HMMer	LBM	Black	Bitonic	FFT	LU	Matrix
PDOM	13.21%	13.55%	0.21%	3.86%	11.77%	0.03%	5.72%
DWF	5.03%	13.57%	0.21%	3.84%	12.53%	0.04%	4.23%
MIMD	3.73%	13.36%	0.21%	3.83%	12.06%	0.04%	5.26%

For most of the benchmarks with significant diverging control flow (HMMer, LBM, Bitonic, LU), MIMD performs the best, and DWF achieves a significant speedup over PDOM. Among them, DWF achieves a speedup for Bitonic and LU purely from better branch-divergence handling, while for HMMer, DWF achieves a speedup in part due to better cache locality as well, as observed from Table III. While LBM also has significant diverging control flow, it is memory bandwidth limited, as shown in Table II and hence sees little gain from DWF. Although the cache miss rate of Bitonic is higher than that of LBM, its much higher pending hit[13] rate significantly lowers the memory bandwidth requirement of this benchmark (see Table IV).

6.1 Effects of Scheduling Policies

In this section and in Section 6.2, we evaluate scheduling policies assuming (a) an idealized scheduler able to sort warps in a single cycle to account for changing priorities (i.e., not using the Max-Heap implementation in Figure 11), (b) no restrictions on the home lane of threads grouped into a warp, (c) the multiported register file shown in Figure 9(b) with an unlimited number of ports (i.e., ignoring the effects of lane conflicts), and (d) a realistic memory subsystem.

Figure 14 compares all the warp scheduling policies described in Section 4.3 to show the potential of each policy. Overall, the default Majority (DMaj) policy performs the best, achieving an average speedup of 66.0%, but in some cases, its performance is not as good as the PC policy (DPC) or PDOM Priority (DPdPri) policy.

To provide additional insight into the differences between the scheduling policies, Figure 15 shows the active thread count distribution of warp issued

[13]A pending hit occurs when a memory access misses the cache, but can be merged with one of the in-flight memory requests already sent to the memory subsystem.

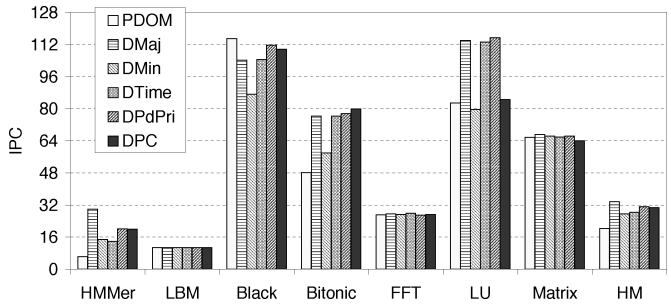


Fig. 14. Comparison of warp scheduling policies. The impact of lane conflicts is ignored.

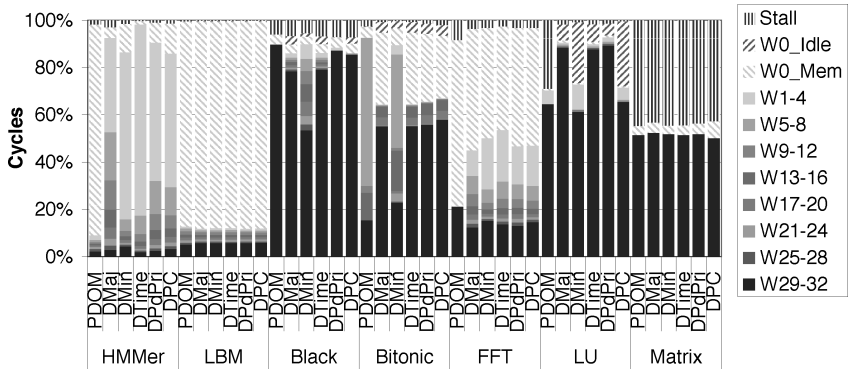


Fig. 15. Active thread count distribution.

each cycle for each policy. Each bar is divided into segments labeled W0, W1-4, ... W29-32, which indicate the fraction of total execution cycles the scheduler issued a warp with 0, (1 to 4), ...(29 to 32) scalar threads. "Stall" indicates a stall due to writeback contention with the memory system (see Figure 3(a)). For policies that do well (DMaj, DPdPri, DPC), we see a decrease in the number of low-occupancy warps relative to those policies, which do poorly (DMin, DTime). Cycles with no scalar threads executed (W0) are classified into "Mem" (W0_Mem) and "Idle" (W0_Idle). W0_Mem denotes the cycles when all threads are waiting for data from global memory. W0_Idle denotes that all warps within a shader core have already been issued to the pipeline and are not yet ready for issue. These "idle" cycles occur because shader cores execute threads in blocks. In our current simulation, the shader cores have to wait for all threads in a block to complete before moving onto a new block. The active thread count distribution reveals the reasons for DMaj's poor performance on Black-Scholes to be "Idle" and less than full warps cycles, which can be mostly eliminated by DPdPri. The data also suggest that it may be possible to further improve dynamic warp formation by exploring scheduling policies beyond those proposed in this article.

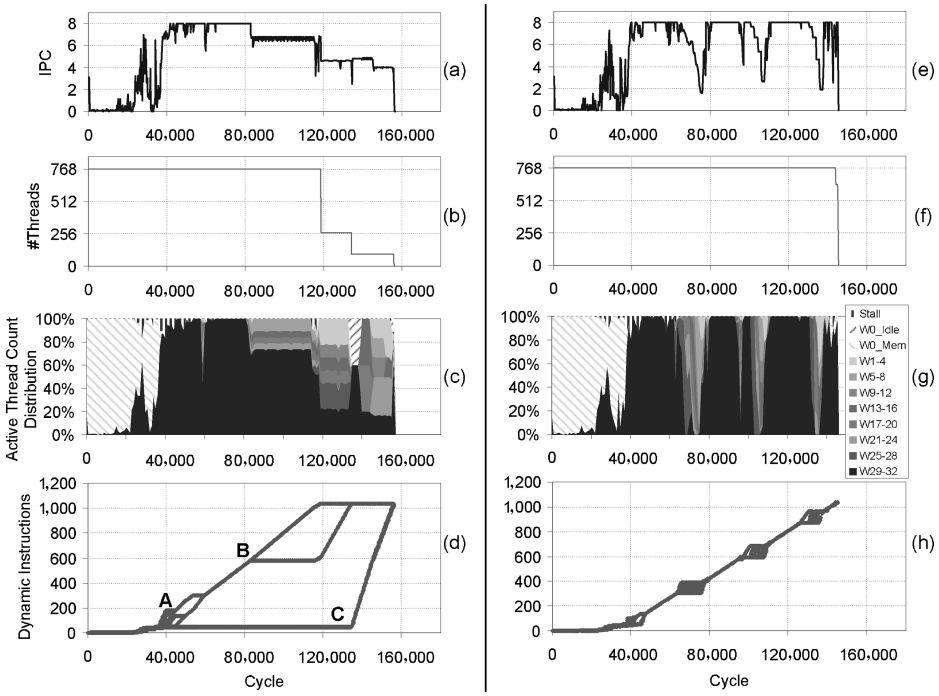


Fig. 16. Dynamic behavior of Black-Scholes using DWF with Majority scheduling policy (a–d) and PDOM Priority scheduling policy (e–h) in a single shader core (warp size of 32 issued over four cycles resulting in max IPC of 8). The active thread count distribution time series in (c) and (g) uses the same classification as in Figure 15.

6.2 Detailed Analysis of Majority Scheduling Policy Performance

The significant slowdown of Black-Scholes (Black) results from several phenomenon. First, in this work, we restrict a shader core to execute a single block. Second, we use software subroutines for transcendentals. Third, a quirk in our default Majority scheduling policy (DMaj) can lead some of the threads in a shader core to suffer from starvation. We explore this latter phenomenon in this section. Figure 16(a–d) shows the runtime behavior (IPC and incomplete thread count (#Thread)[14]) of a single shader core using dynamic warp formation with the Majority scheduling policy.

With DMaj, threads having different control-flow behavior from the rest of the threads can be starved during execution. Black-Scholes has several rarely executed, short basic blocks that suffer from this effect, leaving behind several groups of minority threads.[15] When these minority threads finally execute after the majority of threads have finished, they form incomplete warps and the number of warps formed are insufficient to fill up the pipeline (each thread

[14]A thread is counted as incomplete if it has not reached the end of the kernel call on the cycle in question.

[15]We say a thread is a minority thread if its PC value is consistently given a low priority by the Majority scheduling policy throughout its execution.

is only allowed to have one instruction executing in the pipeline, as described in Section 2.3). This behavior is illustrated in Figure 16(d), which shows the dynamic instruction count of each thread versus time. After several branches diverge at A and B, groups of threads are starved (indicated by the stagnant dynamic instruction count in Figure 16(d)), and only resume after C when the majority groups of threads have finished their execution (indicated by the lower thread count after cycle 120,000 in Figure 16(b)). This is responsible for the low IPC of DWF after cycle 120,000 in Figure 16(a).

Meanwhile, although the majority of threads are proceeding ahead after branch divergences at A and B, the pipeline is not completely utilized (indicated by the IPC<8 from cycle 80,000 to 120,000 in Figure 16(a)) due to the existence of incomplete warps (see the warp size distribution time series in Figure 16(c)). These incomplete warps are formed because the number of threads taking the same execution path is not a multiple of the SIMD width. They could have been combined with the minority threads after the divergence to minimize this performance penalty, but this does not happen when the minority threads are starved by DMaj.

Figure 16(e–h) shows how both of these problems can be mitigated by having a different scheduling policy—PDOM Priority (DPdPri). The dynamic instruction count in Figure 16(h) shows that any thread starvation due to divergence is quickly corrected by the policy, and all the threads have a similar rate of progress. The IPC stays at 8 for most of the execution (see Figure 16(e)), except when DPdPri is giving higher priorities to the incomplete warps inside the diverged execution paths (visible in Figure 16(g,h)) so that they can pass through the diverged execution paths quickly to form complete warps at the end of the diverged paths. Overall, threads in a block finish uniformly as shown in Figure 16(f), indicating that starvation does not happen with DPdPri.

To slowdown the threads that have executed further down a program (so that they can wait for threads left behind in hope of possible reconvergence), DPdPri gives higher priority to threads with lower counts of encountered postdominators. This is effective on benchmarks with simple control flow, such as Black-Scholes. With nested control flow, however, threads that have executed through more divergent branches will have encountered more postdominators and appear to the scheduler, as if they have progressed further down the program than they actually have. A simple example in which DPdPri is ineffective is a loop with an if-branch inside. Some threads may have exited the loop early and are first given lower priority as they hit the postdominator of the loop's backward-branch. Threads remaining in the loop would continually encounter the postdominator of the if-branch and eventually their postdominator count would exceed the early-exit threads. This gives the early-exit threads higher priority for execution, leaving behind the threads inside the loop.

This deficiency of DPdPri with complex control flow is exposed by HMMer. In Section 6.4, we evaluate the performance of DPdPri with lane aware scheduling (LAS) and show that the performance of HMMer is significantly lower than that of DMaj's (12 IPC versus 24 IPC for DMaj). This contributes to a lower average speedup of DPdPri over PDOM (31% versus DMaj's 47%). Despite this deficiency, DPdPri's effect on BlackScholes demonstrates the potential of using

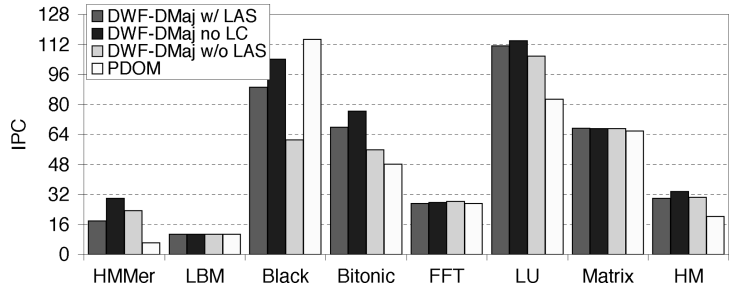


Fig. 17. Performance of dynamic warp formation evaluating the impact of lane aware scheduling and accounting for lane conflicts and scheduler implementation details using Majority scheduling policy. LAS=lane aware scheduling; LC=lane conflict.

static analysis to generate heuristics for improved scheduling. Further investigation of the use of static analysis to aid scheduling for dynamic warp formation is left as future work.

6.3 Effect of Limited Resources in Max-Heap

We have evaluated the performance increase achievable by dynamic warp formation with an infinite amount of hardware resources (infinitely ported and unlimited entries for MHeap LUT and Max-Heap) given to the DMaj scheduler. This unbounded version of DMaj has a speedup of 6.1% over its resource-limit counterpart, and is 56.3% faster than PDOM, on average. This speedup with no resource limit comes completely from HMMer (24 IPC versus 18 IPC with limited resources), which is both a control-flow and data-flow intensive benchmark. These properties of HMMer result in a large number of in-flight PC values among threads as a memory access within a warp following a branch divergence can introduce a variable slowdown among the threads in a warp ranging from tens to hundreds of cycles. The large number of in-flight PC values this causes in turn requires a large Max-Heap, which takes more swaps to rebalance every scheduler cycle and introduces stalls when the limited bandwidth resources are exhausted. However, with other benchmarks the resource-limited version of DMaj achieves similar performance.

6.4 Effect of Lane Aware Scheduling

To reduce the register file design complexity of DWF, we have chosen to use the organization in Figure 9(d), which necessitated the use of lane aware scheduling (LAS) discussed in Sections 4.1 and 4.2. The data in Figure 17 compares the detailed scheduler hardware model we have considered so far (DWF-DMaj w/ LAS) with an idealized version of dynamic warp formation ignoring the impact of lane conflict and hardware resources (DWF-DMaj no LC). This figure shows the impact on the performance of LAS and, for comparison, also shows the impact when not using LAS but assuming the register file organization in Figure 9(c) and modeling register bank conflicts when multiple threads from the same “home” lane are grouped into a single warp (DWF-DMaj w/o LAS). While the idealized version of DWF unconstrained by lane conflict (DWF-DMaj

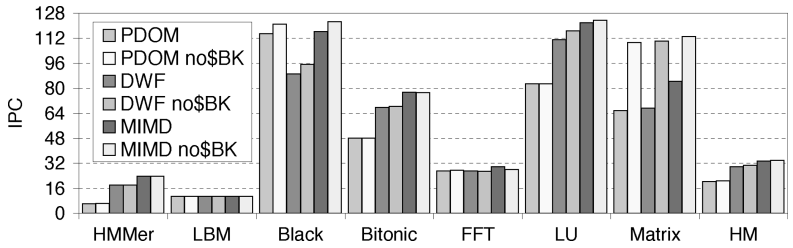


Fig. 18. Performance of PDOM, DWF, and MIMD with cache bank conflict. no$Bk=ignoring cache bank conflict.

no LC), which assumes hardware similar to Figure 9(b), is, on average, 66% faster than PDOM, the realistic implementation of DWF with LAS we have considered so far (DWF-DMaj w/ LAS) is able to achieve 89% of the idealized DWF's performance (DWF-DMaj no LC).

We also evaluated the performance of the PDOM Priority policy (DPdPri) with LAS and found performance for Black-Scholes improves (99 IPC versus 89 IPC for DMaj), while that of HMMer is reduced (12 IPC versus 24 IPC for DMaj) with an average speedup of 31% over PDOM. Overall, DMaj is the best scheduling policy for the realistic implementation by a significant margin (average of 47% improved with DMaj versus 31% with DPdPri).

6.5 Effect of Data Cache Bank Conflicts

Our baseline architecture model assumes a multiported data cache that is implemented using multiple banks of smaller caches. Parallel accesses from a warp to different lines in the same bank creates cache bank conflicts, which can only be resolved by serializing the accesses and stalling the SIMD pipeline.

With a multibanked data cache, a potential performance penalty for dynamic warp formation is that the process of grouping threads into new warps dynamically may introduce extra cache bank conflicts. We have already taken this performance penalty into account by modeling cache bank conflicts in our earlier simulations. However, to evaluate the effect of cache bank conflicts on different branch handling mechanisms, we rerun the simulations with an idealized cache allowing threads within a warp to access any arbitrary lines within the cache in parallel.

The data in Figure 18 compares the performance of PDOM, DWF, and MIMD ignoring cache bank conflicts. Cache bank conflicts have greater effect on benchmarks that are *less* constrained by the memory subsystem (Black-Scholes, LU, and Matrix). On average, performance of these benchmarks increases by a similar amount with an ideal multiported cache regardless of the branch handling mechanism in place. Overall, the data shows the benefit of DWF does not diminish due to additional cache bank conflicts introduced by regrouping threads into new warps.

6.6 Sensitivity to SIMD Warp Size

While DWF with the DMaj scheduling policy provides a significant speedup over PDOM with our default hardware configuration, we can gain further

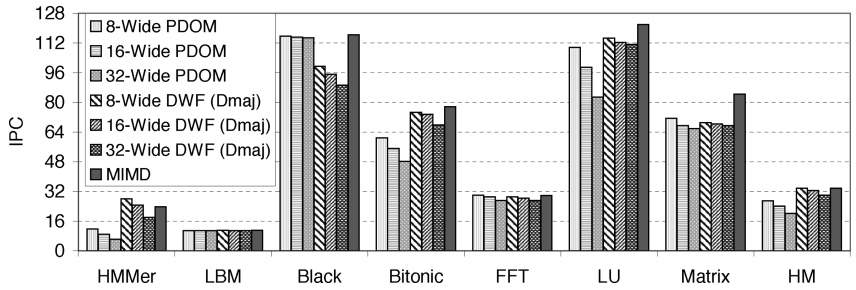


Fig. 19. Performance comparison of PDOM, DWF, and MIMD with a realistic memory subsystem as SIMD warp size increases. The theoretical peak IPC remains constant for all configurations.

intuition into its effectiveness in handling branch divergence as SIMD warp size increases. Figure 19 shows the performance of each mechanism for three configurations with increasing warp size from 8 to 32. Notice that the width of the SIMD pipeline is not changed, a decreased warp size only translates to a shorter execution latency for each warp. As described in Section 2.4, a warp with 32 threads takes 4 cycles to execute on a 8-wide SIMD pipeline, whereas a warp with 8 threads can be executed in a single cycle on the same pipeline.

None of the benchmarks benefit from SIMD warp size increases. This is expected as increasing warp size in an area constrained fashion, as described earlier, does not increase the peak throughput of the architecture while it constraints control-flow and thread-scheduling flexibility. The benefit of increasing SIMD warp size is to improve computational density by relaxing the scheduler's latency requirement, reducing the number of warps to schedule (simplifying scheduler hardware) and lowering instruction cache bandwidth requirements.

Overall, as SIMD warp size increases from 8 to 32, the average performance of PDOM decreases by 24.9%, while the overall performance of DWF decreases by 11.2%. Most of the slowdowns experienced by PDOM as SIMD warp size is increased are due to the control flow intensive benchmarks (HMMer, Bitonic, and LU), while performance is better with DWF. This trend shows that branch divergence becomes a more serious performance bottleneck in control-flow intensive applications as SIMD warp size is increased to improve computational density, but a significant portion of this performance loss can be regained using dynamic warp formation and scheduling.

Notice that the performance of 8-wide PDOM provides an upper bound on the potential gain of a variable rate scheduler for PDOM with a warp size that is a multiple of the SIMD pipeline width. Such a variable rate scheduler might try to skip cycles when no active threads are issued for faster reissue rate. However, this is still 10% slower than DWF with DMaj.

6.7 Sensitivity to Thread Pool Size

Our baseline architecture model assumes that each shader core can accommodate up to 768 threads (24 warps), and that the DWF scheduler is free to combine any of these threads into a new warps. However, resource limits, such as limited register capacity per shader core, may restrict the thread capacity

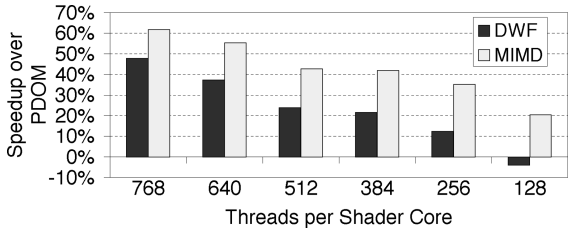


Fig. 20. speedup of DWF and MIMD over PDOM as number of threads per shader core decreases.

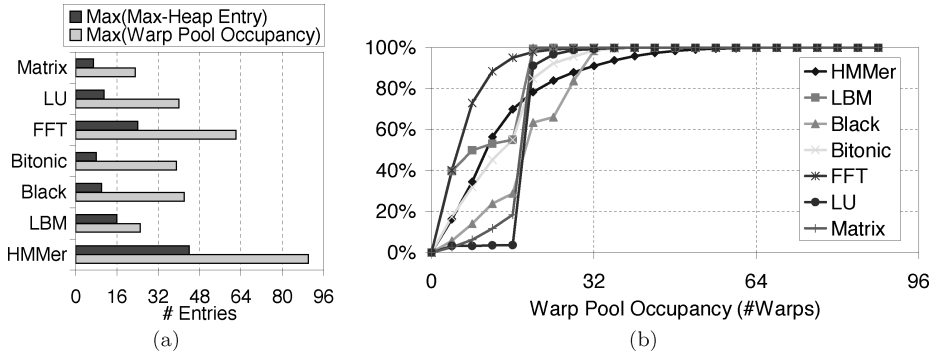


Fig. 21. Warp pool occupancy and Max-Heap size. (a) Maximum #entries. (b) Cumulative distribution of warp pool occupancy.

of each shader core to a fraction of its design limit. Under these circumstances, the DWF scheduler has fewer threads to recombine for new warps, which may lower the performance benefit of DWF.

Figure 20 shows the speedup of DWF (with DMaj and LAS) and MIMD over PDOM with decreasing thread pool size per shader core. While DWF is slower than PDOM by 4% with 128 threads, it shows a speedup of 13% at 256 threads. The speedup increases as the thread pool grows, eventually to 47% with 768 threads. Most of the degraded performance of DWF with fewer threads comes from HMMer (not shown). It is 2.95× faster than PDOM with 768 threads, while with 128 threads, it is 25% slower. The performance gap between DWF and PDOM for Black-Scholes widens with fewer threads (from 23% slower for DWF with 768 threads to 41% slower for DWF with 128 threads–not shown).

6.8 Warp Pool Occupancy and Max Heap Size

Figure 21(a) shows the maximum warp pool occupancy (the number of warps inside the warp pool) and the maximum dynamic size of the Max-Heap for each benchmark. As shown in the table, all of the benchmarks, including the control flow and memory intensive HMMer, use $< 1/6$ of a warp pool sized to the worst-case requirement. This suggests the possibility of using a warp pool with 128 entries (1/6 of the 768-entry warp pool designed for the absolute worst case) without causing any performance penalty. The same argument can be used to

argue that 64 entries is sufficient for the Max-Heap used in implementing the Majority scheduling policy.

Furthermore, Figure 21(b) shows the cumulative distribution of warp pool occupancy for each benchmark in this article. This distribution suggests that it may be feasible to shrink the warp pool to 64 entries, and handle $<0.15\%$ warp pool overflow (or 10% if we reduce the warp pool to 32 entries) by spilling warp pool entries to the global memory (as described in Section 4.5). As our simulator does not currently model this spilling mechanism, we assume structures having enough capacity to avoid spilling in the area estimation in Section 7.

7. AREA ESTIMATION

The total area cost for dynamic warp formation and scheduling is the amount of hardware added for the logic in Figure 10 plus the overhead needed for having an independent decoder for each register file bank. We have estimated the area of the five major parts of the hardware implementation of dynamic warp formation and scheduling with CACTI 4.2 [Tarjan et al. 2006]: warp update registers, PC-warp LUT, warp pool, warp allocator, and scheduling logic. Table V lists the storage sizing and implementation details of these structures used in our performance simulation in Section 6 along with their area estimates. We use our baseline configuration (32-wide SIMD with 768 threads) listed in Table I to estimate the size of the PC-warp LUT and MHeap LUT. Both are set-associative cache-like structures (4- and 8-way, respectively) with two read ports and two write ports capable of two warp lookups in parallel to handle requests from the two diverged parts of a single incoming warp. Based on the maximum occupancy data in Section 6.8 and Figure 21, we use a 128-entry warp pool and a 64-entry Max-Heap. A memory array with two read ports and two write ports is sufficient for the Max-Heap. Instead of being implemented with a multiported memory array, the warp pool can be separated into banks with only one write port, with each bank storing the TIDs of threads executing in the same SIMD lane. This works because every incoming warp (no matter how warps have diverged) has at most one thread would be written to each bank per cycle. The PC value and any scheduler specific data of each warp are stored in a separate bank from the TIDs with two write ports to support two warp creations in parallel. Overall, we have estimated the area of the dynamic warp scheduler in 90nm process technology to be 1.635mm^2 per core (last row, column on right in Table V).

To evaluate the overhead of having individual decoders per register file bank for dynamic warp formation and scheduling, as described in Section 4.1, we first need to estimate the size of the register file. The SRAM model of CACTI 4.2 [Tarjan et al. 2006] estimates a register file with 8,192 32-bit registers and a single decoder reading a row of eight 32-bit registers to be 3.8628mm^2 (we have adjusted the line size to 512 bytes for the minimum area). On the other hand, since the SRAM model of CACTI 4.2 does not directly estimate the area impact of banking, we first estimate the area of a single register file bank with 1024 32-bit registers. The overall register file is then estimated to require $0.573\text{mm}^2 \times 8 = 4.585\text{mm}^2$. Note that for the banked register file, we have divided the

Table V. Area Estimation for Dynamic Warp Formation and Scheduling. RP=read port, WP=write port

Structure	# Entries	Entry Content	Struct. Size (bits)	Implementation	Area (mm^2)
Warp Update Register	2	TID (10-bit) × 32 PC (24-bit), REQ (32-bit)	752	Registers (No Decoder)	0.0080
PC-Warp LUT	64	PC (24-bit) OCC (32-bit) IDX (7-bit)	4032	4-Way Set-Assoc. Mem. (2 RP, 2 WP)	0.2633
Warp Pool	128	TID (10-bit) × 32 PC (24-bit) Scheduler Data (10-bit)	44928	Memory Array (33 Banks) (1 RP, 1 WP)	0.6194
Warp Allocator	128	IDX (7-bit)	896	Memory Array	0.0342
Mheap LUT	128	PC (24-bit) IDX (7-bit)	4992	8-Way Set-Assoc. Mem. (2 RP, 2 WP)	0.4945
Max-Heap	64	PC (24-bit), CNT (10-bit) WPH (8-bit), WPT (8-bit) LUT-IDX (8-bit)	3712	Memory Array (2 RP, 2 WP)	0.2159
Total			59312		**1.6353**

512-byte line of the original single decoder design into eight 64-byte lines, one in each bank. Notice that both register file configurations have two read ports and one write port. This method may under-estimate the area requirement for adding a decoder to every register file bank, as it may not entirely capture all wiring complexity. However, the overhead we estimate in this way is already larger than the 11% overhead estimate by Jayasena et al. [2004] with CACTI and a custom floorplan for implementing a stream register file with indexed access. (Note: 11% of 3.863mm^2 is only 0.425mm^2, which is smaller than our 4.585mm^2−3.863mm^2 = 0.722mm^2 estimate.) Without an detailed layout of an actual GPU register file, the 0.722mm^2 overhead is our best estimate.

Combining the two estimates described earlier, the overall area consumption of dynamic warp formation with DMaj scheduling policy for each core is 2.357mm^2. With 16 cores per chip, as per our initial configuration, this becomes 37.72mm^2, which is 8% of the total area of the GeForce 8800GTX (470mm^2) [Lindholm et al. 2008].

8. LIMITATIONS OF DYNAMIC WARP FORMATION

This section discusses some challenges with adopting dynamic warp formation (DWF) in existing GPU architectures.

Synchronous Execution of Warps. While NVIDIA's CUDA programming model, which our compute model closely resembles, provides a MIMD thread model, there exist a small number of CUDA programs optimized with the assumption that the threads in a statically formed warp are executed synchronously. A example of such an optimization is shown in Figure 22, which illustrates the final steps of a parallel reduction algorithm [Harris 2007]. Each step needs to be executed synchronously, but the placement of barriers between each step can be avoided as the synchronization is done implicitly within a statically formed warp in current GPU microarchitectures.

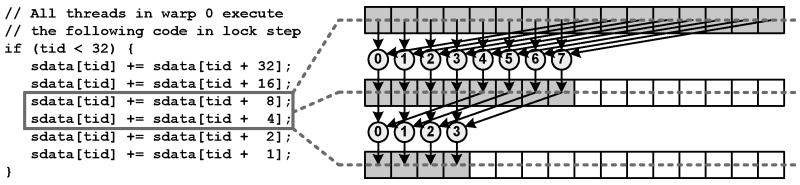


Fig. 22. Parallel reduction assuming synchronous execution of warps. Numbers inside the circles denotes the threads performing the reduction. Only two steps are illustrated here.

With DWF, threads in a statically formed warp may be dynamically reassigned to a different warps. This behavior breaks the implicit synchronization assumed by the optimization and this may render it functionally incorrect. One solution would be to require such behavior be expressed explicitly at the language level by expressing it using partial barriers between each line of code in the example (they are partial in the sense that only those threads in a warp participate in the barrier, rather than all threads in a block as in the `__syncthreads()` primitive in CUDA). Such partial barriers could potentially be optimized out by a compiler, depending upon the target architecture.

Memory Coalescing. A related optimization in CUDA programs is to arrange data access from threads within a warp so that they access contiguous data in memory. Memory coalescing is then achieved by combining these simultaneous memory accesses from multiple threads into fewer accesses [NVIDIA Corp. 2007a]. DWF may disrupt this coalesced access pattern by combining threads from different warps. This generates extra memory accesses, which in turn increases memory traffic. The performance impact from such extra memory traffic is modeled in the simulations we presented earlier (though we did not specifically optimize our benchmarks for memory coalescing).

9. RELATED WORK

Most approaches to supporting branches in a traditional SIMD machine have centered around the notion of guarded instructions [Bouknight et al. 1972]. Also known as a predicated or vector masked instructions, a guarded instruction's execution is dependent on a conditional mask controlled by another instruction. A predicated instruction with a conditional mask set to false will not update architecture state. A SIMD instruction with a vector of conditional masks, each controlled by an element in a stream, provides the functionality of a data dependent branch. This approach has been employed in existing GPU architectures to eliminate short branches and potential branch divergences [AMD, Inc. 2006; NVIDIA Corp. 2007a].

Guarded instructions and their variants put constraints on input-dependent loops. Branch divergence may be inevitable, but the period of divergence can be kept short with reconvergence to minimize performance lost due to unfilled SIMD pipelines. A patent filed by Lindholm et al. [2005] describes in detail how threads executing in a SIMD pipeline are serialized to avoid hazards, but does not indicate the use of reconvergence points to recover from such divergence. Levinthal and Porter [1984] described a SIMD branch handling menchanism using a stack of execution mask that is similar to our immediate post-dominator

branch reconvergence mechanism. Unlike our approach that uses control-flow analysis, they mapped high-level language control-flow constructs directly into actions to the stack. The notion of reconvergence based on control-flow analysis in SIMD branch handling was described in a patent by Lorie and Strong [1984]. However, they propose to insert the reconvergence point at the beginning of a branch and not at the immediate postdominator, as proposed in this article.

The notion of dynamically regrouping the scalar SPMD threads comprising a single SIMD task after control-flow divergence of the SPMD threads was described by Cervini [2005] in the context of simultaneous multithreading (SMT) on a general-purpose microprocessor that provides SIMD function units for exploiting subword parallelism. However, his proposal lacks a specific mechanism for grouping the diverged SPMD threads, whereas such an implementation is a main contribution of this article. Clark et al. [2007] introduce *Liquid SIMD* to improve SIMD binary compatibility by translating annotated scalar instructions into SIMD instructions at runtime with specialized hardware. The translated SIMD instructions still suffer performance loss from branch divergence when they execute on a traditional SIMD unit.

Kapasi et al. [2000] introduce *conditional streams*, a code transformation that seperates a single kernel with conditional code into multiple kernels and connects them via interkernel buffers to increase the utilization of a SIMD pipeline. A filter and a merger kernel are created for each diverging conditional branch to pack and unpack filtered data using an interprocessor switch. While more efficient than predication, this technique may be limited to architectures with software managed interkernel buffers, such as the stream register file in Imagine [Rixner et al. 1998] and Merrimac [Dally et al. 2003]. In comparison, dynamic warp formation is a hardware mechanism applicable to any multithreaded SIMD architecture, and its implementation does not require data movement between register lanes.

Krashinsky et al. [2004] propose the vector-thread architecture exposing two instruction fetch mechanism in the ISA: vector-fetch and thread-fetch. Vector-fetch is issued by a control processor to command all virtual processors (VPs) to fetch the same atomic instruction block (AIB) for execution, whereas thread-fetch can be issued by a VP to fetch an AIB to itself alone when execution diverges. This eliminates branch divergence, but instruction bandwidth can become a bottleneck if each VP is fetching a different AIB.

10. SUMMARY

In this article, we explore the impact of branch divergence on GPU performance for nongraphics applications. Without any mechanism to handle branch divergence, performance of a GPU's SIMD pipeline degrades significantly. While existing approaches to reconverging control flow at join points such as the immediate postdominator improve performance, we found significant performance improvements can be achieved with our proposed dynamic warp formation and scheduling mechanism. We described and evaluated an implementation of the hardware required for dynamic warp formation and tackled the challenge of enabling correct access to register data as thread warps are dynamically regrouped. We found that with 256 threads per shader core, dynamic warp

formation improves performance by 13%, on average, over a mechanism comparable to existing approaches—reconverging threads at the immediate postdominator. The improvement increases to 47% with 768 threads per shader core. Furthermore, we estimated the area of our proposed hardware changes to be around 8%.

Our experimental results highlight the importance of careful prioritization of warps for scheduling in such massively parallel hardware, even when individual scalar threads are executing the same code in the same program phase. Thus, we believe there is room for future work in this area.

ACKNOWLEDGMENTS

We thank the associate editor in charge and the anonymous referees for their helpful comments. We also thank Henry Tran for contributions to our simulation infrastructure.

REFERENCES

AGARWAL, V., HRISHIKESH, M. S., KECKLER, S. W., AND BURGER, D. 2000. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*. ACM, 248–259.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Upper Saddle River, NJ.

ALLEN, J. R., KENNEDY, K., PORTERFIELD, C., AND WARREN, J. 1983. Conversion of control dependence to data dependence. In *Proceedings of the 10th Symposium on Principles of Programming Languages (POPL '83)*. ACM, 177–189.

AMD, INC. 2006. *ATI CTM Guide*, 1.01 ed. AMD, Inc.

BASU, A., KIRMAN, N., KIRMAN, M., CHAUDHURI, M., AND MARTINEZ, J. 2007. Scavenger: A new last level cache architecture with global block priority. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO'07)*. ACM, 421–432.

BLINN, J. F. 1978. Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph. 12,* 3, 286–292.

BOUKNIGHT, W., DENENBERG, S., MCINTYRE, D., RANDALL, J., SAMEH, A., AND SLOTNICK, D. 1972. The Illiac IV System. *Proc. IEEE 60,* 4, 369–388.

BUATOIS, L., CAUMON, G., AND LÉVY, B. 2008. Concurrent number cruncher: A GPU implementation of a general sparse linear solver. *Int. J. Parall. Emerge. Distrib. Syst*.

BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for GPUs: Stream computing on graphics hardware. In *Proceeding of the Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'04)*. ACM, 777–786.

BURGER, D. AND AUSTIN, T. M. 1997. The SimpleScalar Tool Set, Version 2.0. http://www.simplescalar.com.

CERVINI, S. 2005. European Patent EP 1531391 A2: System and method for efficiently executing single program multiple data (SPMD) programs.

CLARK, N., HORMATI, A., YEHIA, S., MAHLKE, S., AND FLAUTNER, K. 2007. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. IEEE, 216–227.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms* 2nd Ed. MIT Press, Cambridge, MA.

CROW, F. C. 1977. Shadow algorithms for computer graphics. *SIGGRAPH Comput. Graph. 11,* 2, 242–248.

DALLY, W. J., LABONTE, F., DAS, A., HANRAHAN, P., AHN, J.-H., GUMMARAJU, J., EREZ, M., JAYASENA, N., BUCK, I., KNIGHT, T. J., AND KAPASI, U. J. 2003. Merrimac: Supercomputing with streams. In *Proceedings of Supercomputing*. IEEE.

DALLY, W. J. AND TOWLES, B. 2004. *Interconnection Networks*. Morgan Kaufmann, San Francisco, CA.

DEL BARRIO, V., GONZALEZ, C., ROCA, J., FERNANDEZ, A., AND ESPASA, R. 2006. ATTILA: A cycle-level execution-driven simulator for modern GPU architectures. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software.* IEEE, 231–241.

FOWLER, H., FOWLER, F., AND THOMPSON, D., EDS. 1995. *The Concise Oxford Dictionary* 9th Ed. Oxford University Press.

FUNG, W. W. L., SHAM, I., YUAN, G., AND AAMODT, T. M. 2007. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO'07)*. ACM, 407–420.

HAKURA, Z. S. AND GUPTA, A. 1997. The design and analysis of a cache architecture for texture mapping. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97).* ACM, *25*, 2, 108–120.

HARRIS, M. 2007. Optimizing parallel reduction in cuda. http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf.

HWU, W.-M., KIRK, D., RYOO, S., RODRIGUES, C., STRATTON, J., AND HUANG, K. 2007. Performance insights on executing non-graphics applications on CUDA on the NVIDIA GeForce 8800 GTX. http://www.hotchips.org/archives/hc19/2 Mon/HC19.02/HC19.02.03.pdf.

INTEL CORP. 2008. Intel 965 Express Chipset Family and Intel G35 Express Chipset Graphics Controller Programmer's Reference Manual. Intel Corporation.

IOANNOU, A. AND KATEVENIS, M. G. H. 2007. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Trans. Netw. 15,* 2, 450–461.

JAYASENA, N., EREZ, M., AHN, J. H., AND DALLY, W. J. 2004. Stream register files with indexed access. In *Proceedings of the 10th International Symposium on High-Performance Computer Architecutre (HPCA'04).* IEEE, 60–71.

KAPASI, U. J., DALLY, W. J., RIXNER, S., MATTSON, P. R., OWENS, J. D., AND KHAILANY, B. 2000. Efficient conditional operations for data-parallel architectures. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Micro-architecture (MICRO'33)*. ACM, 159–170.

KRASHINSKY, R., BATTEN, C., HAMPTON, M., GERDING, S., PHARRIS, B., CASPER, J., AND ASANOVIC, K. 2004. The vector-thread architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04).* ACM, 52–63.

LEVINTHAL, A. AND PORTER, T. 1984. Chap: A SIMD graphics processor. In *Proceedings of SIGGRAPH*. 77–82.

LINDHOLM, E., KLIGARD, M. J., AND MORETON, H. P. 2001. A user-programmable vertex engine. In *Proceeding of the Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'01)*. ACM, 149–158.

LINDHOLM, E., NICKOLLS, J., OBERMAN, S., AND MONTRYM, J. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *Micro IEEE 28,* 2, 39–55.

LORIE, R. A. AND STRONG, H. R. 1984. US Patent 4,435,758: Method for conditional branch execution in SIMD vector processors.

LUEBKE, D. AND HUMPHREYS, G. 2007. How GPUs work. *Computer 40,* 2, 96–100.

MONTRYM, J. AND MORETON, H. 2005. The GeForce 6800. *IEEE Micro 25,* 2, 41–51.

MOY, S. AND LINDHOLM, E. 2005. US Patent 6,947,047: Method and system for programmable pipelined graphics processing with branching instructions.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmanns.

NEEDLEMAN, S. B. AND WUNSCH, C. D. 1970. A general method applicable to the search for similarities in the amino acid sequences of tow proteins. *Mol. Biol.* 48, 443–453.

NVIDIA CORP. CUDA SDK code samples. http://www.nvidia.com/object/cuda get samples.html.

NVIDIA CORP. 2007a. *NVIDIA CUDA Programming Guide*, 1.1 ed. NVIDIA Corp.

NVIDIA CORP. 2007b. *PTX: Parallel Thread Execution ISA*, 1.1 ed. NVIDIA Corp.

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. In *Proceeding of the Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'02)*. ACM, 703–712.

RIXNER, S., DALLY, W. J., KAPASI, U. J., KHAILANY, B., LÓPEZ-LAGUNAS, A., MATTSON, P. R., AND OWENS, J. D. 1998. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st International Symposium on Micro-architecture (MICRO'98)*. ACM, 3–13.

RIXNER, S., DALLY, W. J., KAPASI, U. J., MATTSON, P., AND OWENS, J. D. 2000. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*. ACM, 128–138.

ROTENBERG, E., JACOBSON, Q., AND SMITH, J. E. 1999. A study of control independence in superscalar processors. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecutre (HPCA'99)*. IEEE, 115–124.

SHEAFFER, J. W., LUEBKE, D., AND SKADRON, K. 2004. A flexible simulation framework for graphics architectures. In *Proceedings of the ACM SIGGRAPH / EUROGRAPHICS Conference on Graphics Hardware (HWWS'04)*. ACM, 85–94.

SHEBANOW, M. 2007. ECE 498 AL: Programming massively parallel processors (lecture 12). http://courses.ece.uiuc.edu/ece498/al1/Archive/Spring2007.

SHIN, J., HALL, M., AND CHAME, J. 2007. Introducing control flow into vectorized code. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 280–291.

STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC CPU2006 benchmarks. http://www.spec.org/cpu2006/.

TARJAN, D., THOZIYOR, S., AND JOUPPI, N. P. 2006. CACTI 4.0. Tech. rep. HPL-2006-86, Hewlett Packard Laboratories, Palo Alto, CA.

THISTLE, M. R. AND SMITH, B. J. 1988. A processor architecture for Horizon. In *Proceedings of Super-computing*. IEEE, 35–41.

THORNTON, J. E. 1964. Parallel operation in the control data 6600. In *AFIPS Proceedings of FJCC*. Vol. 26. 33–40.

UPSTILL, S. 1990. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley, Reading, MA.

WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. 1995. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*. ACM, 24–36.

WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. 2005. RPU: a programmable ray processing unit for real-time ray tracing. *ACM Trans. Graph. 24,* 3, 434–444.

Received April 2008; revised November 2008; accepted December 2008

Simultaneous Branch and Warp Interweaving for Sustained GPU Performance

Nicolas Brunie
Kalray and ENS de Lyon
nicolas.brunie@kalray.eu

Caroline Collange
Universidade Federal de Minas Gerais
caroline.collange@inria.fr

Gregory Diamos
NVIDIA Research
gdiamos@nvidia.com

Abstract

Single-Instruction Multiple-Thread (SIMT) micro-architectures implemented in Graphics Processing Units (GPUs) run fine-grained threads in lockstep by grouping them into units, referred to as warps, to amortize the cost of instruction fetch, decode and control logic over multiple execution units. As individual threads take divergent execution paths, their processing takes place sequentially, defeating part of the efficiency advantage of SIMD execution. We present two complementary techniques that mitigate the impact of thread divergence on SIMT micro-architectures. Both techniques relax the SIMD execution model by allowing two distinct instructions to be scheduled to disjoint subsets of the the same row of execution units, instead of one single instruction. They increase flexibility by providing more thread grouping opportunities than SIMD, while preserving the affinity between threads to avoid introducing extra memory divergence. We consider (1) co-issuing instructions from different divergent paths of the same warp and (2) co-issuing instructions from different warps. To support (1), we introduce a novel thread reconvergence technique that ensures threads are run back in lockstep at control-flow reconvergence points without hindering their ability to run branches in parallel. We propose a lane shuffling technique to allow solution (2) to benefit from inter-warp correlations in divergence patterns. The combination of all these techniques improves performance by 23% on a set of regular GPGPU applications and by 40% on irregular applications, while maintaining the same instruction-fetch and processing-unit resource requirements as the contemporary Fermi GPU architecture.

1. Introduction

Graphics Processing Unit (GPU) architectures have proven successful for general-purpose parallel computing in various applicative areas [27]. GPUs group threads into packets, referred to as warps, and run them in lockstep on SIMD units. Architectures based on SIMD execution offer a potential efficiency advantage by sharing a common pipeline front-end across multiple execution units. However, this execution model only benefits applications whose control flow patterns and memory access patterns present enough regularity. Prior work has shown that the performance potential of GPUs is vastly underexploited by many irregular applications [21, 15, 8].

Our goal is to broaden the applicability of GPU architectures by substantially improving their performance on irregular applications, while maintaining their performance on regular applications. To achieve this goal, we introduce two complementary techniques that tackle the divergence problem. They both reclaim SIMD lanes that would have been left idle due to divergence to run instructions of other threads. The first technique, that we name Simultaneous Branch Interweaving (SBI), leverages the inherent parallelism available between divergent branches to co-issue instructions from the same warp. The second technique, Simultaneous Warp Interweaving (SWI), selects instructions from other partial warps to fill the gaps left by divergence.

The idea behind SBI and SWI comes from realizing that the benefits of SIMD execution arise from amortizing the cost of instruction fetch units, instruction decode units and control logic across many execution units. The two micro-architectures we propose maintain the

978-1-4673-0476-4/12/$31.00(c)2012 IEEE

same ratio of instruction control units to execution units as some existing GPU architectures, but relax the constraints of SIMD execution to better tolerate thread divergence. Like Simultaneous Multi-Threading (SMT) on superscalar processors [31], SBI and SWI increase functional unit utilization by filling scheduling bubbles thanks to fine-grained resource allocation to multiple threads. Together, they improve performance by 40% on a set of irregular applications, and also accelerate regular applications by 23%.

We will first examine existing GPU micro-architectures and define our baseline model in section 2. We will then consider the SBI technique that simultaneously co-issues instructions from multiple branches, with a focus on ways to achieve thread reconvergence in section 3. The second technique, SWI, will be described in section 4. Finally, we will quantify the performance and overhead of the proposed solutions in section 5 and review related work in section 6.

2. Background

Current-generation GPUs operate according to an implicit SIMD execution model, also called SIMT (Single Instruction, Multiple Threads) by NVIDIA. From the programmer's and compiler's point of view, this model is similar to SPMD (Single Program, Multiple Data). The programmer writes a single program or *kernel*. A large number of instances of the kernel (or *threads*) are then run in parallel. During execution on a GPU, transparent hardware mechanisms group threads into warps, and execute their instructions in lockstep on SIMD units [27]. The execution order of threads is undefined by the programming model unless the programmer uses explicit synchronization primitives.

Each thread may logically follow a differentiated control flow path, even though it physically runs in lockstep with other threads of the warp. To maintain the illusion of differentiated control flow, most GPUs use implicit instruction predication. The context associated with future branches (PC and mask) are stored in a hardware stack [22]. Entries are popped from the stack as control flow reconverges. Reconvergence points are exposed at the architectural level, and are inserted by the compiler.

For scalability reasons, GPUs are made of several independent warp processors, that we name Streaming Multiprocessors (SMs). Multiple warps are maintained in flight in each SM, and their execution is interleaved at the granularity of an instruction for latency tolerance purposes. Recent GPU architectures such as NVIDIA Fermi have several clusters per SM, in a way that is reminiscent of clustered multi-threaded architectures [11, 19]. Each cluster has its own instruction fetch unit and scheduling logic [27]. This enables the sharing of instruction caches, data caches and lesser-used functional units at a coarse granularity, while keeping the warp width small enough to tolerate thread divergence.

Baseline We consider in this paper a baseline SM micro-architecture closely inspired by Fermi, outlined in Figure 1. Like Fermi, it handles branch divergence using a hardware stack. Warps are split into two warp

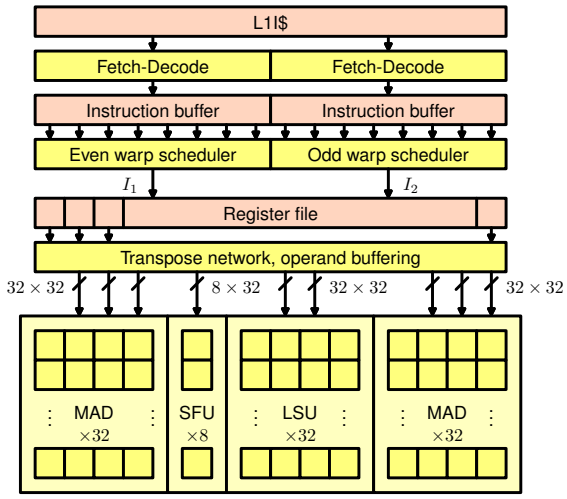


Figure 1. Baseline SM micro-architecture.

pools, respectively made of warps with even identifiers and warps with odd identifiers. Each pool has dedicated independent scheduling resources. Each clock cycle, one instruction from each pool is fetched from the cache and fed into an instruction buffer comprising one entry per warp. Two instruction schedulers pick one ready instruction each from their associated instruction buffer and issue it. Each instruction scheduler selects its oldest instruction [25]. Dependencies between instructions are tracked using a scoreboard. It consists of a table indexed by warp ID, where each entry contains the destination register IDs of the instructions in flight for the warp [7]. When a new instruction is decoded, its source and destination register IDs are compared against the scoreboard entries for its warp. The dependency bit vector that is produced by comparisons is kept in the instruction buffer alongside the decoded instruction. When an instruction is retired, both the scoreboard and the instruction buffer entry are updated to mark the dependency as cleared.

Instructions whose dependency mask reach zero are eligible for scheduling.

The pipeline back-end consists of four groups of SIMD units. To balance throughput and hardware cost, the SIMD width may be lower than the warp width. In this case, the warp is broken down into several waves sent through the pipeline. Operand buffers perform throughput matching between the register file and execution units. The Multiply-Add (MAD) units and Special Function Units (SFU) respectively execute most arithmetic instructions and transcendental functions as in NVIDIA's architectures [27]. The Load-Store Unit (LSU) arbitrates access to a single 128-byte port to the L1 cache. It can coalesce together multiple parallel accesses that fall within the same 128-byte cache block. Memory instructions that encounter conflicts are replayed with an updated activity mask reflecting the transactions that remain to be issued.

We contrast the techniques proposed throughout this paper with the baseline SIMT execution model on a simple example on figure 2. It depicts the execution pipeline when an *if-then-else* block with 2 warps of 4 threads is run. Each instruction is identified by its address, from 1 to 6. The *if* branch contains instructions 2 to 4, and the *else* branch contains instruction 5. SBI improves over the baseline SIMT by simultaneously scheduling instructions from different branches. In figure 2(b), a secondary scheduler issues instruction 5 of the *else* path together with instruction 2. Optional reconvergence constraints can be applied to SBI to re-align threads when control flow reconverges, as with instruction 6 of figure 2(c). SWI issues instructions from other warps, such as instruction 2 of warp 2 together with instruction 3 of warp 1 in figure 2(d). Finally, both techniques can be combined, is in figure 2(e).

3. Simultaneous branch interweaving

In this section, we focus on extracting parallelism from divergent branches of the same warp. Whether a branch is divergent or not is generally only known at runtime: hence, our approach is based on dynamic scheduling. When control flow splits into two branches, current SIMT architectures execute each branch sequentially. However, branch paths are taken by disjoint sets of threads. They are independent from each other and can be executed in any order, including in parallel, without violating the SIMT programming model.

3.1. Enabling parallel branch execution

Decoupling the execution of concurrent branch paths requires additional hardware support beyond conventional stack-based reconvergence. Dynamic Warp Subdivision was proposed to enable the shared instruction scheduler to interleave the execution of instructions from each path [24]. This technique improves memory latency hiding by providing more memory-level parallelism. Instruction throughput is not affected or may decrease, as the execution of individual instructions happens sequentially as in SIMT. Warps are decomposed into multiple *warp-splits*, which correspond to different paths concurrently taken. Each warp-split has its own divergence stack and is scheduled independently from the others. Reconvergence is made possible using a dedicated hardware table and/or by comparing the Program Counters (PCs) of each warp-split. However, current stack-based reconvergence systems do not allow several branches to be executed in parallel. The stack-based algorithm has to be complemented by heuristics, which provide no guarantee on when reconvergence will occur.

To work around these issues, we advocate to switch from stack-based reconvergence to thread-frontier based reconvergence [10]. Diamos et al. have shown that PC-based reconvergence can be made optimal in the sense that it always reconverge at the earliest possible point, given hardware support for a sorted heap and compiler support for laying out the code in the order dictated by thread frontier analysis [10]. It works by always scheduling the warp-split of minimal PC[1]. Like stack-based reconvergence, thread frontier reconvergence runs divergent branches sequentially. However, we show it is amenable to parallel execution by relaxing the scheduling constraints.

We refer to the Common PCs as CPCs to avoid ambiguities. In addition to the first minimum of thread PCs ($\mathrm{CPC}_1 = \min(\mathrm{PC}_i)$), we also compute the second minimum when it exists ($\mathrm{CPC}_2 = \min(\mathrm{PC}_i, \mathrm{PC}_i \neq \mathrm{CPC}_1)$). As a consequence, we have the order $\mathrm{CPC}_1 < \mathrm{CPC}_2 < \mathrm{PC}_i, \mathrm{PC}_i \notin \{\mathrm{CPC}_1, \mathrm{CPC}_2\}$. We will refer to the warp-splits whose PC is CPC_1 and CPC_2 as the primary and the secondary warp-split, respectively. We will assume throughout the next section that both minimums are available at all times. Concrete hardware implementations and their tradeoffs will be described in section 3.4.

[1] We assume in this work that thread-frontier priorities are implicitly encoded in the program order.

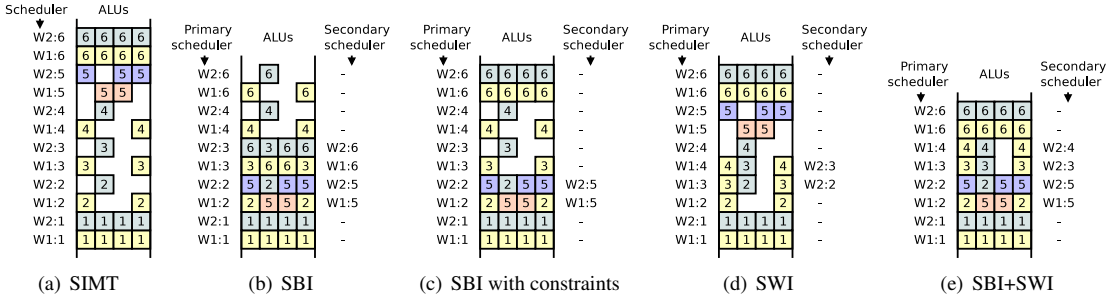


Figure 2. Comparison of the contents of the execution pipeline using classic SIMT, Simultaneous Branch Interleaving with optional contraints, Simultaneous Warp Interleaving, and both.

3.2. SBI architecture

SBI can be implemented with minimal modifications to our baseline architecture if the warp size is doubled to 64 threads. Instead of two warp pools of 32-wide warps, a single warp pool contains entries for two 64-wide warp-splits. The left-hand front-end depicted on figure 3 is essentially unchanged. It schedules instruction I_1 following CPC_1. The second front-end issues instruction I_2 following CPC_2 of the same warp. Instruction delivery is extended to broadcast both instruction I_1 and I_2 to all processing lanes. Per-lane multiplexing selects which instruction to execute according to individual bits in the warp-split predication masks m_1 and m_2.

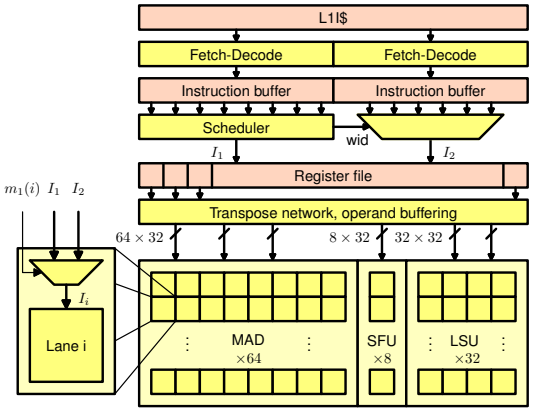


Figure 3. Simultaneous Branch Interweaving micro-architecture.

3.3. Ensuring reconvergence

Warp-split desynchronization Greedy scheduling may delay reconvergence by continuously letting a secondary warp-split run ahead of the primary warp-split, as in figure 2(b). In this example, the secondary scheduler issues instruction 6 early for threads 2 and 3 of warp 1. Several cycles later, the primary scheduler then issues the same instruction for threads 1 and 4. As long as the secondary warp-split does not encounter resource conflicts, warp-splits can continue running out-of-sync from each other. This desynchronization can lead to power consumption increase through redundant instruction fetch, and conflicts for memory resources [20].

Selective synchronization barrier We consider in this section a conservative policy, which prevents parallel execution of divergent branches to happen past the reconvergence point. We emit a synchronization instruction at each reconvergence point. Its payload is the address $\mathrm{PC}_{\mathrm{div}}$ of the divergence point, which we define as the last instruction of the immediate dominator. The thread-frontier aware program layout ensures that each reconvergence point occurs at a higher address in the code than the divergence point [10]. Synchronization instructions are treated as selective synchronization barriers among warp-splits. The secondary warp-split is suspended if $\mathrm{CPC}_1 \in [\mathrm{PC}_{\mathrm{div}}, \mathrm{PC}_{\mathrm{rec}}]$, and can continue otherwise, where $\mathrm{PC}_{\mathrm{rec}}$ is the address of the reconvergence point.

The idea behind selective synchronization is to allow parallel execution across outer branches of nested control-flow blocks, while still ensuring synchronization at the end of inner blocks. We illustrate this idea on the control-flow graph of figure 4, which consists of two nested *if-then-else* blocks. Basic blocks are labeled from

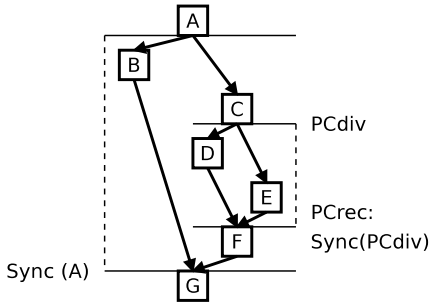


Figure 4. Control flow graph laid out by PC order, with divergence and reconvergence point annotations.

A to G. The vertical axis denotes the program-counter order. We suppose the secondary warp-split has just reached the F block, so that $\mathrm{CPC}_2 = \mathrm{PC}_{\mathrm{rec}}$. It encounters a synchronization instruction that points at $\mathrm{PC}_{\mathrm{div}}$, the last instruction of block C. We distinguish two cases.

1. $\mathrm{PC}_{\mathrm{div}} < \mathrm{CPC}_1 < \mathrm{PC}_{\mathrm{rec}}$. The secondary warp-split is suspended until the primary warp-split, which is currently in D or E, reaches the reconvergence point at the beginning of the F block.

2. $\mathrm{CPC}_1 \leq \mathrm{PC}_{\mathrm{div}}$. As CPCs are strictly ordered, no other warp-split has its PC between $\mathrm{PC}_{\mathrm{div}}$ and $\mathrm{PC}_{\mathrm{rec}}$. Synchronization for the inner *if-then-else* block is achieved, and the execution of the secondary warp-split can continue through F. This way, blocks B and F are free to run in parallel, which does not delay reconvergence.

Though we illustrated this technique on an *if-then-else* nest, it can be applied to other control structures such as loop nests as well. Unstructured control flow may involve multiple divergence points for a single reconvergence point. Our choice of the immediate dominator is conservative as it may place the divergence point earlier than strictly necessary. While it means that opportunities for parallel execution may be missed, it guarantees that synchronization will occur at the reconvergence point.

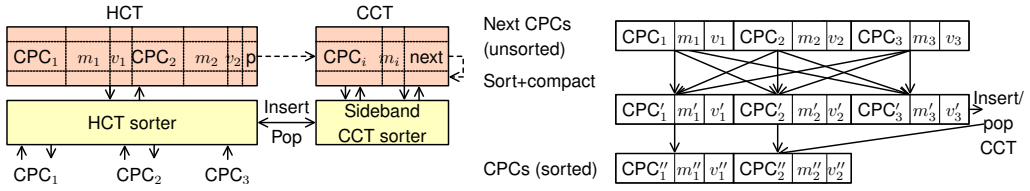
No additional hardware is needed to release warp-splits from their wait-state. As the primary warp-split reaches the synchronization point, its CPC_1 matches the CPC_2 of the secondary warp-split, causing both warp-splits to be merged together as the new primary warp-split.

3.4. Implementation

Sorted heap We use a sorted-heap based implementation as proposed by Fung et al. [15] and Diamos et al. [10] to store warp-splits. Each warp-split context is an entry (CPC, m, v) containing the program counter, the activity mask and a valid bit. Contexts are arranged into a Hot Context Table (HCT) and a Cold Context Table (CCT), as outlined in figure 5(a). The HCT is indexed by the warp identifier and contains the two active contexts of each warp, as well as a pointer to the next inactive context in the CCT. The CCT is organized as a linked list of contexts per warp, where each entry has a pointer to the next element. Another linked list contains free blocks. All lists share the same CCT.

Each table is managed by a separate unit. The HCT sorter unit is responsible for keeping the first two entries of each warp sorted, and merge them as needed. It receives the updated CPC_1 and CPC_2 from the PC update logic. When divergence occurs, it also receives an additional CPC_3 from the branch and memory arbitration logic. We enforce the restriction that at most one divergence (branch or memory) can happen each cycle. It guarantees that at most one new warp-split can be created each cycle. The HCT sorter consists of a sorting network that can compact and merge entries (fig. 5(b)). During the first stage, entries $(\mathrm{CPC}_1, \mathrm{CPC}_2, \mathrm{CPC}_3)$ are sorted, compacted and merged into $\mathrm{CPC}'_1 < \mathrm{CPC}'_2 < \mathrm{CPC}'_3$, and their valid bits are updated. If three valid entries remain after compaction ($v'_3 = 1$), then the third entry is inserted into the CCT. If there is one valid entry only ($v'_2 = 0$), the second entry is popped from the CCT.

Likewise, the CCT is kept sorted by a dedicated unit that also handles insertions. While the HCT runs synchronously with the pipeline, insertions in the CCT are asynchronous. The sideband sorter is a state machine that walks linked lists in the CCT to perform an insertion sort. In the worst case, an insertion can take 64 non-pipelined cycles. However, warp-split order in the heap does not affect execution correctness ; it is only an optimization that avoids unnecessary thread divergence. In case the sideband sorter is unable to keep up with insertions of new warp-split, the sorted heap will be degraded into a stack, where popping the top entry will just return the last inserted entry. Interestingly, this degraded mode matches the behavior of divergence stacks used on today's GPUs. Prior work has also shown that the heap size including hot entries rarely exceed 3 in real programs, so even an $\mathrm{O}(n^2)$ sorting algorithm is adequate [10].



(a) General architecture (b) HCT sorter

Figure 5. Dual context table organization

Scoreboard As warp-splits diverge and reconverge, individual threads may "jump" from one warp-split to the other, creating data dependencies between instructions. Conversely, register read/write dependencies between non-intersecting warp-splits should be ignored. The brute-force approach to dependency tracking would store the execution mask of each instruction in the pipeline alongside its scoreboard entry. To reduce storage requirements, we compute instead dependencies between instructions by taking the transitive closure of the warp-split divergence-convergence graph.

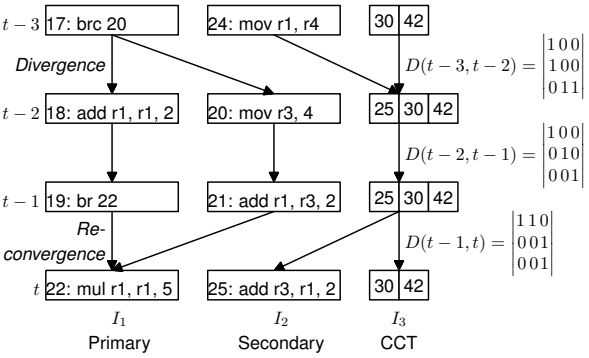


Figure 6. Divergence-convergence graph and dependency matrices for an example instruction sequence.

We define the dependency matrices $D(t_1, t_2)$ between two scheduling cycles t_1 and t_2 such that $D_{i,j}(t_1, t_2)$ is set when common threads execute $I_i(t_1)$ and $I_j(t_2)$. I_3 wraps all inactive entries in the heap. Figure 6 illustrates the dependency matrices on a divergence-convergence graph example. Along with the destination registers of instructions I_1 and I_2, each scoreboard entry k contains the dependency matrix $D(t-k, t)$ with the instructions waiting to be issued. When new instructions are scheduled, dependency matrices are multiplied by the dependency matrix $D(t, t+1)$ of the instruction just issued and shifted by one position. Dependency bits are ANDed together with the result of the register ID comparison to form the dependency mask stored in the instruction buffer. The complexity of the proposed scoreboard only depends on the warp count, pipeline depth and instruction issue width. It is not affected by the warp size.

4. Simultaneous warp interweaving

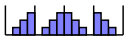
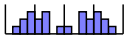
SBI improves throughput when branch paths are balanced, but provides no benefit when the workload of each thread of a warp is unbalanced, as with *if* blocks with no *else* counterparts. We consider in this section Simultaneous Warp Interweaving, the counterpart of SBI that schedules other warps in the gaps left by the first scheduled warp. It is described in isolation with SBI in this section, although both techniques can be combined.

Our asymmetric SM design is based on two cascaded schedulers. As with SBI, the primary scheduler selects a ready instruction I_1 from the instruction buffer. The issue of I_1 is then delayed during one pipeline stage dedicated to secondary scheduling. The secondary scheduler receives the predicate mask m_1 from the initial instruction and looks for a non-conflicting instruction I_2. The secondary instruction may be scheduled to the same SIMD group as the primary instruction, as long as its predicate mask m_2 does not overlap with the primary mask m_1. Alternatively, it can be scheduled to another free SIMD group, as in a conventional multiple-issue SIMD processor. Both instructions I_1 and I_2 are then issued simultaneously to the execution units. The main idea consists in trading scheduler latency for execution unit throughput. The front-end latency increases due to cascaded scheduling. On the other hand, the utilization rate of the back-end execution units benefits from the additional scheduling opportunities created.

Lane shuffling Many parallel algorithms exhibit regular thread imbalance patterns inside each warp. For instance, the first thread of each warp may receive a larger

share of work than its neighbors. Such correlation increases the likeliness of conflicts for execution lanes. To turn the otherwise negative effect of correlations to our advantage, we considered several static thread rearrangement heuristics, that we list on table 1. The physical lane id is computed from the thread-in-warp ID tid and warp ID wid. $\oplus$ is the XOR operator and bitrev is the bit-reversal function. The diagrams on the right illustrate the effect on 4 warps of 4 threads each by plotting the lane ID as a function of $4 \times wid + tid$. Inside each warp, we apply a permutation to the thread-to-lane mapping. At this point, this amounts to changing the way their thread identifier is computed. It requires no additional hardware nor data migration. The mapping

Table 1. Lane shuffle functions.

Name	Function	Illustration
Identity	tid	
MirrorOdd	$n - tid$ if wid odd, tid otherwise	
MirrorHalf	$n - tid$ if $wid > m/2$, tid otherwise	
Xor	$tid \oplus wid$	
XorRev	$tid \oplus \mathrm{bitrev}(wid)$	

functions preserve memory locality and allow the same memory coalescing opportunities as the straightforward mapping. We found that *XorRev* provides the most consistent performance gain across the applications considered (described in section 5).

Scheduler conflict avoidance As both schedulers operate in parallel, the secondary scheduler on phase n may pick the same instructions as the primary scheduler on phase $n + 1$. Rather than prevent this situation by introducing tighter coupling between the two schedulers that would increase their complexity, we detect conflicts *a posteriori*. After a conflict is identified, the instruction copy selected by the primary warp is discarded, and its execution mask set to empty. The other copy is issued on phase n as the secondary warp. During the next cycle, the secondary scheduler is free to select any ready instruction, substituting itself to the primary scheduler.

As conflicts still restrict the scheduling choice and decrease the overall power-efficiency, we ensure they remain unlikely by avoiding correlations between the scheduling policies of each scheduler. Primary scheduling is based on instruction age, while secondary scheduling uses a best-fit policy (maximize occupancy) with pseudo-random tie-breaking.

Limited associativity The secondary scheduler looks for a ready instruction whose mask is a subset of the free lane mask. This can be achieved using a Content-Addressable Memory (CAM). Zero bits in masks are interpreted as "don't care" bits. As CAMs are power-hungry structures [29], we consider set-associative lookup techniques as an alternative. The mask inclusion lookup hardware is similar to a cache tag lookup. Instead of looking for an address, we look for the subset of a mask. In a way similar to caches, tags can be partitioned into multiple sets. A set-associative lookup only searches through elements of a single set. The set index is computed from the low-order bits of the primary warp identifier. The benefits of set-associative lookup are two-fold. First, the need for large and expensive CAMs is avoided. Second, the instruction buffer memory and scoreboard memory can each be partitioned into separate banks, reducing the number of ports of the instruction buffer. Scheduling restrictions from the set-associative design guarantee conflict-free access.

5. Evaluation

5.1. Performance evaluation

Methodology We used a cycle-accurate simulator of the SM pipeline based on the Barra functional simulator [3]. Simulation parameters are listed on table 2. We follow the methodology of Gebhart et al. [16] by modeling a throughput-limited constant-latency memory, but also simulate a local L1 data cache to better take into account memory divergence effects [20]. We add one extra pipeline stage in addition to scheduling stages to account for the wire delay of instruction delivery.

We observed that NVIDIA's CUDA compiler backend would generally lay out code in the exact same order as the order dictated by divergence frontier analysis. In fact, we found only one CUDA kernel for which a different order was used. We include it in our evaluation as the *TMD1* benchmark. This observation matches the results of the analysis conducted on the SPEC INT 2000 by Collins et al. in the context of control-flow reconvergence for out-of-order processors, who find that 94% of reconvergence points are placed below divergent branches [6]. The synchronization instructions described in section 3.3 are placed at the same

Table 2. Micro-architecture parameters.

Parameter	Baseline	SBI	SWI
Warps × warp-width	32×32	16×64	16×64
Clock rate	1 GHz		
Scheduler latency	1 cycle	1 cycle	2 cycles
Instruction delivery latency	0 cycle	1 cycle	1 cycle
Execution latency	8 cycles		
Scoreboard	6 entries / warp		
L1 cache	48K, 6-way, 128B blocks, 3 cycles		
Memory	10 GB/s (1 SM), 330 ns		

addresses as reconvergence markers in the Tesla binary code. Divergence points are computed from Tesla's divergence instructions. We consider benchmarks from CUDA benchmark of Rodinia [2] and applications from the NVIDIA CUDA SDK [28], as well as two implementations of the Table Maker Dilemna (TMD) application in computer arithmetic that exhibit highly irregular control flow [13]. We distinguish regular application, whose average IPC with 64-wide warps is above 30, from irregular applications.

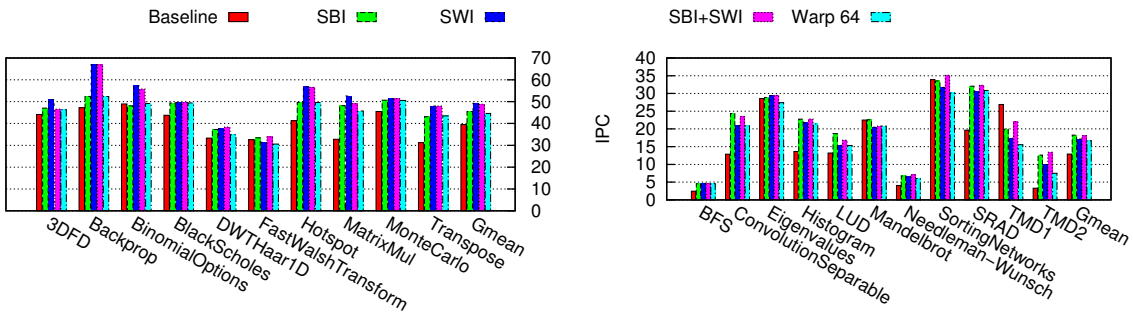
Summary Figure 7 summarizes the relative performance of the baseline architecture, SBI with and without constraints, SWI, and combinations of SBI and SWI. We plot the number of thread instructions per cycle on the SM. The peak IPC of the baseline architecture is 64, as it is limited by the issue rate of two 32-wide warps. SBI and SWI move the peak to 104, as processing units can benefit from the increased relative instruction issue bandwidth. For reference, we consider an implementation based of thread frontiers and 64-wide warps in the comparison. On regular applications, SBI and SWI provide the advantage of wide warps by increasing the relative front-end bandwidth, resulting in average performance increases of 15% and 25%, respectively. In this case, the benefit comes from scheduling more instructions to distinct SIMD groups, rather than warp interweaving per se. The benefits of SBI and SWI are fully realized on irregular applications. Despite the larger warp size and increased front-end latency in the case of SWI, performance is respectively increased by 41% and 33% on average using SBI and SWI.

Specific benchmark discussion *TMD1* performs worse in our experiments, due to the improper code layout issue mentioned above. *TMD2* shows vastly improved performance compared to stack-based execution, even though the latter has dedicated break and return support as the Tesla implementation does [14, 10]. It illustrates the benefits of thread frontier-based reconvergence on unstructured control flow. As the TMD application reflects properties of thread-frontier based reconvergence rather than SBI and SWI, we do not take it into account when computing the performance means. Mandelbrot shows no noticeable performance variation throughout the experiments. We found that a thread block synchronization barrier instruction prevents warp-splits from running ahead across iterations.

Constraints When applied to SBI in isolation, constraints have negligible (less than 0.1%) effect on performance, as seen on figure 8(a). However, constraints reduce the number of instruction issued by 1.3% on regular applications and by 5.5% on irregular applications. SWI takes advantage of the execution resources freed to improve execution throughput : *SortingNetworks* is 2.4% faster when reconvergence constraints are applied to the combination of SBI and SWI. On the other hand, *BFS* and *Histogram* benefit from running threads ahead past reconvergence points and their performance is held back by reconvergence constraints. Overall, the performance impact is modest: we found that most applications have explicit synchronization barriers in their main loop, which enforce warp-split synchronization regardless of reconvergence constraints.

Lane shuffling Figure 8(b) illustrates the effect of the lane shuffling policies listed on table 1 on irregular applications. Speedup of the *XorRev* policy over *Linear* ranges from -1.8% (3dfd, not shown) to +7.7% (Needleman-Wunsch). The geometric mean is respectively +0.3% and +1.4% on regular and irregular applications. While the improvements are small, they come at essentially no extra cost.

Set-associative matching Figure 9 quantifies the performance impact of reduced lookup associativity of SWI on irregular applications. We find that associativity bears a moderate impact on performance: even a direct-mapped configuration achieves at least 85% of the performance of the fully-associative configuration. Compared to the baseline, direct-mapped SWI achieves a speedup of 26%, while fully-associative SWI achieves



(a) Regular applications

(b) Irregular applications

Figure 7. Performance of SBI, SWI, and combination of SBI and SWI, with a thread-frontier based 64-wide warp implementation for reference.

34%. On regular applications, the performance difference is even thinner, as even fully associative lookup finds few warp interweaving opportunities. Direct-mapped SWI achieves 96% of the fully-associative performance, maintaining an 18% speedup over the baseline.

5.2. Hardware overhead

We now estimate the hardware cost of SBI and SWI. Table 3 summarizes the main hardware requirements of each technique. We conservatively assume that the fully-populated CCT can accommodate 8 entries per warp, totaling 10 warp-splits per warp with the HCT. Prior work suggests that the CCT could be made smaller with no significant performance impact [24, 10]. The baseline implementation uses a stack with 3 blocks of 4 entries of 64-bit each per warp.

To deliver two instructions to each lane, the fanout of the instruction broadcast network is increased by a factor of two, and additional wiring is needed. The width of a half-datapath on Fermi is 0.65mm from our measurements on a die photography. This distance is within reach of single clock cycle at 1GHz. The register file needs to be extended to support two distinct addresses. The overhead can be bound by the cost of breaking the register file into one bank per lane, which was estimated at 0.722mm^2 in 90nm for a 8192-entry register file by Fung et al. [15]. Accounting for process scaling and register file size differences, our conservative estimate is 0.57mm^2. We synthesized the other major components of the design using a production RTL compiler with a gate library from a state-of-the-art process technology. To allow comparison with the baseline implementation, area results are scaled to the older 40nm process used by Fermi. Results are reported on table 4. A Fermi SM being 15.6 mm^2 from measurements on a publicly-available die photograph, the respective area overheads of SBI, SWI and both are 3.0%, 2.9% and 3.7%.

Table 4. Area of each component.

	Area ($\times$1000 μm^2)			
Component	Baseline	SBI	SWI	SBI+SWI
RF	–	+570	+570	+570
Scoreboard	87.6	65.6	87.6	131.2
Scheduler	–	–	+27.4	+27.4
HCT	66.8	88.8	43.8	88.8
CCT	584.4	480.8	480.8	480.8
Insn. Buffer	52.8	52.8	33.4	67.4
Total	791.6	1258	1243	1365.6
Overhead	–	466.4	451.4	574

6. Related work

Dynamic warp formation (DWF) mitigates the impact of divergence by dynamically building warps [15]. While it improves the utilization of execution units, it may act at the expense of memory divergence and thread reconvergence may be delayed [14, 20]. SBI and SWI preserve memory access locality by keeping threads of the same warp together, and reconvergence constraints guarantee that threads reconverge. DWS [24] was discussed in section 3.1. SBI goes beyond DWS by running concurrent branches simultaneously rather than interleaved in time, with the same execution resources re-

Table 3. Summary of the hardware requirements for each proposed technique.

Component	Baseline	SBI	SWI	SBI+SWI
RF	Single-decoder	Segmented	Segmented	Segmented
Scoreboard	$2 \times 24 \times 48$-bit	24×144-bit	$2 \times 24 \times 48$-bit	24×288-bit
Scheduler	Symmetric	Warp-split	Associative lookup	Associative lookup
Warp pool/HCT	$2 \times 24 \times 64$-bit	24×201-bit	24×104-bit	24×201-bit, banked
Stack/CCT	144×256-bit	128×104-bit	128×104-bit	128×104-bit
Insn. buffer	48×64-bit	48×64-bit	24×64-bit, dual-ported	48×64-bit, dual-ported

quirements. The reconvergence policy and constraints we propose may be applied to both DWF and DWS.

Block Compaction [14] and Large Warps [26] rely on coarser scheduling units than the warps of today's architectures and compact successive pipeline waves. Our work is orthogonal to these approaches, as SBI and SWI could be used together with compaction of larger warps. Thread frontiers [10] define an optimal execution order of basic blocks that ensures early reconvergence for arbitrary control flow, assuming sequential execution of branch path. We generalize it to parallel execution of basic blocks for the same warp. Control-flow reconvergence was extensively studied in the context of superscalar processors to reduce branch misprediction penalties [6, 1]. Al-Zawawi et al. use a control-flow stack and PC comparisons to detect reconvergence in a checkpointing micro-architecture [1]. Stack updates happen in program order, so parallel path traversal was not considered.

Architectures originating from vector processors that blend together thread-level, data-level and instruction-level parallelism have been proposed [12, 18, 30]. These approaches also leverage a data-parallel computing substrate while allowing more flexible means of execution. The problems to address differ between these works and SBI/SWI, as the former use an explicit vector programming model and the latter are based on an SIMD model with implicit predication.

Minimal Multi Threading shares identical instructions across threads in an out-of-order SMT architecture [23]. Control-flow reconvergence is detected by comparing the PC histories of threads. The out-of-order micro-architecture described runs 4 threads, targeting a significantly different design point than the 1536-thread in-order GPU architecture we consider.

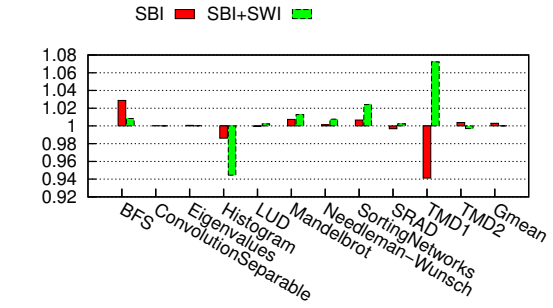
Glew outlines a Dual Instruction, Multiple Threads (DIMT) micro-architecture in a talk [17]. DIMT consists in issuing two different instructions to a wider array of execution units. SBI can be considered as an implementation of the DIMT idea. However, no discussion is made in [17] about concrete implementations, nor about ways to achieve reconvergence. These two issues and SWI are the main contributions of the present work. Dasika et al. propose Divergence-Folding [8], which improves utilization of functional units by statically scheduling instructions from different branch paths to subsets of the same datapath. Divergence-Folding uses two functional units and two RF write ports per lane, while SBI shares the same execution and RF resources and relies on dynamic scheduling.

In addition to sharing instructions, recent works also factor out common computations or common data across threads [4, 5, 9, 23]. We expect that SBI and SWI can be combined with data-sharing techniques to improve their flexibility in the face of data divergence.

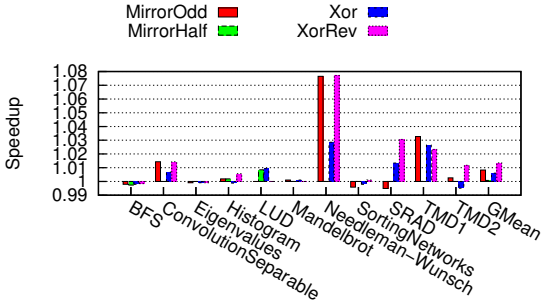
7. Conclusion

We presented and evaluated two complementary techniques that leverage the computational power of GPU architectures that is underexploited due to branch and memory divergence. Each technique performs fine-grained execution resource allocation by incorporating one additional form of parallelism. SBI takes advantage of branch-level parallelism across SIMD instructions that are guaranteed not to overlap. Two problems that arise consist in ensuring the reconvergence of warp-splits and tracking instruction dependencies. We overcome them by respectively introducing reconvergence constraints and an improved scoreboard design. SWI exploits warp-level parallelism. Instructions from different warps are guaranteed to be free of dependencies. The difficulty lies in finding instructions with non-overlapping activity masks. We solve it using a set-associative mask lookup and improve warp affinity using lane shuffling.

Both SBI and SWI execute threads of close identifiers together, in order to preserve memory access patterns and their locality. These techniques rely on full dynamic scheduling and require minimal compiler involvement.



(a) SBI constraints



(b) SWI lane shuffling policy

Figure 8. Effect of constraints and lane shuffling on respective performance of SBI and SWI (irregular applications).

SBI and SWI complement each other. SBI works best on irregular workloads, achieving an average speedup of 41%. Regular workloads benefit most from SWI, which accelerates their execution by 25%. The combination of both techniques retain most of their individual advantages, leading to respective speedups of 40% and 23% on irregular and regular workloads.

Remaining work includes developing robust adaptive policies to schedule warps, perform lane shuffling and decide when running ahead is beneficial. Flexibility may be improved further by allowing more decoupling between lanes, without compromising efficiency. We expect to see more blurring of the lines between SIMT and clustered SMT architectures in the future, as each architecture incorporates successful aspects of the other.

Acknowledgements

We thank Michael Shebanow and Andy Glew for early discussions on the original idea, and Mourad Gouicem and Pierre Fortin for their help with the TMD application. We also thank the anonymous reviewers for their constructive suggestions.

The second author was affiliated with ÉNS de Lyon and Università degli Studi di Siena during the realization of this work. This project was supported by Kalray and ÉNS de Lyon through a research collaboration on advanced computer architecture and parallel processing and its application to the Kalray MPPA® line of products.

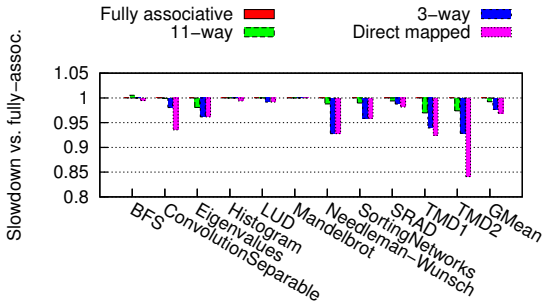


Figure 9. Slowdown of SWI lookup set-associativity compared to fully-associative lookup.

References

[1] A. S. Al-Zawawi, V. K. Reddy, E. Rotenberg, and H. H. Akkary. Transparent control independence (TCI). *SIGARCH Comput. Archit. News*, 35:448–459, June 2007.

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. *IEEE Workload Characterization Symposium*, 0:44–54, 2009.

[3] C. Collange, M. Daumas, D. Defour, and D. Parello. Barra: a parallel functional simulator for GPGPU. In *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 351–360, 2010.

[4] C. Collange, D. Defour, and Y. Zhang. Dynamic detection of uniform and affine vectors in GPGPU computations. In *Europar 3rd Workshop on Highly Parallel Processing on a Chip (HPPC)*, volume LNCS 6043, pages 46–55, 2009.

[5] C. Collange and A. Kouyoumdjian. Affine Vector Cache for memory bandwidth savings. Technical Report ensl-00649200, ENS Lyon, Dec. 2011.

[6] J. D. Collins, D. M. Tullsen, and H. Wang. Control flow optimization via dynamic reconvergence prediction. In *IEEE/ACM International Symposium on Microarchitecture*, pages 129–140. IEEE Computer Society, 2004.

[7] B. W. Coon, P. C. Mills, S. F. Oberman, and M. Y. Siu. Tracking register usage during multithreaded processing using a scoreboard having separate memory regions and storing sequential register size indicators. US Patent 7434032, October 2008.

[8] G. Dasika, A. Sethia, T. Mudge, and S. Mahlke. PEPSC: A power-efficient processor for scientific computing. In *PACT*, 2011.

[9] M. Dechene, E. Forbes, and E. Rotenberg. Multithreaded instruction sharing. Technical report, North Carolina State University, 2010.

[10] G. Diamos, A. Kerr, H. Wu, S. Yalamanchili, B. Ashbaugh, and S. Maiyuran. SIMD re-convergence at thread frontiers. In *MICRO 44: Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, December 2011.

[11] R. Dolbeau and A. Seznec. CASH: Revisiting hardware sharing in single-chip parallel processor. *Journal of Instruction-Level Parallelism*, 6:1–16, 2004.

[12] R. Espasa and M. Valero. Multithreaded vector architectures. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, HPCA'97, pages 237–244, 1997.

[13] P. Fortin, M. Gouicem, and S. Graillat. Towards solving the table maker dilemma on GPU. In *20th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 12)*, 2012.

[14] W. Fung and T. Aamodt. Thread block compaction for efficient SIMT control flow. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, pages 25–36, February 2011.

[15] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. *ACM Trans. Archit. Code Optim.*, 6:7:1–7:37, July 2009.

[16] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. In *Proceeding of the 38th annual international symposium on Computer architecture*, pages 235–246, 2011.

[17] A. Glew. Coherent vector lane threading. *Berkeley ParLab Seminar*, 2009.

[18] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The Vector-Thread architecture. *IEEE MICRO*, 24(6):84–90, 2004.

[19] R. Kumar, N. P. Jouppi, and D. M. Tullsen. Conjoined-core chip multiprocessing. In *IEEE/ACM International Symposium on Microarchitecture*, pages 195–206, 2004.

[20] A. Lashgar and A. Baniasadi. Performance in GPU architectures: Potentials and distances. In *9th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD11), in conjunction with ISCA-38*, 2011.

[21] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, pages 451–460, 2010.

[22] J. E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[23] G. Long, D. Franklin, S. Biswas, P. Ortiz, J. Oberg, D. Fan, and F. T. Chong. Minimal multi-threading: Finding and removing redundant instructions in multi-threaded processors. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 337–348, 2010.

[24] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. *SIGARCH Comput. Archit. News*, 38(3):235–246, 2010.

[25] P. C. Mills, J. E. Lindholm, B. W. Coon, G. M. Tarolli, and J. M. Burgess. Scheduler in multi-threaded processor prioritizing instructions passing qualification rule. US Patent 7949855, May 2011.

[26] V. Narasiman, C. J. Lee, M. Shebanow, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving GPU performance via large warps and two-level warp scheduling. In *MICRO 44: Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, December 2011.

[27] J. Nickolls and W. J. Dally. The GPU computing era. *IEEE Micro*, 30:56–69, March 2010.

[28] NVIDIA CUDA SDK, 2010. `http://www.nvidia.com/cuda/`.

[29] K. Pagiamtzis and A. Sheikholeslami. Content-addressable memory (CAM) circuits and architectures: a tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, march 2006.

[30] S. Rivoire, R. Schultz, T. Okuda, and C. Kozyrakis. Vector lane threading. In *Proceedings of the 2006 International Conference on Parallel Processing*, ICPP '06, pages 55–64, 2006.

[31] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. *SIGARCH Comput. Archit. News*, 23:392–403, May 1995.

GPU-Initiated On-Demand High-Throughput Storage Access in the BaM System Architecture

Zaid Qureshi*†
zqureshi@nvidia.com
NVIDIA/UIUC
USA

Vikram Sharma Mailthody*†
vmailthody@nvidia.com
NVIDIA/UIUC
USA

Isaac Gelado
igelado@nvidia.com
NVIDIA
USA

Seungwon Min†
davmin@nvidia.com
NVIDIA/UIUC
USA

Amna Masood†
ammasood@amd.com
AMD/UIUC
USA

Jeongmin Park
jpark346@illinois.edu
UIUC
USA

Jinjun Xiong
jinjun@buffalo.edu
University at Buffalo
USA

C. J. Newburn
cnewburn@nvidia.com
NVIDIA
USA

Dmitri Vainbrand
dvainbrand@nvidia.com
NVIDIA
USA

I-Hsin Chung
ihchung@us.ibm.com
IBM Research
USA

Michael Garland
mgarland@nvidia.com
NVIDIA
USA

William Dally
bdally@nvidia.com
NVIDIA/Stanford
USA

Wen-mei Hwu
whwu@nvidia.com
NVIDIA/UIUC
USA

ABSTRACT

Graphics Processing Units (GPUs) have traditionally relied on the host CPU to initiate access to the data storage. This approach is well-suited for GPU applications with known data access patterns that enable partitioning of their dataset to be processed in a pipelined fashion in the GPU. However, emerging applications such as graph and data analytics, recommender systems, or graph neural networks, require fine-grained, data-dependent access to storage. CPU initiation of storage access is unsuitable for these applications due to high CPU-GPU synchronization overheads, I/O traffic amplification, and long CPU processing latencies. GPU-initiated storage removes these overheads from the storage control path and, thus, can potentially support these applications at much higher speed. However, there is a lack of systems architecture and software stack that enable efficient GPU-initiated storage access. This work presents a novel system architecture, BaM, that fills this gap. BaM features a fine-grained software cache to coalesce data storage requests while minimizing I/O traffic amplification. This software cache communicates with the storage system via high-throughput queues that enable the massive number of concurrent threads in modern GPUs to make I/O requests at a high rate to fully utilize the storage devices and the system interconnect. Experimental results show that BaM delivers 1.0× and 1.49× end-to-end speed up for BFS and CC graph analytics benchmarks while reducing hardware costs by up to 21.7× over accessing the graph data from the host memory. Furthermore, BaM speeds up data-analytics workloads by 5.3× over CPU-initiated storage access on the same hardware.

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**; • **Software and its engineering** → **Software organization and properties**; • **Computer systems organization** → **Architectures**; • **Information systems** → **Storage architectures**.

KEYWORDS

GPUs, GPUDirect, SSDs, Memory capacity, Memory hierarchy, Storage systems

ACM Reference Format:
Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, C. J. Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. 2023. GPU-Initiated On-Demand High-Throughput Storage Access in the

*Both authors contributed equally to this research.
†Work was done while at UIUC.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9916-6/23/03.
https://doi.org/10.1145/3575693.3575748

BaM System Architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3575693.3575748

1 INTRODUCTION

After over a decade of phenomenal growth in compute throughput and memory bandwidth [33, 45], GPUs have become popular compute devices for HPC and machine learning applications. Emerging high-value data-center workloads such as graph and data analytics [39, 46, 59, 68], graph neural networks (GNNs) [22, 30], and recommender systems [1, 20, 40, 41, 74] can potentially benefit from the compute throughput and memory bandwidth of GPUs. These applications access massive datasets organized into array data structures whose sizes range from tens of GBs to tens of TBs today, and are expected to grow rapidly in the foreseeable future.

Storing these datasets as in-memory objects enables applications to naturally and efficiently process the data. However, the memory capacity of GPUs, in spite of a 53× increase from that of G80 to A100 [33, 44], is only at 80GB, far smaller than the required capacity to accommodate entire datasets of these workloads. Thus, some state-of-the-art approaches rely on CPU user/OS code to partition datasets into chunks and orchestrate the appropriate storage access and data transfers into the GPU memory for application processing.

Another approach is to rely on memory-mapped files and GPU page faults to active the CPU page fault handler to transfer data whenever the application accesses data not present in GPU memory. In this paper, we refer to both alternatives as a **CPU-centric approach**. Our experiments show that CPU-centric approaches are bottlenecked by CPU software and CPU/GPU synchronization overheads, resulting in poor performance (See § 5 and [50]).

To avoid the inefficiencies of the CPU-centric approaches, some state-of-the-art solutions use the host memory [39], whose capacity typically ranges from 128GB to 2TB today, or pool together multiple GPUs' memories [1] to hold the entire datasets. We refer to the use of host memory or pooling multiple-GPUs' memory to extend GPU memory capacity as a *DRAM-only* solution.

Extending the host memory to the level of tens of TBs is an extremely expensive proposition in the foreseeable future. Similarly, pooling multiple GPUs' memory to reach tens of TBs can be very expensive as well. The memory capacity of each A100 GPU is only 80GB, so a 10TB pool would require 125 A100 GPUs. Furthermore, using the host memory and pooling GPU memories both require pre-loading the dataset into the extended memory. Some of the preloaded data may not be ultimately used due to data-dependent accesses. We will show with results from graph analytics that the performance of a host-memory solution is comparable to or lower than our proposed approach despite its much higher cost.

Proposal: We propose a novel system architecture called BaM (Big accelerator Memory). BaM capitalizes on the recent improvements in latency, throughput, cost, density, and endurance of storage devices to realize another level of the accelerator memory hierarchy. The goal of BaM's design is to provide efficient abstractions for the GPU threads to easily make on-demand, fine-grained accesses to massive datasets in the storage and achieve much higher application performance than state-of-the-art solutions.

Prior attempts[57, 58] to enable GPU threads to generate on-demand storage requests achieved low throughput (~823K IOPs on the NVIDIA A100 GPU), as shown in § 5.1. In this paper, we present and evaluate the effectiveness of the key components and the overall design of BaM in addressing two key technical challenges in efficient on-demand storage accesses for accelerator applications.

First, there is currently a lack of **fast** mechanisms for the GPU application code to generate on-demand storage access requests without incurring the CPU software bottlenecks such as the OS page fault handler. To fill this gap, BaM features a scalable, highly concurrent, high-throughput configurable software cache that takes advantage of the massive memory bandwidth and atomic operation throughput of modern GPUs. The software cache coalesces redundant on-demand accesses and facilitates reuse of storage data while providing high application-perceived throughput.

Second, while the CPU-centric approaches suffer from the low-degree of CPU thread-level parallelism available to page fault handlers and device drivers, there is currently a lack of GPU mechanisms for orchestrating storage accesses without relying on the CPU. To address this issue, BaM provides a user-level GPU library of highly concurrent submission/completion queues in GPU memory that enables GPU threads whose on-demand accesses do not hit in the software cache to make storage accesses in a high-throughput manner. This user-level approach removes the page fault handling bottleneck, and uses fine-grained synchronization and minimal critical sections to reduce software overhead for each storage access and support a high-degree of thread-level parallelism.

There is a trend towards increasing autonomy and asynchrony of GPUs. The GPUDirect Async family [42] of technologies accelerate the control path when moving data directly into GPU memory from memory or storage. Each transaction involves initiation, where structures like work queues are created and entries within those structures are readied, and triggering, where transmissions are signaled to begin. To our knowledge, BaM is the first ***accelerator-centric*** approach where GPUs can create on-demand accesses to data where it is stored, be it memory or storage, **without relying on the CPU to initiate or trigger the accesses.** Thus, BaM marks the beginning of a new variant of this family that is GPU kernel initiated (KI): *GPUDirect Async KI Storage*.

While the user-level storage device queues raise security concerns for traditional monolithic server architectures, the recent shift in data centers toward zero-trust security models that provide security guarantees through trusted hardware or software services have provided the new system framework for securing accelerator-centric user-level storage access models like BaM [26–28].

We have built a prototype BaM system through novel organization of off-the-shelf hardware components and development of a novel custom software stack that takes advantage of the advanced architectural features in recent GPUs. Evaluation using a variety of workloads with multiple datasets shows that BaM is on-par with a 21.7× more expensive host-memory DRAM-only solution and up to 5.3× faster than a state-of-the-art CPU-centric software solution. Overall, we make the following contributions. We

(1) propose BaM, an accelerator-centric system architecture in which GPU threads perform on-demand accesses to array data where it is stored, be it memory or storage, without relying on the CPU to initiate these accesses;

(2) enable on-demand, high-throughput fine-grained access to storage through a novel library of highly concurrent submission/completion protocol queues;
(3) provide high-throughput, scalable software-defined caching and software API for programmers to exploit locality and control data placement for their applications; and
(4) construct and evaluate a prototype design for GPUs to access massive storage data in a cost-effective manner.

We have published a full version of the paper that includes appendix covering in-depth analysis on limitations of current system with additional evaluations [50]. BaM is implemented completely in open-source, and both hardware and software requirements are publicly accessible [5, 36, 49].

2 BACKGROUND

This section covers the limitations with the common solution that keeps large datasets in abundant CPU memory or pooled multi-GPU memory (§2.1) and then describe why GPUs can tolerate long storage access latency (§2.2).

2.1 Leveraging CPU or Pooled Multi-GPU Memory

Applications can leverage CPU memory or even pool together the memory of multiple GPUs to host large data structures. Previous works show that the GPU provides sufficient memory-level parallelism to tolerate the access latency of these memories and significantly out-perform the UVM [48] solution for graph traversal applications [39].

However, regardless of whether CPU memory or pooled GPU memory is used to host these data structures, this approach suffers from two major pitfalls. First, *data must still be loaded from the storage to the memory before any GPU computation can start.* Often this initial data loading can be the main performance bottleneck. (see the Target (T) system of Figure 7). Second, *hosting the dataset in CPU memory or pooled GPU memory requires scaling the available memory*, by either increasing the CPU DRAM size or the number of GPUs in the system, with the dataset size, *and thus can be prohibitively expensive for massive datasets.*

2.2 Tolerating Storage Access Latency

As the storage device latency is reduced due to technological advancements like Optane [23] or Z-NAND [55] media, software overhead is becoming a significant fraction of overall I/O access latency. Our experiments show that even with `io_uring`, a highly optimized CPU software stack[4], *as device latency decreases, OS kernel software overhead severely limits the storage access throughput and becomes a significant fraction, up to 36.4%, of the total storage access latency.*

To address this shift, emerging storage systems allow applications to make direct user-level I/O accesses to storage [17, 25, 26, 31, 34, 35, 53, 67, 69]. The storage system allocates user-level queue pairs, akin to NVMe I/O submission (SQ) and completion (CQ) queues, which the application threads can use to enqueue requests and poll for their completion. Using queues to communicate with storage systems forgoes the userspace to kernel crossing of traditional file system access system calls. Instead, isolation and other file system properties are provided through trusted services running as trusted user-level processes, kernel threads, or even storage system firmware running on the storage server/controller [25, 26, 53, 67].

In such systems, the parallelism required to tolerate access latency and achieve full throughput of the device is governed by Little's Law: $T \times L = Q_d$, where T is the target throughput, L is the average latency, and Q_d is the minimal queue depth required at any point in time to sustain the target throughput. To achieve the full potential of the critical resource, PCIe ×16 Gen4 connection providing ~26GBps of bandwidth, then T is $26GBps/512B = 51M/sec$ and $26GBps/4KB = 6.35M/sec$ for 512B and 4KB access granularities, respectively. The average latency, L, depends on the SSD devices used and is measured when the SSD provides its maximal throughput. For the experiments reported in this paper, L is $11us$ and $324us$ for the Intel Optane and Samsung 980pro SSDs, respectively. From Little's Law, to sustain a desired 51M accesses of 512B each, the system needs to accommodate a queue depth of $51M/s \times 11us = 561$ requests (70 requests for 4KB) for Optane SSDs. For the Samsung 980pro SSDs, the required Q_d for sustaining the same target throughput is $51M \times 324us = 16,524$ (2057 for 4KB). Note that Q_d can be spread across multiple physical device queues. To sustain T over a computation phase, there need to be a substantially higher number of concurrently serviceable access requests than Q_d over time.

The emerging queue-based storage systems present major challenges to sustaining T in massively parallel execution paradigms. Take as an example using NVMe queues to make I/O requests. After requests are enqueued into a NVMe SQ, the queue's doorbell must be rung with the updated queue tail to notify the storage controller of the new request(s). As these doorbell registers are write-only, when a thread rings a doorbell, it must make sure that no other thread is writing to the same register and that the value it is writing is valid and is a newer value than any value written to that register before. Implementing queue insertion as a critical section, while simple, imposes significant serialization latency, which might be fine for the CPU's limited parallelism but can cause substantial overhead for thousands of GPU threads. Storage interfaces like [25, 26, 53] face similar serialization challenges. These challenges are addressed in the design of the BaM queues.

3 BAM SYSTEM AND ARCHITECTURE

The goal of BaM's design is to provide high-level abstractions for accelerators to make on-demand, fine-grained, high-throughput access to storage while enhancing the storage access performance. To this end, BaM provisions storage I/O queues and buffers in the GPU memory as shown in Figure 1, and builds on the recent memory-map capability of GPUs to map the storage doorbell registers to the GPU address space. Although doing so enables the GPU threads to access terabytes of data on storage, BaM must address three key challenges in providing an efficient and effective solution:

(1) As storage protocols and devices exhibit significant latency, BaM must leverage the GPU's massive parallelism (up to three orders of magnitude more than the CPU) to keep many requests in flight, efficiently tolerate such latency, and overlap it with computation. (See §3.3)

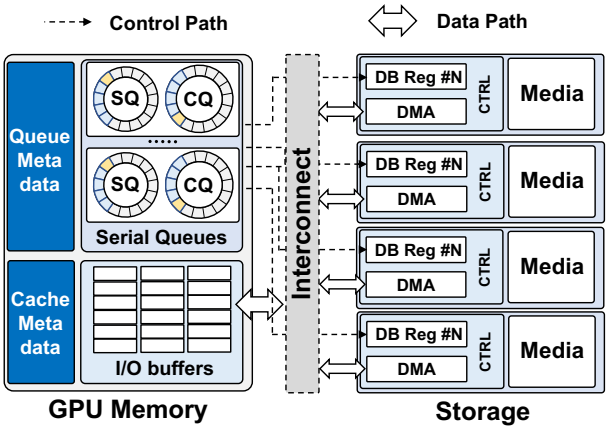


Figure 1: Logical view of BaM design.

(2) As storage devices have relatively low bandwidth and GPUs have limited memory capacity, BaM must optimally utilize these resources. (See §3.4)
(3) As GPU kernels generally don't expect to make storage accesses, BaM must provide high-level abstractions that hide BaM's complexity and make it easy for the programmers to integrate BaM into their GPU kernels. (See §3.5)

Next, we provide an overview of BaM and then explain how BaM addresses these challenges.

3.1 BaM System Overview

BaM presents the `bam::array` high-level programming abstraction, enabling programmers to easily integrate BaM into their existing GPU applications. An application can call BaM APIs to map the `bam::array` to data on storage, akin to `mmap`'ing a file.

Figure 2 shows how a GPU thread uses BaM to access data. When a GPU thread accesses data with the `bam::array` abstraction ❶, it uses the abstraction to determine the offset, i.e. cache line, for the data being accessed ❷. Threads in a warp can coalesce their accesses ❸ if multiple threads access the same cache line. For each unique cache line being accessed, a single thread probes the cache line's metadata ❹ on behalf of the rest of the threads, improving cache access efficiency.

If an access hits in the cache, the thread can directly access the data in GPU memory. If the access misses, the thread needs to fetch data from the backing memory. The BaM software cache is designed to optimize the bandwidth utilization to the backing memory in two ways: (1) by eliminating redundant requests to the backing memory and (2) by allowing users to configure the cache for their application's needs.

If the storage system or device is backing the data, the GPU thread enters the BaM I/O stack to prepare a storage I/O request ❺, enqueues it to a submission queue ❻, and then waits for the storage controller to post the corresponding completion entry ❼. The BaM I/O stack aims to amortize the software overhead associated with the storage submission/completion protocol by leveraging the GPU's immense thread-level parallelism, and enabling low-latency batching of multiple submission/completion (SQ/CQ) queue entries

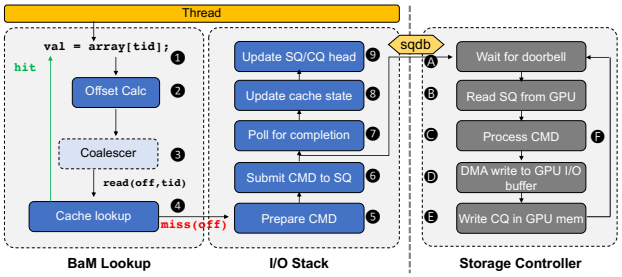


Figure 2: Life of a GPU thread in BaM.

to minimize the cost of expensive doorbell register updates and reducing the size of critical sections in the storage protocol.

On receiving the doorbell update (A), the storage controller fetches the corresponding SQ entries (B), processes the command (C) and transfer the data between SSD and the GPU memory (D). At the end of the transfer, the storage controller posts a completion entry in the CQ (E). After the completion entry is posted, the thread updates the cache state ❽, update the SQ/CQ state ❾ and then can access the fetched data in GPU memory.

3.2 Comparison With the CPU-Centric Approaches

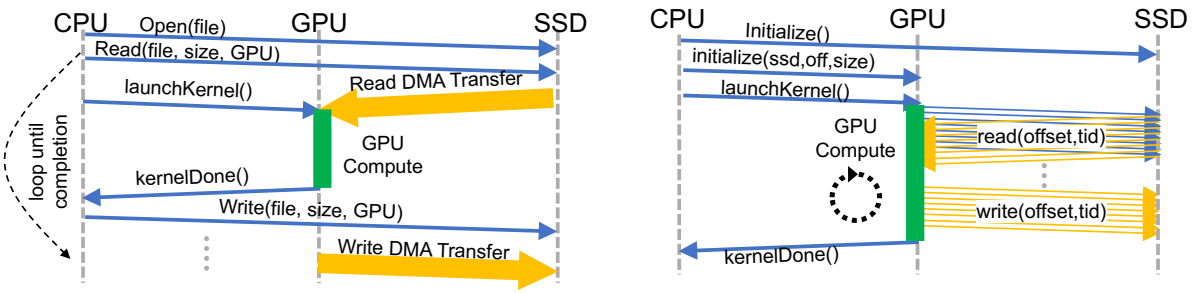
When compared to the proactive tiling CPU-centric approach shown in Figure 3a, BaM has three main advantages. First, with proactive tiling, the CPU ends up copying data between the storage and the GPU memory and launching compute kernels multiple times to cover a large dataset. Each kernel launch and termination incurs costly synchronization between the CPU and the GPU. BaM allows GPU threads to both compute and fetch data from storage, as shown in Figure 3b, which reduces the frequency of CPU-GPU synchronization and GPU kernel launches. Furthermore, the storage access latency of some threads can also be overlapped with the compute of other threads,which improves the overall performance.

Second, in proactive tiling, because the compute is offloaded to the GPU and the data movement is orchestrated by the CPU, the CPU cannot accurately determine which parts of the data are needed and when they are needed, thus it ends up fetching many unneeded bytes. In contrast, with BaM, a GPU thread fetches bytes only when they are used, reducing the I/O amplification overhead.

Third, with proactive tiling, programmers expend effort to partition the application's data and overlap compute with data transfers to hide storage access latency. BaM allows the programmer to naturally access the data through the array abstraction and harness GPU thread parallelism across large datasets to hide the storage access latency.

3.3 High-Throughput I/O Queues

BaM leverages GPU's massive thread-level parallelism and fast hardware scheduling to maintain the queue depths needed to hide storage access latency and to achieve peak storage throughput. However, ringing doorbells after enqueueing commands or cleaning up SQ entries, in the existing storage I/O protocols requires serialization. A critical section that encompasses the process of enqueuing a command and ringing the doorbell, albeit simple, would



(a) CPU-Centric Model.

(b) BaM Model.

Figure 3: Comparison between the traditional CPU-centric and BaM computation model is shown in (a) and (b). BaM enables GPU threads to directly access storage enabling compute and I/O overlap at fine-grain granularity.

result in poor throughput and significant serialization latency when thousands of threads enqueue I/O requests concurrently. Instead, BaM uses fine-grained memory synchronization enabling many threads to enqueue into the SQ, poll the CQ, or mark queue entries for cleanup in parallel without large critical sections.

We will use the SQ implementation in BaM as an example to explain how fine-grained memory synchronization helps to achieve the above design goals and principles. BaM keeps the following metadata per SQ in the GPU memory: 1) local copies of the queue's head and tail, 2) atomic `ticket` counter, 3) `turn_counter` array, an integer array of the same length as the queue, 4) `mark` bit-vector of the same length as the queue, and 5) a `lock`.

To enqueue requests, threads first atomically increment the ticket counter by two. The returned `ticket` value is an index into a virtual queue with 2^{32} entries and can be divided by the physical queue size to assign each thread an `entry` in the physical queue, the remainder, and a `turn` value along with a valid bit for that entry, the quotient. Using its `entry` and `turn`, each thread can wait for both the enqueue and dequeue operations of all previous commands in its assigned `entry` in the queue. Each thread uses its `entry` to index into the `turn_counter` array, and polls on the location until the counter equals the thread's `turn`. That is, the `turn_counter` array tracks the mapping of each physical queue entry to an virtual queue entry. It effectively creates sub-queues whose waiting threads are ordered by their `turn` value for each physical queue entry.

When it is the thread's turn, i.e., the virtual queue entry assigned to the thread becomes active, the thread can copy its I/O access command into its assigned (`entry`) position in the physical queue. Afterwards, each thread sets the position's corresponding bit in the `mark` bit-vector. The `turn_counter` array enables as many threads as the size of the physical queue to copy their commands in parallel.

The threads call the `move_tail` routine to complete the insertion of their requests. One thread will successfully acquire the lock and will move the tail past the consecutive new entries inserted by the calling threads and reset the `mark` bits associated with these entries with the `reset_marks` routine. The thread then rings the queue's doorbell at the storage controller with the new tail and releases the lock. This coalesces the doorbell writes, which are expensive operations over the PCIe interconnect, for multiple calling threads and improves the effective throughput. The rest of the calling threads will return if their `mark` bits are reset, signifying the SQ's tail will be moved past their enqueued entries. If the `mark` bit for a thread has not been reset, the thread keeps trying to take the queue's `lock` and complete the insertion of its request. Once the thread knows that its position's `mark` bit has been reset, it atomically increments the position's `turn_counter` value by one (to an odd value).

After the thread's command is submitted, the thread can poll, without any lock, on the CQ to find the completion entry for its submitted request. When it finds the completion entry, it marks the entry for dequeuing by the next movement of the CQ head in the CQ's `mark` bit-vector. The CQ's head and doorbell are managed similarly as the SQ's tail, except a thread stops trying to reset its CQ entry's `mark` bit if it notices the CQ's head has already been moved past the entry it marked for dequeueing. A winning thread among the calling threads rings the doorbell with a new `head` position to communicate forward progress to the storage system.

The storage controller communicates forward progress by specifying a new SQ head in each CQ entry. The thread with the CQ `lock` reads this field from the last CQ entry whose mark bit was reset by it, and then iterates from the current SQ head until the specified new head, incrementing each position's `turn_counter` value by one (to an even value), allowing threads waiting for those positions to enqueue their commands. The thread then updates the SQ head and releases the CQ `lock`.

3.4 BaM Software Cache

The BaM software cache is designed to enable optimal use of the limited GPU memory and storage-access bandwidth. Traditional OS-kernel-mode memory management (allocation and translation) implementations must support diverse, legacy application/hardware needs. As a result, they contain large critical sections that limit the effectiveness of multi-threaded implementations. BaM addresses this bottleneck by allocating all the virtual and physical memory required for the software cache when starting each application. This approach reduces critical sections to only require a lock when inserting or evicting a cache line, which allows the BaM cache to support many more concurrent accesses.

The BaM cache aims to minimize the number of redundant I/O requests to the storage, avoiding unnecessary I/O traffic. To this end, when a thread probes the cache with an offset, it checks the corresponding cache line's state. If the accessed cache line is not in the cache, the thread locks the cache line, finds a victim to evict, and requests the cache line from the backing memory. When the request completes, the requesting thread unlocks the cache line by making its state valid and incrementing its reference count. This locking forces other threads that access the same cache line to wait until the cache line has been inserted and thus prevents multiple requests to the backing memory for the same cache line, exploiting locality and minimizing the number of requests to the backing memory. If, when the thread probes the cache line state, it is valid, the thread atomically increments the cache line's reference count. When the thread is done using the cache line, it decrements it's reference count.

To avoid contention among concurrent evictions, the BaM cache uses a clock replacement algorithm [14]. The cache has a global counter that gets incremented when a thread needs to find a cache slot. The returned value of the counter assigns the thread a cache slot to use, allowing concurrent threads to evict unique cache slots in parallel. If the assigned cache slot is currently mapped to a cache line that has a non-zero reference count, that is it is pinned, the thread will increment the counter again to attempt to replace another cache slot until it finds a cache slot that is not mapped to a pinned cache line. The thread will then mark the cache line invalid and change the mapping of the cache slot to point to the newly inserted cache slot.

Warp Coalescing: Threads in a warp may contend among themselves in accessing the BaM software cache, especially when consecutive threads try to access contiguous bytes in memory. This contention incurs significant overhead when the needed cache line is already in the GPU memory. To overcome this, BaM's cache implements coalescing in software using the `__match_any_sync` warp primitive to synchronize among threads in the warp and a mask is computed letting each thread know which other threads in the warp are accessing the same offset. From that group, the threads decide on a leader and only the leader queries the cache and manipulates the requested cache line's state. The threads in the group synchronize using the `__shfl_sync` primitive, and the leader broadcasts the address of the requested offset in the GPU memory to the group.

Compared to warp coalescing implemented in prior works [57] where the unique cache probes in a warp are serialized and require many instructions, BaM's coalescer enables all threads in a warp to divide themselves into groups according to the cache lines they are accessing with a single instance of the new `__match_any_sync` warp synchronization primitive. A leader is determined for each group in parallel and each leader can independently probe the cache for the group's needed cache line, with no inter-group dependencies or synchronization.

3.5 BaM Abstraction and Software APIs

BaM's software stack provides the programmer an array-based high-level API (`bam::array<T>`), consistent with array interfaces defined in modern programming languages (e.g. C++, Python, or Rust). As GPU kernels operate on such arrays, BaM's abstraction minimizes the programmer's effort to adapt their kernels, as shown in Listing 1. `bam::array`'s overloaded subscript operator enables the accessing threads to coalesce their accesses, query the cache, make I/O requests on misses, and returns the appropriate element of type T to the calling function. The `bam::array<T>` can also be used to develop higher-level abstractions that can transparently provide optimizations like cache line reference reuse so threads do not need to probe the cache more than once when accessing the same cache line multiple times. In contrast, the proactive tiling CPU-centric model requires full, non-trivial application rewrites to decompose the compute and data transfers into tiles that fit within the limited GPU memory.

```
__global__
void kernel(bam::array<float> data, size_t n,
bam::array<float> out, bam::array<int> randidx) {
size_t tid = ...;
...
for(; tid < n; tid += (blockIdx.x * blockDim.x))
        out[tid] = data[randidx[tid]];
};
```

Listing 1: Example GPU kernel with `bam::array<T>`.

BaM initialization requires allocating a few internal data structures that are reused during the application's lifetime. Initialization can happen implicitly through a library construction if no customization is needed. Otherwise, the application specializes the memory through template parameters to BaM's initialization call, a standard practice in C++ libraries. However, in most cases, specialization and fine-tuning are unnecessary, as we show later in § 5 where only BaM's default parameters are used.

4 BAM PROTOTYPE

We use off-the-shelf hardware including NVIDIA GPUs and arrays of NVMe SSDs to construct a BaM prototype and show the benefits of allowing GPUs to directly access storage with enough random access bandwidth to take full advantage of a GPU's PCIe Gen4 x16 link. *Once this level of data access bandwidth is achieved, a storage-based solution is as good as a host memory accessed through PCIe in terms of performance but much cheaper.* For simplicity, we describe the prototype assuming bare-metal, direct access to the NVMe SSDs.

4.1 Enable Direct NVMe Access From GPU Threads

For simplicity, we will use the NVMe SSD controllers as a simple example of storage controllers to explain the key features of the BaM prototype. In order to enable GPU threads to directly access data on NVMe SSDs we need to: 1) move the NVMe queues and I/O buffers from the host CPU memory to the GPU memory and 2) enable GPU threads to write to the queue doorbell registers in the NVMe SSD's BAR space. To this end, we create a custom Linux driver that creates a character device per NVMe SSD in the system, like the one by SmartIO [38]. Applications use BaM APIs to open the character device for each SSD they wish to use.

In the custom Linux driver, BaM leverages GPUDirect RDMA APIs to pin and map NVMe queues and I/O buffers in the GPU

Table 1: BaM prototype system specification

BaM Config	Specification
System	Supermicro AS-4124GS-TNR
CPUs	2× AMD EPYC 7702 64-Core Processors
DRAM	1TB Micron DDR4-3200
GPU	NVIDIA A100-80GB PCIe
PCIe Expansion	H3 Platform Falcon-4016 [21]
SSDs	Refer to Table 2
Software	Ubuntu 20.04 LTS, NVIDIA Driver 470.82, CUDA 11.4

memory. This enables the SSD to perform peer-to-peer data reads and writes to the GPU memory.

We leverage GPUDirect Async [42] to map the NVMe SSD doorbells to the CUDA address space so GPU threads can ring the doorbells on demand. This requires the SSD's BAR space to be first memory-mapped into the application's address space. Then it is mapped to CUDA's address space with the `cudaHostRegister` API. Other storage systems can be enabled similarly.

4.2 Scalable Hardware

Scaling BaM using the PCIe slots available within a data-center grade 4U server is challenging as the number of PCIe slots available in these machines is limited. For instance, Supermicro AS-4124 system has five PCIe Gen4 ×16 slots per socket. If a GPU occupies a slot it can only access 4 ×16 PCIe devices without crossing the inter-socket fabric and suffering significant performance degradation. However, as no single NVMe SSD can match the throughput of a PCIe x16 Gen4 link, the BaM hardware must scale the number of NVMe devices to provide the necessary throughput and match the bandwidth of the GPU's ×16 PCIe Gen4 interconnect.

To address this, we built a custom prototype machine for the BaM architecture using the off-the-shelf components as shown in Table 1. The BaM prototype uses a PCIe expansion chassis with custom PCIe topology for scaling SSDs. The PCIe switches provide low-latency, high-throughput peer-to-peer access. The expansion chassis has two identical drawers that can be connected independently to the host. Each drawer supports 8 ×16 PCIe slots. We use one ×16 slot in each drawer for an NVIDIA A100 GPU and the rest of the slots are populated with different types of SSDs. Currently, each drawer can only support 10 U.2 SSDs as they take significant space. With PCIe bifurcation, a multi-SSD riser card can enable more than 16 M.2 SSDs per drawer.

SSD Technology trade-offs: Table 2 lists the metrics that significantly impact the design, cost and efficiency of BaM systems for three types of off-the-shelf SSDs. The `$/GB` is based on the current list price per device, the expansion chassis, and the risers needed to build the system. A comparison of these metrics across SSD types shows that the consumer grade NAND Flash SSDs are inexpensive with more challenging characteristics, while the low-latency drives such as Intel Optane SSD and Samsung Z-NAND are more expensive with desirable characteristics. For example, for write intensive applications using BaM, Intel Optane drives provide the best througput and endurance.

Irrespective of the underlying SSD technology, as shown in Table 2, the BaM prototype provides 4.3-21.8× advantage in cost-per-GB, even with the expansion chassis and risers, over a DRAM-only solution. Furthermore, this advantage grows with additional capacity added per device, which makes BaM highly scalable as SSD capacity and application data size increase.

4.3 BaM Raw Throughput

We establish that BaM can generate sufficient I/O requests to saturate the underlying storage system by measuring raw throughput of BaM using microbenchmarks with Intel Optane SSDs and an NVIDIA A100 GPU. We allocate all the available SSD SQ/CQ queues into the GPU memory with a queue depth of 1024. We then launch a CUDA kernel with each thread requesting a random 512-byte block from the SSD via a designated queue. The requests are uniformly distributed across all queues with round-robin scheduling. We vary the number of threads and SSDs mapped to the GPU. For multiple SSDs, the requests are uniformly distributed across SSDs using round-robin scheduling. We measure I/O operations per second (IOPs) as the ratio of the number of requests submitted and the kernel execution time.

Results: Figure 4 presents the measured IOPs for 512B random read and write benchmarks. BaM can reach peak IOPs per SSD and linearly scale with additional SSDs for both reads and writes. With a single Optane SSD, BaM only requires about 16K-64K GPU threads to reach near peak IOPs (see Table 2). With ten Optane SSDs, BaM achieves 45.8M random read IOPs and 10.6M random write IOPs, the peak possible for 512B accesses to the Intel Optane SSDs. This is 22.9GBps (90% of the measured peak bandwidth for Gen4 ×16 PCIe links) and 5.3GBps of random read and write bandwidth, respectively. Further improvements in write bandwidth, which isn't yet hitting PCIe limits, can be achieved by scaling to more SSDs. Similar performance and scaling is observed with Samsung SSDs and also at 4KB access sizes but are not reported here due to space constraints. *These results validate that BaM's infrastructure software can match the peak performance of the underlying storage system.*

4.4 Discussion

GPUDirect RDMA I/O Consistency: As prior works[43, 61] have noted, when a third-party device writes into the GPU's memory with GPUDirect RDMA over PCIe, without a PCIe read following the writes, the order of these writes might not be preserved from the viewpoint of the GPU threads running concurrently. In the BaM prototype, we allow a GPU thread to submit a second I/O request after successfully polling for the first one's completion. As the storage device must read a submitted request over PCIe before writing the completion entry for it, when the thread finds the completion entry for the second command, it knows a PCIe read has been completed after all the writes for the first command. However, doing so incurs a 100% performance overhead.

To reduce the total number of additional I/O requests submitted, we implement a shared global virtual queue with a `lock`. All threads who find the CQ entries for their commands race for the `lock` of the virtual queue, and the winner enters the BaM I/O stack to submit an additional I/O request on behalf of all competing threads whose request can be coalesced with its own. After the winner finds the completion entry for the second request, it notifies the threads whose second requests were coalesced and releases the `lock`. Other threads continuously poll to see if their respective second requests are covered through coalescing. If not they try to

Table 2: Comparison of different types of SSDs with DRAM DIMM. Prices are taken from a well known retailer [13] and near the time of writing this paper. The cost of required PCIe expansion hardware [21] is included for systems using SSDs.

Technology	Sources	Product	RD IOPs (512B, 4KB)	WR IOPs (512B, 4KB)	Latency (μs)	DWPD	$/GB	Gain
DRAM	multiple	DIMM (DDR4)	>10M	>10M	O(0.1)	>1000	11.13	1.0×
Optane [23]	single	Intel P5800X	5.1M, 1.5M	1M, 1.5M	O(10)	100	2.54	4.4×
Z-NAND [55]	single	Samsung PM1735	1.1M, 1.6M	351K, 351K	O(25)	3	2.56	4.3×
NAND Flash [54]	multiple	Samsung 980pro	700K-800K, 700K-800K	172K, 172K	O(100)	0.3	0.51	21.8×

obtain the `lock` again. We find that this solution incurs only less than 8% performance overhead.

Programming Model: BaM provides the GPU threads a memory-like abstraction, with the same memory consistency properties as the GPU memory. If the application threads read and write to the same words in BaM-backed memory, then it is up to the application to implement the necessary synchronization to avoid races, just as if the threads were reading and writing to the GPU memory.

BaM supports writes at all levels of its software stack: block write I/O requests, tracking dirty cache lines, and the write method in the high-level abstractions. When the high-level abstraction's write method is called, the access to the cache line's state will set the dirty bit. The BaM cache is a write-back cache and has APIs for the user to flush a specific or all dirty cache lines.

If the system crashes in the middle of a GPU kernel that writes, there are no guarantees unless the application itself takes the responsibility of check-pointing application state and data, which is common practice in GPU accelerated applications.

CPU-GPU Data Sharing: If the application instantiates a BaM cache in the GPU memory and another BaM cache in the host CPU memory, it is up to the application to implement the appropriate synchronization and communication to maintain a consistent view of shared data, just like when the application uses GPU memory without BaM.

5 EVALUATION

BaM excels on applications that have data-dependent data access patterns. Data dependent access patterns are widely used in popular applications, e.g. graph analytics, and frameworks, e.g. RAPIDS for accelerating analytical queries. In this section, we present an evaluation of the prototype BaM system using these two workloads with a variety of datasets and show that a) BaM's performance is either on-par with or outperforms the state-of-the-art solutions (see § 5.2 and § 5.3). b) BaM's design is agnostic to the SSD storage medium used, enabling application-specific cost-effective solutions. c) BaM reduces I/O amplification and CPU orchestration overhead significantly for data-analytics workloads (see § 5.3).

5.1 Comparison With GDS and ActivePointers

We evaluate BaM's performance benefits over NVIDIA GDS [47] for different I/O block sizes, ranging from 4KB to 1MB. For GDS, we use `fio` to benchmark sequential access performance to transfer 128GB of data from four SSDs to GPU memory with 16 CPU threads. In the case of BaM, each warp is assigned a cache-line, the same size as the I/O blocks used for GDS, and consecutive warps access consecutive cache-lines in the 128GB dataset. Recall that in BaM the cache-line size defines the I/O access granularity. As shown in

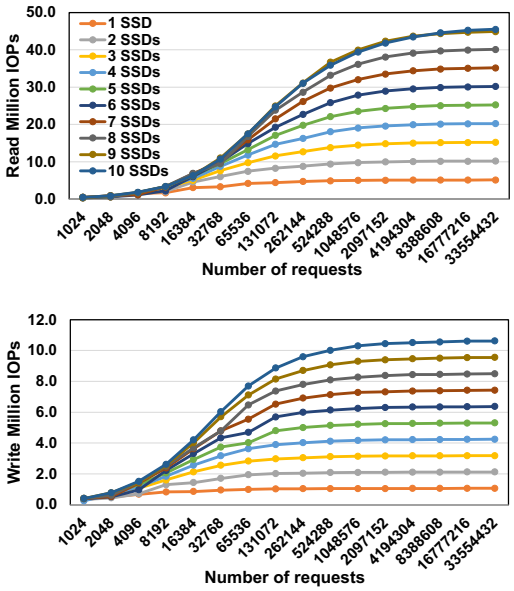


Figure 4: 512B random read (top) and write (bottom) benchmark scaling with BaM on Intel Optane P5800X SSDs. BaM's I/O stack can reach peak IOPs per SSD and linearly scale for random read and write accesses.

Figure 5, GDS can only saturate the GPU's PCIe link at the large I/O granularity of 32KB and only reaches 23.6% of the PCIe bandwidth at 4KB. Regardless of the number of CPU threads used, GDS is limited by the high overhead incurred by the Linux software stack. In contrast, BaM easily achieves 25GBps using four SSDs, which is the measured peak bandwidth of the GPU's PCIe link.

Next, we compare BaM and ActivePointers [57]. For this evaluation, a warp is assigned to read 1024 contiguous 8-byte elements in a file where threads access the elements in a coalesced manner. With ActivePointers, the file is pinned in the Linux page cache in CPU memory, favoring ActivePointers as *any misses in the ActivePointers cache only require data transfers from CPU memory to GPU memory, avoiding storage I/O requests and latencies.* For BaM, the data is kept on four SSDs from where it is requested in case of a miss in the BaM cache, incurring storage access latency. Due to space constraints, we only show the results for 64K and 1 Million threads in Figure 6; measurements with up to 32 Million threads show similar trends.

With a cold cache, ActivePointers' I/O performance is greatly hindered by the GPUfs [58] mechanism that GPU threads use to request the data transfers from the CPU, achieving only an effective

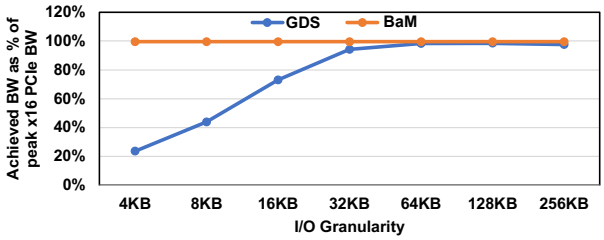


Figure 5: Performance of BaM compared with NVIDIA GDS [47]. With granularities less than 32KB, GDS is not able to saturate the PCIe interface due to the overheads of the traditional CPU software stack. In contrast, BaM is able to saturate the interface (~25GBps) at even 4KB I/O granularity.

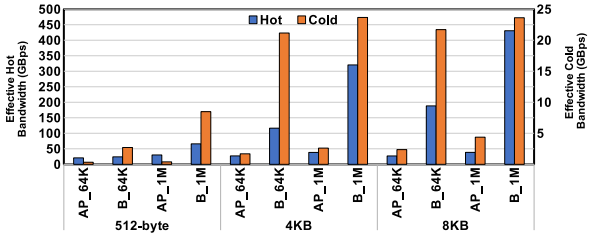


Figure 6: Performance of BaM (B) and ActivePointers+GPUfs [57, 58] (AP) with 64K and 1Million GPU threads with a 8GB cache in the GPU memory, in both hot and cold state, using 512-byte, 4KB, and 8KB cache-line sizes. BaM provides a peak miss-handling throughput of 17MIOPs with 4 Optane SSDs, which is 20.7× higher than ActivePointers' peak miss-handling throughput with the fast CPU memory. BaM's provides a peak hot cache bandwidth of 430GBps, 11.2× higher than ActivePointers' cache.

Table 3: Graph Analytics Datasets.

Graph	Num. Nodes	Num. Edges	Size (GB)
GAP-kron (**K**) [7]	134.2M	4.22B	31.5
GAP-urand (**U**) [7]	134.2M	4.29B	32.0
Friendster (**F**) [68]	65.6M	3.61B	26.9
MOLIERE_2016 (**M**) [59]	30.2M	6.67B	49.7
uk-2007-05 (**Uk**) [10, 11]	105.9M	3.74B	27.8

bandwidth of 4.4 GBps for 8 KB data transfers out of CPU memory. ActivePointers provides a peak miss-handling throughput of 823 KIOPs, with 512-byte cache-lines. In contrast, BaM nearly saturates the GPU's PCIe link at around 24 GBps with 4 KB and 8 KB cache-lines. Even with 512-byte cache-lines, BaM achieves 85% of the peak throughput supported by the 4 SSDs, at 17 MIOPs. Furthermore, when both caches are hot, BaM can provide an effective bandwidth of up to 430 GBps, 11.2× more than ActivePointers' peak bandwidth. BaM outperforms ActivePointers by more than an order of magnitude in both cache miss handling and hit data delivery.

5.2 Performance Benefit for Graph Analytics

We evaluate the performance benefit of BaM for graph analytics applications. We use the graphs listed in Table 3 for the evaluation. K, U, F, M are the four largest graphs from the SuiteSparse Matrix collection [16] while the Uk is taken from LAW [9].

A goal of BaM is to provide competitive performance against the host-memory-based DRAM-only graph analytics solution. To this end, *optimistic* target baseline **T** allows the GPU threads to directly perform coalesced fine-grain access to the graph data stored in the host-memory [39]. As there is sufficient CPU memory for the input graphs used in this experiment, we can make a direct performance comparison between BaM and **T**.

We run two graph analytics algorithms, Breadth-first-search (BFS) and Connected Components (CC), on the target system and BaM with different SSDs listed in the Table 2. Porting the state-of-the-art GPU implementations for both applications [39] to BaM requires minimal code changes as described in §3.5. For BFS, we report the average run time after running at least 32 source nodes with more than two neighbors. We do not execute CC on the Uk dataset since CC operates only on undirected graphs. Finally, we fix the BaM software cache size to 8GB, the cache-line size to 4KB and unless explicitly stated, we use four Intel Optane SSD with 128 queue-pairs at 1024 depth.

Overall performance with Intel SSD: Figure 7 shows the performance of both the target system (**T**) and BaM with single (`B_1I`) and four (`B_4I`) Intel Optane SSDs. For the single SSD (`B_1I`) configuration, BaM is on average slower by 1.43× and 1.27× for BFS and CC workload, respectively. This is because of the limited storage throughput available for BaM with single SSD (×4 PCIe Gen4 interface).

Scaling the number of SSDs to four (`B_4I`) and replicating data increases the BaM's aggregate bandwidth and provides similar bandwidth as the ×16 Gen4 PCIe interface as in the target system. Comparing with the target system **T** with file-loading time, *BaM with four Optane drives provides on average 1.00× and 1.49× speedup on BFS and CC applications, respectively.*

In both workloads, BaM overlaps the SSD data transfers for some threads with the compute of other threads, fully utilizing PCIe Gen4 ×16 bandwidth while incurring much less I/O amplification. In contrast, the target system **T** needs to wait until the file is loaded into memory before it can offload the compute to GPU. The superior host-memory bandwidth of the **T** system cannot overcome the initial file loading latency. As a result, BaM achieves either on-par or higher end-to-end performance.

Compared to the single-SSD BaM configuration, the four-SSD configuration scales to 3.48× for BFS and 4× for CC. The main detractor of scaling for BFS is the `Uk` dataset. Many nodes in the `Uk` graph have tiny neighborlists, resulting in deep BFS traversal (>100+ iterations) with small frontiers. Consequently, the total number of overlapping I/O requests in each iteration of BFS is insufficient to tolerate the latency.

BaM performance breakdown: We now slice up the overall execution time of BaM to understand how each BaM component contributes to the overall execution time, as shown in Figure 7. We first load the entire dataset in the GPU-HBM memory and measure the execution time (`Compute` in green color). Next, we measured the

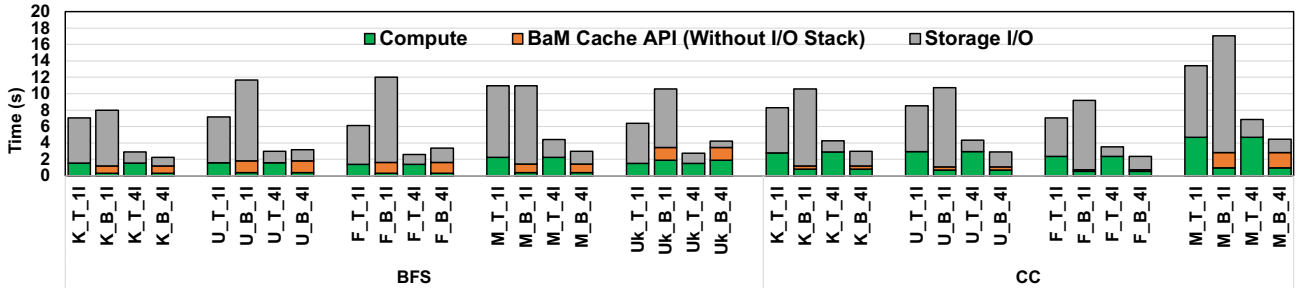


Figure 7: Graph analytics performance of BaM (CL size 4KB) and the Target (T) system with a single Intel Optane SSD. On average, BaM's end-to-end time is 1.0× (BFS) and 1.49× (CC) faster than the Target.

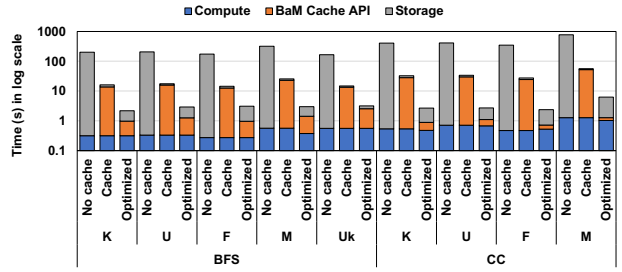


Figure 8: Sources of performance improvement in BaM. Adding a naive cache in BaM provides on average 11.9× (BFS) and 12.65×(CC) speedup over not having a cache. When the application leverages and exploits BaM's warp coalescing and reference reuse efficiently (`Optimized`), the application is sped up further by 6.07× (BFS) and 11.24× (CC) on average.

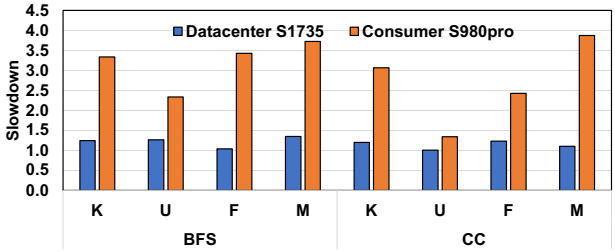


Figure 9: The slowdown observed by BaM with four Samsung DC PM1735 (S1735) and Samsung 980pro SSDs when compared with four Intel Optane SSDs.

total execution time when all the data is in the GPU-HBM memory, but each application access must go through the BaM cache API. This captures the best case performance one can achieve with BaM cache with no I/O requests. Subtracting `Compute` time from this measured execution time provides the cache API overhead (shown in orange color). Next, we constrain the BaM cache to 8GB and measure the total execution time. Storage I/O time (shown in grey color) can be computed by subtracting `Compute` and cache API time from this total time.

From Figure 7, we make the following key observations. First, the cache overhead is about 2-15% for a single SSD and goes to 4-45% for four SSDs. With a single SSD, the application performance is bounded by the storage throughput, and additional SSDs help to alleviate this problem by improving bandwidth, thus significantly minimizing the storage access overhead. The rest of the cache overhead is from cache metadata contention, long latency atomic operation, and warp scheduling in the face of polling threads. Even with this overhead, BaM outperforms the *most optimistic baseline system available.* Second, with four SSDs, BaM is still bounded by the storage I/O throughput (5-6.2M IOPs which is >80% of peak storage throughput), but there is an upper limit beyond which scaling SSDs will not help. This limit is bounded by the I/O request generation rate and how efficiently the applications utilize the cache. Further performance improvement requires application modification in work assignment/scheduling to trigger the BaM cache misses earlier during execution.

Sources of performance improvement: Figure 8 shows the sources of performance improvement with BaM. Using a naive cache (without warp coalescing or cache-line reference reuse) in BaM provides on average 11.9× (BFS) and 12.65× (CC) speedup over a no-cache implementation. This is because the BaM cache minimizes I/O amplification. When the application exploits BaM's warp coalescing and reference reuse efficiently, we observe an additional 6.07× (BFS) and 11.24× (CC) speedup on average.

Impact of SSD type: We now evaluate BaM with different types of SSDs: datacenter grade Samsung DC 1735 and consumer grade Samsung 980pro. Figure 9 shows the slowdown observed by BaM with each type using four SSDs when compared to Optane drives. Samsung DC 1735 and the Intel Optane SSD have similar performance for almost all workloads, since both achieve similar peak 4KB read IOPs. With the Samsung 980pro SSD, BaM prototype is on average 3.21× and 2.68× slower for BFS and CC workload. These results are very encouraging as the consumer-grade SSDs provide by far the (lowest cost) among all the SSD technologies.

Sensitivity Analysis: Next, we evaluate the effect of varying the cache size and the number of I/O queue pairs on the application performance with BaM. We limit our discussions to the K dataset as similar trends are observed for other datasets.

Cache Size: As shown in Figure 10, even with a 1GB cache, BaM does not experience any performance degradation as the cache can still capture the same level of spatial and temporal locality as 8GB.

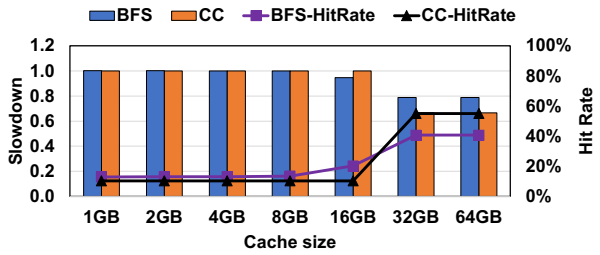


Figure 10: Impact of BaM cache capacity for K dataset relative to an 8GB cache. BaM maintains the same performance with a 1GB cache as with a 8GB cache.

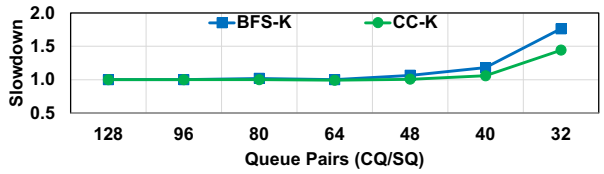


Figure 11: Impact of the number of NVMe SQ/CQ queue pairs on performance for K dataset relative to 128 queue pairs. BaM maintains the same performance as the number of queue pairs decreases and only starts degrading with 40 queue pairs.

Increasing the cache size to 32GB and 64GB enables the BaM cache to keep the entire working-set and only incur the cold cache misses.

Number of Queue Pairs: As shown in Figure 11, the application performance holds up well as the number of queue pairs decreases and degrades only at 40 (or lower) queue pairs, when the queue contention and the NVMe protocol serialization required per queue start to impact performance.

5.3 I/O Amplification Benefit for Data Analytics

Next, we evaluate the performance benefit of the BaM prototype for enterprise data analytics workloads to illustrate BaM's benefits in reduced I/O amplification and reduced software overhead while working on large structured datasets. These emerging data analytics are widely used to interpret, discover or recommend meaningful patterns in data that is collected over time. Although data-analytics applications are currently a small subset of all GPU applications, it is worth noting that the market size for this workload was $205 billion in 2020 and $230 billion in 2021 [18].

Setup: For this evaluation we use the NYC taxi ride dataset [60] and six queries, to compare the performance of BaM against the state-of-the-art GPU accelerated data analytics framework, RAPIDSv21.12 [46]. The queries used for evaluation altogether answer the final query: "Q5: What is the average dollar/mile the driver makes for trips that are at least 30 miles?" We start with a scan of the distance column (Q0), and add the total cost (Q1), surcharges (Q2), hail fee (Q3), tolls (Q4), and taxes (Q5) for only the trips that are at least 30 miles to get the penultimate query. For RAPIDS, we pin the entire dataset file in the Linux CPU page cache, enabling RAPIDS to read the file contents directly from the CPU DRAM without issuing an I/O request to the storage. This captures

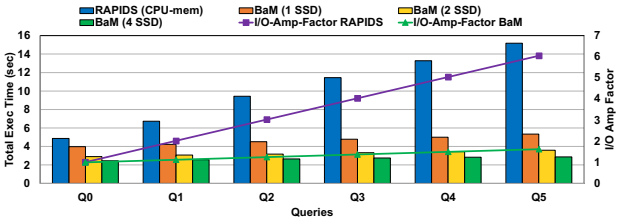


Figure 12: Performance of BaM and RAPIDS for data analytics queries on NYC Taxi dataset. BaM is up to 5.3× faster than the RAPIDS framework due to the reduced I/O amplification and reduced overhead in managing GPU memory.

the best performance modern systems can achieve. For evaluating BaM, we replicate data across SSDs, using up to four Intel Optane P5800X SSDs with 4KB cache-lines and 8GB cache capacity.

Results: Even with the single SSD, BaM outperforms RAPIDS performance for all queries as shown in Figure 12. For `Q0`, BaM observes a speed up of 1.22× over baseline even without any I/O amplification benefit. This is because, despite having the entire dataset preloaded into the CPU page cache, baseline RAPIDS experiences software overheads on the CPU to find and move data and manage the GPU memory.

With each additional data-dependent metrics (Q1 through Q5), the BaM performance advantage over RAPIDS increases as shown in Figure 12. The additional performance gain is attributed to BaM's reduced I/O amplification due to on-demand data fetching, while RAPIDS must transfer entire columns to the GPU memory. With additional data-dependent metrics, as shown in Figure 12, the baseline suffers from increased I/O amplification. In contrast, BaM's ability to make on-demand access to data as well as overlap compute, cache management, and many I/O requests helps it to handle multiple data-dependent columns nearly as efficiently as a single data-dependent column.

BaM's end-to-end application time scales by up to 1.46× with two SSDs and 1.62× with four SSDs when compared to a single SSD BaM configuration. The sub-linear scaling is due to BaM's setup overhead for pinning and mapping I/O queues and I/O buffers for DMA in GPU memory becomes more significant as more SSD's are used. Nevertheless, with four SSDs, BaM achieves up to 5.3× speed-up over the baseline.

5.4 VectorAdd Workload

In this section, we evaluate BaM on vectorAdd, a write-intensive workload. The vectorAdd workload takes two input arrays with four billion elements each, where each element is of eight-byte size to generate one output array of four billion elements. We assume the input vectors are in storage, and output requires to be written to the storage. For the baseline, we use a proactive tiling approach and split the four billion elements into five tiles. This allows the baseline to fully overlap the output write operation of the current tile with the loading of input vectors for the next tile. For BaM, the GPU kernel works on an entire dataset (no tiling), where each warp is assigned to a cache-line of the output vector.

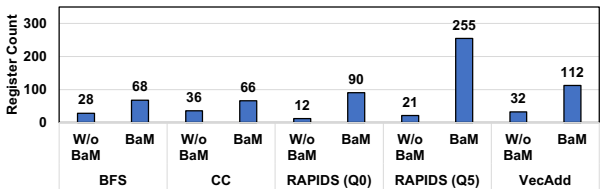


Figure 13: Per-thread register usage in applications. Even though BaM requires more registers per thread, all studied applications are storage I/O bound and register spilling or the reduced occupancy does not bottleneck performance.

BaM is 1.51× slower than the baseline implementation. BaM currently does not support overlapping between read miss handling and write-back activities, thus exposing the entire write latency to the application. We can address this by enabling asynchronous write-back in the BaM system, which we leave as future work.

5.5 SM Resource Utilization

Figure 13 shows the register usage per thread with and without BaM in all studied applications. Register spilling is observed in the RAPIDS workload. Neither BaM nor the applications use any shared memory. We do not enforce any occupancy constraints to limit register usage.

6 RELATED WORK

6.1 Optimized CPU-Centric Model

Most existing GPU programming models and applications were designed with the assumption that the working dataset fits in the GPU memory. If it doesn't, application specific techniques are employed to process large data on GPUs [8, 12, 19, 24, 29, 32, 47, 52, 56, 62, 63].

SPIN [8] and NVMMU [70] propose to enable peer-to-peer (P2P) direct memory access using GPUDirect RDMA from SSD to GPU and exclude the CPU from the data path. SPIN integrates the P2P into the standard OS file stack and enables page cache and read-ahead schemes for sequential reads. GAIA [12] further extends SPIN's page cache from CPU to GPU memory. Gullfoss [62] provides a high-level interface that helps in initializing and using GPUDirect APIs efficiently. Hippogriffdb [32] provides P2P data transfer capabilities to the OLAP database system. GPUDirect Storage [47] is a product that removes the CPU memory from the data path between SSDs and GPUs using GPUDirect RDMA technology. AMD has had similar efforts in RADEON-SSG products [3]. These works still require the CPU to orchestrate data movement. BaM allows any GPU thread to initiate data accesses to the SSDs.

6.2 Prior Accelerator-Centric Systems

ActivePointers [57], GPUfs [58], GPUNet [29], and Syscalls for GPU [64] have previously attempted to enable accelerator-centric model for data orchestration. GPUfs [58] and Syscalls for GPU [64] first allowed GPUs to request file data from the host CPU. ActivePointers [57] added a memory-map like abstraction on top of GPUfs to allow GPU threads memory-like access to file data. Dragon [37] incorporated storage access to the UVM [48] page faulting mechanism. However, as shown in § 5 and [50], these approaches use less-parallel CPUs to handle data demands from the massively parallel GPU and result in poor overall performance.

We also acknowledge the prior work in moving network control to the GPU [15], however storage presents a new set of challenges. With BaM, we enable GPUs to efficiently and directly access storage and show the performance and cost benefits for real applications.

6.3 Hardware Extensions

Prior work has proposed to replace or integrate GPU's global memory with non-volatile memories [65, 66, 71–73]. DCS [2] proposed enabling direct access between storage, network, and accelerators with an FPGA providing the required translation services for coarse-grain data transfers. Enabling persistence within the GPU has recently been proposed [6]. We acknowledge these efforts and further validate the need to enable large memory capacity for emerging workloads. More importantly, the BaM prototype aims to use emerging disaggregated storage hardware components to provide significant performance advantages to end-to-end applications with very large real-world datasets.

7 CONCLUSION

In this work, we make a case for enabling GPUs to orchestrate high-throughput, fine-grained accesses to storage, without the CPU software overhead, in a new system architecture called BaM. BaM mitigates the I/O amplification problem by allowing the GPU application compute code to read or write at finer granularities on-demand. As BaM supports the storage access control plane functionalities including caching, translation, and protocol queues on the GPU, it avoids the costly CPU-GPU synchronization, OS kernel crossing, and software bottlenecks that have limited the achievable storage access throughput. Using off-the-shelf hardware components, we built a prototype of BaM and show on multiple applications and datasets that BaM is a viable/superior alternative to DRAM-only and other state-of-the-art solutions.

ACKNOWLEDGMENTS

We would like to acknowledge all of the help from members of the IMPACT research group, the IBM-Illinois Center for Cognitive Computing Systems Research (C3SR) and NVIDIA Research without which we could not have achieved any of the above reported results. Special thanks to Kun Wu from the IMPACT research group for his suggestion on using advanced warp primitives for implementing fast warp coalescing. We would also like to acknowledge the valuable insight and help we received from stake holders including NVIDIA, Intel, Samsung, Phison, H3 Platform, Broadcom, and University of Illinois. Especially discussions with Yaniv Romem, Brian Pan, Jean Chou, Jeff Chang, Yt Huang, Annie Foong, Andrzej Jakowski, Allison Goodman, Venkatram Radhakrishnan, Max Simmons, Michael Reynolds, John Rinehimer, In Dong Kim, Young Paik, Jaesung Jung, Yang Seok Ki, Jeff Dodson, Raymond Chan, Hubertus Franke, Paul Crumley, Sanjay Patel, Deming Chen, Josep Torrellas, David A. Padua and several others have been fundamental for this work to come to fruition. This work uses GPUs donated by NVIDIA and is partly supported by the IBM-ILLINOIS C3SR and by the IBM-ILLINOIS Discovery Accelerator Institute (IIDA).

A ARTIFACT APPENDIX

A.1 Abstract

BaM is a novel system architecture that defines mechanisms for GPU threads to efficiently initiate and orchestrate storage accesses. As such, with the artifact, i.e., a prototype implementation of BaM, the authors demonstrated the following: (1) BaM is functional with a single SSD and provides the contributions described in the paper, (2) BaM is functional with multiple SSDs - up to 2 SSDs can be tested with the provided system, (3) BaM is functional with different types of SSDs - consumer-grade Samsung 980 Pro and datacenter-grade Intel Optane SSDs available in the provided system, (4) BaM works with different applications - microbenchmarks and graph applications (with real datasets) that are provided for artifact evaluation. Based on these demonstrations, this paper has been awarded the Artifact Available and Artifact Functional badges.

A.2 Artifact Checklist (Meta-information)

- **Compilation:** For the artifact evaluation reviewers, the authors provided access to a machine where the required compilation tool chains was already set up. For general details, refer to software dependencies (§A.3.3).
- **Dataset:** As the dataset size is over 500GB, the datasets were preloaded on the SSDs in the provided system for the artifact evaluation reviewers. For general details refer to (§A.3.4).
- **Run-time environment:** Linux Kernel 5.8.x. CUDA, etc. Refer to software dependencies (§A.3.3) for details. Root access is required.
- **Hardware:** For the artifact evaluation reviewers, the authors provided access to a machine where the required hardware was already set up. For general details refer to hardware dependencies (§A.3.2).
- **Run-time state:** Yes
- **Execution:** Only one artifact evaluation reviewer can execute the evaluation applications at a time on the provided machine. 2-3 hours aggregate run time.
- **Metrics:** Execution time, bandwidth and IOPS.
- **Output:** Console. See provided log files for expected output.
- **Experiments:** Follow the instructions at https://github.com/ZaidQureshi/bam/tree/master/asplosaoe#readme
- **How much disk space required (approximately)?:** >500GB.
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes
- **How much time is needed to complete experiments (approximately)?:** 3 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** BSD-2-Clause
- **Archived at:** https://doi.org/10.5281/zenodo.7217356 [51]

A.3 Description

A.3.1 Access. BaM's codebase is publicly available here: https://github.com/ZaidQureshi/bam.git

A.3.2 Hardware dependencies. Refer to https://github.com/ZaidQureshi/bam#hardwaresystem-requirements

A.3.3 Software dependencies. Refer to https://github.com/ZaidQureshi/bam#system-configurations

A.3.4 Datasets. Refer to https://github.com/ZaidQureshi/bam#example-applications

```
# Building the BaM library and applications
$ git submodule update --init --recursive
$ mkdir -p build; cd build
$ cmake ..
$ make libnvm -j      # builds library
$ make benchmarks -j # builds benchmark program

# Building the BaM kernel module
$ cd module
$ make -j
```

Listing 2: Building BaM library, applications and kernel module.

```
$ dmesg | grep nvme0
[126.497670] nvme nvme0: pci function 0000:65:00.0
[126.715023] nvme nvme0: 40/0/0 default/read/poll queues
[126.720783] nvme0n1: p1
[190.369341] EXT4-fs (nvme0n1p1): ...

# Unbind the SSD from kernel NVMe driver
$ echo -n "0000:65:00.0" >
   /sys/bus/pci/devices/0000\:65\:00.0/driver/unbind

# Load the BaM driver.
$ cd build/module
$ sudo make load
```

Listing 3: Identifying SSD to use and binding it to the BaM driver.

```
$ sudo ./bin/nvm-block-bench --threads=262144 --blk_size=64 --
      reqs=1 --pages=262144 --queue_depth=1024  --page_size=512 --
      num_blks=2097152 --gpu=0 --n_ctrls=1 --num_queues=128 --
      random=true
```

Listing 4: Basic test.

A.4 Installation

The complete installation instructions can be found at the following link https://github.com/ZaidQureshi/bam/blob/master/asplosaoe/README.md#compiling. We briefly describe them in this section.

A.4.1 Building the Project. From the project root directory, follow the steps shown in Listing 2. The CMake configuration is supposed to auto-detect the location of CUDA, Nvidia driver and project library. After this, we should also compile the custom `libnvm` kernel module for NVMe devices as shown in the last two lines in Listing 2.

A.4.2 Loading/Unloading the Kernel Module. In order to use the custom kernel module for the NVMe device, we need to unbind the NVMe device from the Linux NVMe driver. To do this, first we need to find the PCIe ID of the NVMe device. If the required NVMe device want is mapped to the `/dev/nvme0` block device, Listing 3 shows how to find its PCIe ID.

Next, we need to unbind the SSD from the Linux NVMe driver, as shown in Listing 3. Repeat the process for all NVMe devices that are to be remapped to the BaM driver. We next load the BaM kernel module from the build directory. This should create a `/dev/libnvm*` device file for each NVMe SSD that is not bound to the NVMe driver.

A.4.3 Basic Test. The evaluator can run the command available at https://github.com/ZaidQureshi/bam/blob/master/asplosaoe/README.md#running-the-io-stack-component (also shown in Listing 4) as a basic test. The expected output (sans performance numbers) can be found at https://github.com/ZaidQureshi/bam/blob/master/asplosaoe/nvm_block_bench_1_sam.log.

A.5 Evaluation

A.5.1 Experimental workflow. Each benchmark is an executable compiled in the process described earlier. Each benchmark has various command line parameters (explained in its help) and outputs relevant performance metrics like time and throughput.

A.5.2 Evaluation and expected results. Evaluation should validate the goals that are defined in the artifact evaluation abstract. For each of the goals, we have prepared a set of commands, their descriptions and expected results at https://github.com/ZaidQureshi/bam/tree/master/asplosaoe#readme.

REFERENCES

[1] B. Acun, M. Murphy, X. Wang, J. Nie, C. Wu, and K. Hazelwood. 2021. Understanding Training Efficiency of Deep Learning Recommendation Models at Scale. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA'21)*. IEEE Computer Society, Los Alamitos, CA, USA, 802–814.

[2] Jaehyung Ahn, Dongup Kwon, Youngsok Kim, Mohammadamin Ajdari, Jaewon Lee, and Jangwoo Kim. 2015. DCS: A fast and scalable device-centric server architecture. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Association for Computing Machinery, New York, NY, USA, 559–571.

[3] AMD. 2021. RADEON-SSG API Manual. https://www.amd.com/system/files/documents/ssg-api-user-manual.pdf.

[4] Jens Axboe. 2020. Efficient IO with io_uring.

[5] BaM 2022. BaM GitHub Repository. https://github.com/ZaidQureshi/bam.

[6] Ardhi Wiratama Baskara Yudha, Keiji Kimura, Huiyang Zhou, and Yan Solihin. 2020. Scalable and Fast Lazy Persistency on GPUs. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. 252–263.

[7] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619 http://arxiv.org/abs/1508.03619

[8] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. 2017. SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 167–179.

[9] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: a scalable fully distributed Web crawler. *Software: Practice and Experience* 34, 8 (2004), 711–726.

[10] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM Press, 587–596.

[11] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[12] Tanya Brokhman, Pavel Lifshits, and Mark Silberstein. 2019. GAIA: An OS Page Cache for Heterogeneous Systems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 661–674.

[13] CDW 2022. CDW. https://www.cdw.com.

[14] F. J. Corbato. 1968. A Paging Experiment With The Multics System. *Technical Report, Massachusetts Institute of Technology, Cambridge, Project MAC* (1968).

[15] Feras Daoud, Amir Watad, and Mark Silberstein. 2016. GPUrdma: GPU-Side Library for High Performance Networking from GPU Kernels. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers* (Kyoto, Japan) *(ROSS '16)*. Association for Computing Machinery, New York, NY, USA.

[16] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1 (Dec. 2011), 25 pages.

[17] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and Protection in the ZoFS User-Space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, 478–493.

[18] Fortune Business Insights. 2021. Big Data Analytics Market | 2021 Size, Growth Insights, Share, COVID-19 Impact, Emerging Technologies, Key Players, Competitive Landscape, Regional and Global Forecast to 2028. https://tinyurl.com/2p8a8sbx.

[19] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. 2010. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, Pennsylvania, USA) *(ASPLOS XV)*. Association for Computing Machinery, New York, NY, USA, 347–358.

[20] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. 2020. The Architectural Implications of Facebook's DNN-Based Personalized Recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 488–501.

[21] H3 Platform 2022. H3 Platform. https://www.h3platform.com.

[22] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. 2021. OGB-LSC: A Large-Scale Challenge for Machine Learning on Graphs. *arXiv preprint arXiv:2103.09430* (2021).

[23] Intel. 2021. Intel® Optane™ Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[24] Thomas B. Jablin, James A. Jablin, Prakash Prabhu, Feng Liu, and David I. August. 2012. Dynamically Managed Data for CPU-GPU Architectures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. Association for Computing Machinery, New York, NY, USA, 165–174.

[25] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 494–508.

[26] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. 2018. Designing a True Direct-Access File System with DevFS. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA.

[27] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. Hyperloop: Group-Based NIC-Offloading to Accelerate Replicated Transactions in Multi-Tenant Storage Systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA.

[28] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. 2021. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 756–771.

[29] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. 2014. GPUnet: Networking Abstractions for GPU Programs. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 201–216.

[30] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations (ICLR'17)*.

[31] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Shanghai, China.

[32] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. *Proceedings of the VLDB Endowment* 9, 14 (Oct. 2016), 1647–1658.

[33] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* 28, 2 (March 2008), 39–55.

[34] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. Scale and Performance in a Filesystem Semi-Microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 819–835.

[35] Jay Lofstead, Ivo Jimenez, Carlos Maltzahn, Quincey Koziol, John Bent, and Eric Barton. 2016. DAOS and Friends: A Proposal for an Exascale Storage System. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 585–596.

[36] Vikram Sharma Mailthody. 2022. *Application Support And Adaptation For High-throughput Accelerator Orchestrated Fine-grain Storage Access*. Ph. D. Dissertation. University of Illinois Urbana-Champaign.

[37] Pak Markthub, Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Satoshi Matsuoka. 2018. DRAGON: Breaking GPU Memory Capacity Limits with Direct NVM Access. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, 13 pages.

[38] Jonas Markussen, Lars Bjørlykke Kristiansen, Pål Halvorsen, Halvor Kielland-Gyrud, Håkon Kvale Stensland, and Carsten Griwodz. 2021. SmartIO: Zero-Overhead Device Sharing through PCIe Networking. *ACM Transactions on Computing System* 38, 1–2, Article 2 (jul 2021), 78 pages.

[39] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen mei W. Hwu. 2020. EMOGI: Efficient Memory-access for Out-of-memory Graph-traversal In GPUs. *Proceedings of VLDB Endowment* 14 (2020), 114–127.

[40] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie Amy Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. 2021. Software-Hardware Co-design for Fast and Scalable Training of Deep Learning Recommendation Models.

[41] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019).

[42] Nvidia 2016. State of GPUDirect Technologies. https://on-demand.gputechconf.com/gtc/2016/presentation/s6264-davide-rossetti-GPUDirect.pdf.

[43] Nvidia 2019. How to make your life easier in the age of exascale computing using NVIDIA GPUDirect technologies. https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9653-how-to-make-your-life-easier-in-the-age-of-exascale-computing-using-nvidia-gpudirect-technologies.pdf.

[44] Nvidia 2020. NVIDIA DGX A100. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf.

[45] Nvidia 2020. NVIDIA Tesla A100 Tensor Core GPU Architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[46] Nvidia 2021. CUDA RAPIDS: GPU-Accelerated Data Analytics and Machine Learning. https://developer.nvidia.com/rapids.

[47] Nvidia 2022. GPUDirect Storage: A Direct Path Between Storage and GPU Memory. https://developer.nvidia.com/blog/gpudirect-storage/.

[48] Nvidia 2022. Unified Memory for CUDA Beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners.

[49] Zaid Qureshi. 2022. *Infrastructure to Enable and Exploit GPU Orchestrated High-Throughput Storage Access on GPUs.* Ph. D. Dissertation. University of Illinois Urbana-Champaign.

[50] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seung Won Min, Amna Masood, Jeongmin Park, Jinjun Xiong, CJ Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. 2022. GPU-Orchestrated On-Demand High-Throughput Storage Access in the BaM System Architecture. arXiv. https://arxiv.org/abs/2203.04910

[51] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelago, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, CJ Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. 2022. *GPU-Initiated On-Demand High-Throughput Storage Access in the BaM System Architecture.* https://doi.org/10.5281/zenodo.7217356 This zendo version has the ASPLOS AEC evaluation. As this project is in continuous development, the most updated version of the project can be accessed in the following link: https://github.com/ZaidQureshi/bam.git.

[52] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, 14 pages.

[53] Yujie Ren, Changwoo Min, and Sudarsun Kannan. 2020. CrossFS: A Cross-layered Direct-Access File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 137–154.

[54] Samsung. 2021. Samsung 980 PRO SSD. https://www.samsung.com/us/computing/memory-storage/solid-state-drives/980-pro-pcie-4-0-nvme-ssd-1tb-mz-v8p1t0b-am/.

[55] Samsung ZNAND 2021. Samsung Z-NAND Technology Brief. https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf.

[56] Hyunseok Seo, Jinwook Kim, and Min-Soo Kim. 2015. GStream: A Graph Streaming Processing Method for Large-Scale Graphs on GPUs. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Francisco, CA, USA) *(PPoPP 2015)*. Association for Computing Machinery, New York, NY, USA, 253–254.

[57] Sagi Shahar, Shai Bergman, and Mark Silberstein. 2016. ActivePointers: A Case for Software Address Translation on GPUs. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, 596–608.

[58] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating a File System with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 485–498.

[59] Justin Sybrandt, Michael Shtutman, and Ilya Safro. 2017. MOLIERE: Automatic Biomedical Hypothesis Generation System. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Halifax, NS, Canada) *(KDD '17)*. Association for Computing Machinery, New York, NY, USA, 1633–1642.

[60] The City of New York. 2021. TLC Trip Record Data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[61] Maroun Tork, Lina Maudlej, and Mark Silberstein. 2020. *Lynx: A SmartNIC-Driven Accelerator-Centric Architecture for Network Servers.* Association for Computing Machinery, New York, NY, USA, 117–131.

[62] Hung-Wei Tseng, Yang Liu, Mark Gahagan, Jing Li, Yanqin Jin, and Steven Swanson. 2015. Gullfoss : Accelerating and Simplifying Data Movement among Heterogeneous Computing and Storage Resources.

[63] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. 2016. Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, 53–65.

[64] Ján Veselý, Arkaprava Basu, Abhishek Bhattacharjee, Gabriel H. Loh, Mark Oskin, and Steven K. Reinhardt. 2018. Generic System Calls for GPUs. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. 843–856.

[65] Jeffrey S. Vetter and Sparsh Mittal. 2015. Opportunities for Nonvolatile Memory Systems in Extreme-Scale High-Performance Computing. *Computing in Science Engineering* 17, 2 (2015), 73–82.

[66] Bin Wang, Bo Wu, Dong Li, Xipeng Shen, Weikuan Yu, Yizheng Jiao, and Jeffrey S. Vetter. 2013. Exploring hybrid memory for GPU energy efficiency through software-hardware co-design. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. 93–102.

[67] Weka.io. 2021. WekaFS Architecture Whitepaper. https://www.weka.io/wp-content/uploads/files/2017/12/Architectural_WhitePaper-W02R6WP201812-1.pdf.

[68] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities based on Ground-truth. *CoRR* (2012).

[69] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. 2017. Spdk: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 154–161.

[70] Jie Zhang, David Donofrio, John Shalf, Mahmut T. Kandemir, and Myoungsoo Jung. 2015. NVMMU: A Non-volatile Memory Management Unit for Heterogeneous GPU-SSD Architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 13–24.

[71] Jie Zhang and Myoungsoo Jung. 2020. ZnG: Architecting GPU Multi-Processors with New Flash for Scalable Data Analysis. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 1064–1075.

[72] Jie Zhang and Myoungsoo Jung. 2021. Ohm-GPU: Integrating New Optical Network and Heterogeneous Memory into GPU Multi-Processors. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 695–708.

[73] Jie Zhang, Miryeong Kwon, Hyojong Kim, Hyesoon Kim, and Myoungsoo Jung. 2019. FlashGPU: Placing New Flash Next to GPU Cores. In *Proceedings of the 56th Annual Design Automation Conference 2019* (Las Vegas, NV, USA) *(DAC '19)*. Association for Computing Machinery, New York, NY, USA, 6 pages.

[74] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. 2020. Distributed Hierarchical GPU Parameter Server for Massive Scale Deep Learning Ads Systems. In *Third Conference on Machine Learning and Systems*.

Received 2022-07-07; accepted 2022-09-22

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Torpor: GPU-Enabled Serverless Computing for Low-Latency, Resource-Efficient Inference

Minchen Yu, *The Chinese University of Hong Kong, Shenzhen; and Hong Kong University of Science and Technology;* Ao Wang, *Alibaba Group;* Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, and Wei Wang, *Hong Kong University of Science and Technology;* Ruichuan Chen, *Nokia Bell Labs;* Dapeng Nie, Haoran Yang, and Yu Ding, *Alibaba Group*

https://www.usenix.org/conference/atc25/presentation/yu

This paper is included in the Proceedings of the 2025 USENIX Annual Technical Conference.

July 7–9, 2025 • Boston, MA, USA

ISBN 978-1-939133-48-9

Open access to the Proceedings of the 2025 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

Torpor: GPU-Enabled Serverless Computing for Low-Latency, Resource-Efficient Inference

Minchen Yu[†‡] Ao Wang[§] Dong Chen[‡] Haoxuan Yu[‡] Xiaonan Luo[‡] Zhuohao Li[‡]
Wei Wang[‡] Ruichuan Chen[*] Dapeng Nie[§] Haoran Yang[§] Yu Ding[§]
[†]*CUHK-Shenzhen* [‡]*HKUST* [§]*Alibaba Group* [*]*Nokia Bell Labs*

Abstract

Serverless computing offers a compelling cloud model for online inference services. However, existing serverless platforms lack efficient support for GPUs, hindering their ability to deliver high-performance inference. In this paper, we present `Torpor`, a serverless platform for GPU-efficient, low-latency inference. To enable efficient sharing of a node's GPUs among numerous inference functions, `Torpor` maintains models in main memory and dynamically swaps them onto GPUs upon request arrivals (i.e., late binding with model swapping). `Torpor` uses various techniques, including asynchronous API redirection, GPU runtime sharing, pipelined model execution, and efficient GPU memory management, to minimize latency overhead caused by model swapping. Additionally, we design an interference-aware request scheduling algorithm that utilizes high-speed GPU interconnects to meet latency service-level objectives (SLOs) for individual inference functions. We have implemented `Torpor` and evaluated its performance in a production environment. Utilizing late binding and model swapping, `Torpor` can concurrently serve hundreds of inference functions on a worker node with 4 GPUs, while achieving latency performance comparable to native execution, where each model is cached exclusively on a GPU. Pilot deployment in a leading commercial serverless cloud shows that `Torpor` reduces the GPU provisioning cost by 70% and 65% for users and the platform, respectively.

1 Introduction

The remarkable advances in machine learning (ML) and its widespread adoption in various domains have fueled a surging demand for cloud-based ML inference services [24, 32, 49, 65, 66]. Severless computing offers a compelling cloud model for inference serving [20, 58, 61]. In a serverless cloud, users publish ML models as inference functions, and delegate resource provisioning and scaling responsibilities to the cloud platform. Serverless computing is also economically appealing as users only pay for the resources consumed by their functions (i.e., pay-per-use billing), eliminating resource idling costs.

However, today's serverless computing platforms, such as AWS Lambda [6] and Alibaba Function Compute [1], lack efficient support for GPUs. They typically run an ML model in a container (or a microVM) and early bind it to a GPU before starting to serve requests. To avoid considerable startup overhead of on-demand GPU function provisioning (e.g., tens of seconds as shown in Table 1), an inference function is maintained as a long-lived, provisioned instance on a designated GPU to handle future requests [4, 7]. This approach essentially follows the "serverful" inference serving practice [47, 49, 65], requiring users to pay for the occupied GPUs even during function idling. Furthermore, our analysis in a production cloud demonstrates that inference functions exhibit varying request rates, with 85% functions being invoked no more than once per minute (Fig. 2). Early binding these functions to GPUs results in low utilization and imbalanced load across GPUs, making it inefficient for cloud providers.

We believe that an efficient serverless inference platform should provide four desirable properties. First, it should enable *pay-per-GPU-use billing* for users, with charges incurred only when the functions are invoked and running on GPUs. Second, the platform should achieve optimal GPU utilization through efficient *GPU sharing* for concurrent inference functions, minimizing resource provisioning costs for cloud providers. Third, the platform should be aware of the user-specified *latency SLOs* and strive to meet them for all inference requests, if feasible. Lastly, the platform should achieve the aforementioned three properties *without requiring detailed knowledge about inference models* due to intellectual property and business-critical confidentiality reasons. We notice that there have been several relevant systems developed in recent years [23, 32, 34, 36, 46, 47, 49, 58, 63], none of which, however, provide all of these properties for serverless inference. They often suffer from cost inefficiency, SLO violations, or necessitate model-specific knowledge (see §2.2 and §9).

In this paper, we present `Torpor`, a GPU-efficient serverless inference platform that achieves all four desirable properties and is readily-deployable onto real-world serverless platforms without intrusive changes. `Torpor` follows a late binding de-

sign principle, whereby idle inference models are maintained in host memory and dynamically swapped to GPUs upon request arrivals. Compared to GPU memory, host memory is less expensive and has a much larger capacity, making it an ideal storage for holding numerous idle functions. This approach naturally supports pay-per-GPU-use billing, as idle functions no longer occupy GPU resources. Furthermore, by dynamically swapping models from host to GPUs, it enables fine-grained GPU sharing among concurrent inference functions, substantially improving GPU utilization and load balancing across GPUs. Model swapping can also be efficiently performed through pipelined loading, yielding significantly lower latency compared to function cold starts. All these are achieved without detailed knowledge about inference models – a must-have in a commercial environment for intellectual property and confidentiality protection. These techniques, combined with intelligent request scheduling, enable the platform to optimize the SLO attainment for users.

Specifically, to realize late binding while being readily-deployable on real-world serverless platforms, `Torpor` leverages a GPU pooling architecture. In this design, each worker node manages a pool of local GPUs and allows its inference functions to access any of these GPUs freely through CUDA API redirection. This enables seamless model swapping within a GPU pool and is transparent to users. However, this approach also presents three key challenges.

First, GPU pooling and model swapping incur high communication overhead compared to native execution (i.e., executing a model directly on a GPU). To address this challenge, `Torpor` proposes *asynchronous API redirection* to avoid frequent synchronizations between the inference functions and the GPU pool, eliminating the high communication overheads for model inference. `Torpor` further utilizes *pipeline execution* to overlap the host-to-GPU model swapping and the inference execution, thereby reducing end-to-end latency. It also utilizes high-speed NVLink for fast model swapping between GPUs whenever feasible and beneficial. Combined with low-latency API redirection, `Torpor` can efficiently execute models on any available GPUs. `Torpor` is intentionally designed to be model-agnostic to meet the confidentiality requirements while being generally applicable to various models, including even large generative models where runtime states (e.g., KV cache) can be managed as part of the model.

The second challenge is that GPU pooling and model swapping necessitate an efficient GPU memory management system. `Torpor` designs such a system that automatically tracks the addresses of models as they are swapped across multiple GPUs and adjusts each memory access of CUDA APIs accordingly during inference execution. It also efficiently organizes and shares memory blocks to avoid high memory allocation overheads, improving the overall performance of model swapping. Additionally, `Torpor` offers two GPU runtime management modes—runtime sharing and runtime isolation—to meet various needs for resource efficiency and cross-model isolation.

The third challenge is that the platform should meet the latency SLOs for inference functions while maintaining low GPU costs. `Torpor` proposes three policies to achieve this objective. First, `Torpor` designs a request scheduling algorithm that minimizes model swapping overheads, resulting in reduced end-to-end inference latency. It categorizes models into two groups, heavy or light, based on whether these models incur high overhead during swapping via PCIe. `Torpor` then prioritizes NVLink over PCIe for transferring heavy models across GPUs, effectively reducing concurrent PCIe traffic. Second, `Torpor` globally manages GPU memory in the pool and leverages model heaviness to guide eviction. Together with request scheduling, this approach significantly minimizes model swapping overhead. Third, `Torpor` proposes an SLO-aware request queuing policy that prioritizes requests to functions that have a higher likelihood of meeting SLOs, effectively improving the SLO attainment.

We have implemented and evaluated `Torpor` through a pilot deployment in Alibaba Cloud [1], one of the world's largest commercial serverless platforms. Evaluation results show that `Torpor` achieves low-latency model inference, comparable with native executions. `Torpor` can share a single GPU across hundreds of inference functions and load-balance GPUs with model swapping, resulting in over 10× cost reduction compared with current GPU offering in Alibaba Cloud. With its efficient SLO-aware scheduling and queuing policies, `Torpor` can serve 480 functions on a 4-GPU worker node while achieving low tail latency and satisfying millisecond-scale SLOs for all functions. Cluster experiments further demonstrate that `Torpor` scales well with the number of inference functions at low resource cost and meets per-function latency SLOs for thousands of functions. `Torpor` has been beta-released in a pilot production cluster in Alibaba Cloud, saving 70% of user costs on average and 65% of GPU provisioning costs for Alibaba Cloud.

2 Background and Motivation

In this section, we first give an overview of serverless inference. We then describe the inefficiency of existing solutions to enabling GPUs in serverless platforms, and highlight four key requirements in this regard.

2.1 Serverless Inference

As a leading serverless platform with a global presence, our Alibaba Cloud has observed a growing adoption among enterprise customers who opt to deploy their inference services using serverless functions, known as *serverless inference*. In comparison to existing inference services based on a "serverful" cloud model, such as AWS SageMaker [5], serverless

[1] We have open-sourced `Torpor`'s single-node prototype at https://github.com/FCSLab/torpor.

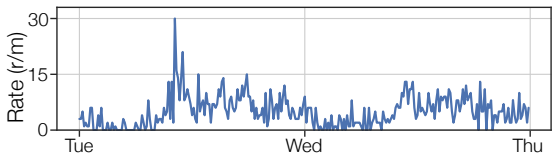


Figure 1: A two-day request trace of a typical GPU inference function in Alibaba Cloud.

inference significantly alleviates the burden of server management for cloud users. Specifically, the serverful approach requires users to manually configure various system-level parameters (e.g., VM types, GPUs, CPU cores, etc.) and manage resource provisioning (e.g., scaling the number of VMs up or down according to demand changes). In contrast, serverless inference enables users to simply publish models with inference code as functions, and then cloud providers automatically handle resource provisioning, autoscaling, scheduling, and fault tolerance. Furthermore, compared with the serverful approach, serverless inference also offers substantial cost savings as users do not pay for idle resources under the pay-per-use pricing model [20, 58, 61, 65]. In Alibaba Cloud, the requests to a function typically exhibit dynamic, bursty arrival patterns as shown in Fig. 1, consistent with previous research findings [24, 25, 32, 33, 41, 42, 47, 49, 67]. By leveraging the high elasticity of a serverless platform, inference functions can quickly scale in response to the changing workload, while users are billed based on the actual function runtime at a fine granularity, such as 1 ms [6, 8].

2.2 GPU Support in Serverless Platforms

Despite the benefits of the serverless inference model, existing serverless platforms, including Alibaba Cloud and other leading platforms, currently lack efficient support for GPUs, which impedes their ability to achieve high-performance serverless inference. Alibaba Cloud users also have expressed a compelling need to execute their models in GPU-enabled functions.

Existing solutions and their inefficiency. A number of recent systems have been proposed to support GPUs in serverless platforms [1, 27, 29, 58]. They, however, still follow the approach of existing serverful model serving systems (e.g., Nexus [49] and INFaaS [47]), and deploy inference models as long-running containers where each container, when created, is bound to a specific GPU (i.e., early binding). The deployed model remains in the memory of a designated GPU to handle future requests, and the occupied GPU resources can only be reclaimed after the model serving terminates.

However, the early-binding approach deviates from the serverless paradigm and is costly for both cloud users and providers. First, binding inference functions to GPUs occupies resources for extended duration, even when idling. Thus, users are obligated to pay for the allocated GPUs regardless of actual usage [3], leading to high expenses that undermine

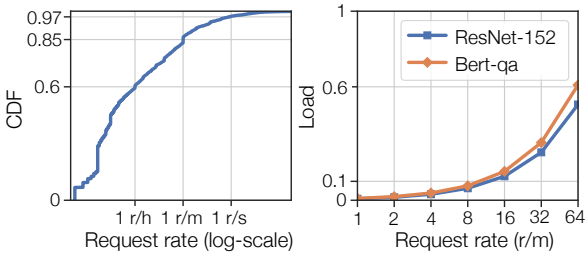


Figure 2: CDF of average function request rates from a one-week production trace (left) and the GPU load under various per-function request rates when running multiple functions on a `V100` GPU to saturate its 32 GB memory (right).

Table 1: Model startup times (s) under `Torpor` (runtime isolation mode in §4.5) and cold-starts. `Torpor`'s startup time is broken down into model loading and runtime resumption.

Model	Torpor		Cold-start	Mem. footprint
	Model	Runtime		
ResNet-152 [17]	0.03	0.26	8	1.6 GB
Bert-qa [26]	0.14	0.19	11	2.4 GB
Stable Diffusion [18]	0.24	1.5	25	5.1 GB
Llama3-8B [11]	1.6	1.4	48	13 GB
Qwen-14B [16]	2.1	1.5	57	20.1 GB
Llama2-13B [10]	2.5	1.9	61	24.5 GB

the cost-saving benefits of serverless inference. Second, this approach results in severe GPU underutilization, considering the low average request rates of most inference functions and the cross-GPU load imbalancing. Fig. 2 (left) depicts the distribution of the average request rates of Alibaba Cloud functions in a one-week trace, revealing that 85% (97%) of functions were invoked only once per minute (second) on average[2]. These findings align with the observations from other production traces [8, 48]. Fig. 2 (right) further illustrates that consolidating multiple models to fill GPU memory can still lead to low GPU load. Meanwhile, packing models into a GPU can cause temporary overloading due to the bursty request patterns (Fig. 1), thus inevitably leading to hotspots and load imbalancing in a multi-GPU setting. The impact of load imbalancing will be shown in Fig. 9 in §7.2.

To reduce costs, current systems need to frequently reclaim GPU resources when functions are inactive, avoiding charges for unused GPUs and allowing other functions to utilize idle resources. Unfortunately, this approach leads to frequent function cold starts, leading to significant overhead for model inference. Table 1 shows model startup times under cold starts, which need tens of seconds for GPU container setup, ML framework startup, GPU runtime creation, and model initialization[3]. Therefore, the cold-start overhead far exceeds the

[2]For confidentiality reasons, we depict the request rates of both CPU and GPU functions, which exhibit similar patterns (see Fig. 1).

[3]We exclude the delay of fetching a remote container image or a model file for cold starts, which can take extra seconds to minutes to complete [54]. A detailed discussion of `Torpor`'s performance is provided in §8.

Table 2: A comparison of Torpor and existing solutions that offer GPU support on serverless platforms.

Solution	GPU pay-per-use	GPU efficient	SLO compliant	Model agnostic
Alibaba Cloud [1]	×	×	×	✓
Molecule [27]	×	×	×	✓
DGSF [29]	×	×	×	✓
INFless [58]	×	×	*	×
Torpor	✓	✓	✓	✓

typical SLO requirement of model inference.

Requirements of serverless inference. Table 2 summarizes key requirements of serverless inference and compares Torpor with other existing solutions. Serverless users should be billed only when their functions are invoked and running on GPUs to achieve substantial cost savings (***pay-per-GPU-use***)[4]. Serverless platforms like Alibaba Cloud should serve as many inference functions as possible using a minimum number of GPUs, thereby attaining high GPU utilization (***GPU efficient***). The platform should allow users to specify their latency SLOs and strive to meet the latency SLOs for all functions (***SLO compliant***). For confidentiality reasons, the serverless platform should avoid inspecting detailed model structure, which can be of high business value (***Model agnostic***).

Compared with Torpor, none of existing solutions can meet all desired requirements. Alibaba Cloud and Alibaba Function Compute [1] are leading commercial serverless platforms with GPU supports; Molecule [27] introduces a serverless platform that supports GPUs and other hardware devices; DGSF [29] enables serverless functions to access GPUs in a remote cluster. These systems employ the early-binding approach as previously discussed, failing to enable pay-per-GPU-use billing and achieve high GPU efficiency. Moreover, they are oblivious to the semantics of model inference and unable to meet latency SLOs. INFless [58] presents a serverless inference system that early-binds functions to GPUs. While INFless proposes function scheduling and keep-alive schemes aimed at low-latency inference, it still leads to function cold starts and SLO violations (details in §7.3). Furthermore, INFless requires model knowledge for operator-level profiling. We leave more discussions on related work to §9.

3 Key Insight and Challenges

Key insight. As described in §2.2, the current early-binding approach of retaining inference models in GPU memory leads to high idling costs and underutilized resources. Therefore, an efficient serverless inference platform should enable *late binding*, where GPUs are managed as a resource pool and idle inference models reside in host memory, dynamically swapping into any available GPUs upon request. This approach should also be *easily deployable* on real-world serverless platforms without requiring intrusive changes. Late binding offers several key advantages in meeting the requirements in §2.2. **First**, keeping models in host memory eliminates GPU memory usage during idle periods, enabling pay-per-GPU-use billing and cost savings for cloud users. **Second**, host memory is significantly larger than GPU memory (e.g., a few TB vs. tens of GB), allowing for consolidation of multiple low-frequency functions onto a single GPU with improved GPU utilization. Late binding also facilitates load-balancing across multiple GPUs in a pool. **Third**, model swapping provides an efficient method to resume function execution compared to cold starts in the early-binding approach, thereby facilitating SLO compliance. **Finally**, late binding can be performed transparently to users within the GPU pool, which holds a holistic view of memory usage without requiring detailed model-specific knowledge.

Challenges. Implementing GPU pooling and late binding in the serverless platform presents three challenges. ***C1: Efficient GPU pooling and model swapping***. GPU pooling requires inference functions to synchronize with a remote GPU pool [28, 31], which introduces additional communication overhead compared to local executions and presents challenges in achieving low-latency inference. ***C2: GPU memory management.*** To enable seamless late binding, the platform should automatically monitor and manage memory usage without detailed model knowledge. This requires a unified and efficient GPU memory management system across the GPU pool. ***C3: SLO compliance and resource efficiency***. The platform should provide efficient request scheduling and model placement algorithms that effectively utilize the late binding mechanism to meet latency SLOs and enhance resource efficiency.

In the following sections, we present Torpor, a GPU-enabled serverless platform that addresses the aforementioned challenges and, importantly, is readily-deployable onto real-world serverless platforms without intrusive changes.

4 Torpor System Design

4.1 Architecture overview

Fig. 3 provides an overview of the architecture of Torpor, which comprises two main components: the cluster manager and worker nodes. The cluster manager handles cluster-level tasks, including request routing, node allocation, and resource scaling. It dynamically schedules function instances and routes inference requests to maintain load balancing across worker nodes and ensure fault tolerance (§ 4.5). At each worker node, Torpor employs GPU pooling, where a GPU server manages all local GPUs as a pool and allows functions to dynamically access any available GPUs. Within the GPU

[4] In our experiences, enterprise customers are willing to pay a nominal fee to retain idle functions in host memory for substantially improved performance (§8), similar to the function keep-alive charge meant to avoid cold starts [3, 7, 37].

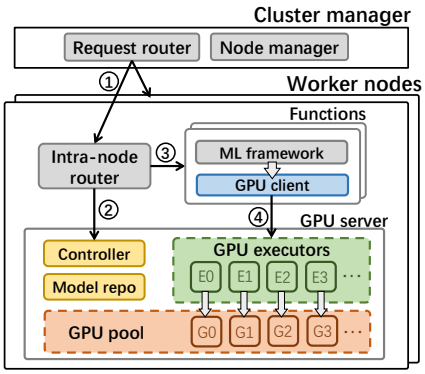


Figure 3: Architecture overview of Torpor. A request arriving at a Torpor cluster is first routed to a worker node hosting its target function ①. The router in the worker node synchronizes with the GPU server to query the executor for this request ②, and then routes it to the function instance with the target executor ID ③. The function instance next processes the request and uses a GPU client to automatically redirect CUDA API calls to this executor ④, and finally returns the result to the user after request completion.

server, a model repository manages models in host memory; GPU executors handle CUDA execution, perform necessary model swapping, and manage GPU memory on the associated GPUs; the controller maintains a global view of GPU memory and executor status, and decides how to schedule requests to executors. Additionally, each worker node runs an intra-node router to signal the GPU server about request arrivals and route requests to local inference functions. Once the target function receives a request, it interacts with the scheduled executor through a GPU client by remoting CUDA API calls. All components within the worker node—the GPU server, intra-node router, and functions—are deployed as containers.

Key to Torpor is to develop an efficient GPU server that enables low-latency model inference and addresses the challenges discussed in §3. Specifically, we will elaborate upon Torpor's designs to address the following questions: 1) how Torpor achieves low-latency GPU pooling (§4.2) and model swapping (§4.3); 2) how Torpor tracks the memory footprint of functions and manages GPU memory (§4.4); 3) how Torpor ensures isolation and handles failures (§4.5).

4.2 GPU Remoting

Asynchronous API redirection. Existing GPU remoting solutions [28, 45] introduce high communication overhead due to synchronizations for individual API calls during model inference (details in §7.1). Leveraging the computing pattern of model inference, Torpor proposes *asynchronous* API redirection to reduce synchronizations. Specifically, we observe that the intermediate steps in an inference execution are typically performed asynchronously on the GPU, where intermediate data is generated and consumed in GPU memory without requiring data transfer to the host until the execution completes. Consequently, a function can redirect intermediate CUDA calls to the GPU executor asynchronously without waiting for their results, and only perform synchronizations for the final output. This approach preserves the execution order of CUDA APIs, ensuring the correctness of the inference results.

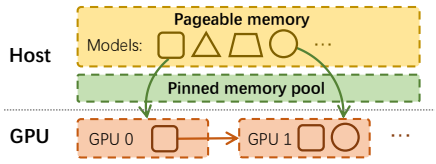


Figure 4: An example of model swapping. Models can be swapped from host to GPU through PCIe (green arrows), or across GPUs through NVLink (red arrow).

Following this insight, Torpor categorizes CUDA APIs into two groups based on their semantics: 1) *synchronous, blocking* APIs that require the host to await their completion before proceeding, such as cudaMalloc; and 2) *asynchronous, non-blocking* APIs that do not alter the host's runtime state, such as cudaLaunchKernel, which allows for asynchronous API redirection. Most APIs issued during inference fall into the asynchronous category, presenting opportunities to mitigate the synchronization overhead. Torpor can further batch consecutive CUDA API calls to enhance asynchronous API redirection (see APIs and batching details in our technical report [62]).

4.3 Model Swapping

As serverless platforms are constrained from examining the detailed model structures, it poses challenges to achieve seamless and efficient model swapping. Torpor overcomes the problem by leveraging two insights: 1) tracking general memory footprint of model inferences is feasible within a GPU pool, and 2) the memory access pattern of a model—the addresses and access order of model parameters—generally remains consistent across requests. Therefore, Torpor only tracks the first function execution (i.e., cold start), and applies the pattern to future request executions (see memory tracking in §4.4). Torpor performs model swapping on demand at the request level, and enhances performance through pinned memory pool and pipeline execution.

Model swapping and pipeline execution. Fig. 4 illustrates the model swapping in Torpor, where it utilizes pinned memory to enhance the host-to-GPU model loading and supports fast, cross-GPU model swapping via high-speed NVLink. Torpor's swapping can be generally applied to various types of models, including those that require maintaining runtime states (e.g., KV cache in LLMs) by treating these states as part of the model itself. Torpor further employs pipeline execution to enhance swapping performance, which is particularly effective for models computed in one forward pass, allowing the transmission of subsequent layers to overlap with the

computation of previous layers.

Key to pipeline execution is to judiciously group model parameters for swapping. Grouping too few parameters triggers a large number of transmissions and high synchronization overhead; conversely, grouping too many parameters impairs pipeline efficiency. Pipeline strategies in previous systems like PipeSwitch [22] and DeepPlan [36] do not suit serverless inference as they assume model structure is provided and require extensive model profiling. `Torpor` employs a model-agnostic approach to determine the group size for model pipelining. We observe that the transmission performance experiences an "elbow point" concerning group sizes: increasing the group size improves overall transmission throughput, but the improvement becomes marginal after a certain point. We therefore select this elbow point as the group size, which can achieve good swapping performance without substantially compromising pipeline efficiency. This group size depends only on hardware configurations such as PCIe, and can be easily determined by profiling various-sized data (e.g, about 2 MB in our testbed). This approach requires no detailed model knowledge and can be directly applied to various models.

Model eviction. We observe that model unloading from GPUs to host can result in considerable overhead and can interfere with concurrent inference executions. Therefore, `Torpor` always maintains a copy of the model in the host, and only invalidates its GPU memory region during eviction.

4.4 Memory Management

`Torpor` presents two key requirements for GPU memory management, as compared with other systems [12, 23, 44]. First, late binding requires a pool of GPUs to share the same logical memory space. This is because the inference functions do not recognize backend GPUs, and consistently access models using identical memory addresses even across different GPUs. Second, model swapping triggers frequently GPU memory (de)allocation, which leads to substantial overhead when using native methods like `cudaMalloc` (see Fig. 13). We therefore design a GPU memory management system that can effectively hide memory address differences between various GPUs and provide low-latency memory (de)allocations.

Memory address management. We observe ML frameworks like PyTorch typically organize data into blocks, each containing multiple parameters. `Torpor` leverages such memory layout to perform memory mapping at the block level. In particular, `Torpor` monitors memory blocks for each function and maintains a mapping to their actual physical addresses after model swapping. Since the internal data layout within each block remains unchanged (e.g., parameter offsets), `Torpor` can easily obtain the physical address of a parameter using its associated block address and offset. This approach eliminates the need for extensive metadata maintenance for individual data pointers, enabling the efficient address translation with low management overhead.

Memory block allocation. To mitigate the high overhead of native GPU memory allocation, `Torpor` reserves all GPU memory at bootstrap and internally manages memory blocks. This provides a shim layer to service memory requests from functions, without needing the native method. Key to this approach is to avoid memory fragmentation, which can decrease available GPU memory and harm overall efficiency. `Torpor` effectively addresses this issue by extending the Buddy memory allocation scheme [40] and leveraging unique characteristics of inference. It consolidates memory blocks from the same models to minimize fragmentation and enables sharing of common-sized blocks across different models (see details in our technical report [62]).

4.5 Isolation and Fault Handling

Resource isolation and GPU runtime management. `Torpor` provides container-level isolation for CPU and memory resources[5], similar to existing serverless platforms [1, 52, 60]. For GPUs, `Torpor` executes only one function on a GPU at a time and isolates GPU memory regions across functions. This is achieved through `Torpor`'s GPU server, which has full control over CUDA API execution and GPU memory access.

`Torpor` offers two isolation modes for GPU runtime: 1) runtime sharing, which runs a single runtime on a GPU for multiple models, for instance, in a more trusted environment, and 2) runtime isolation, which maintains a dedicated runtime for each model and suits better for a more untrusted environment. By default, `Torpor` employs runtime sharing to improve resource efficiency. When stricter isolation is necessary, `Torpor` can switch to runtime isolation mode. Indeed, in our pilot deployment (§8), `Torpor` runs in the runtime isolation mode. The overhead incurred by runtime isolation is generally acceptable, e.g., hundreds of milliseconds to 1.5 seconds as shown in Table 1, which is still over one order of magnitude latency improvement compared with cold starts.

Fault handling. `Torpor` sustains various system component failures. In case of function failures, `Torpor` restarts them to resume the execution. For executor failures or GPU runtime errors, `Torpor` migrates the affected models to other working GPUs (executors) via swapping, and then restarts the failed ones. When runtime isolation is employed, `Torpor` can ensure that buggy function executions do not affect others, achieving stronger fault isolation. The GPU server also persists runtime states (e.g., models and metadata) in local storage to allow fast recovery from an entire failure of the GPU server.

At the cluster level, `Torpor` persists metadata of individual nodes in a database, which enables the cluster manager to retain these states and recover from failures, aligning with current practices in Alibaba Cloud. It also keeps periodic health checks with the router on each worker node, and handles node

[5] `Torpor` makes no assumption on function sandboxes and can also support microVMs [19, 53].

Table 3: Latency (ms) of model pipelining execution when concurrently swapping other models through PCIe. The diagonal values indicate the latencies without concurrent models.

Model	DenseNet-169	ResNet-152	Bert-qa
DenseNet-169	**27**	27 (+0%)	27 (+0%)
ResNet-152	31 (+7%)	**29**	43 (+48%)
Bert-qa	166 (+11%)	240 (+61%)	**149**

failures by launching a new node and migrating all relevant functions.

5 `Torpor` Policy Design

We present how `Torpor` meets the latency SLOs and delivers resource efficiency (i.e., Challenge *C3* in §3). We start with the design overview, followed by individual policies.

5.1 Design Overview

Objective. The objective of `Torpor`'s policy design is to meet latency SLOs for inference functions while minimizing the resource cost. We define a function to comply with latency SLOs if its tail request latency is not longer than a user-specified deadline, and meter the resource cost by the number of worker nodes. Key to achieving this goal is to *maximize the number of SLO-compliant functions* at each worker, such that `Torpor` can efficiently exploit per-worker GPU resources to host as many functions as possible, which in turn reduces the total number of workers required.

Challenges. Previous systems have proposed various schemes to meet latency SLOs [32, 58, 65, 67]; however, their policies do not apply to `Torpor` for two reasons. First, previous systems like INFless [58] and Shepherd [67] assume sufficient GPU memory and employ early binding, so they schedule model serving instances to GPUs and then batch and route requests to them. In contrast, `Torpor` focuses on late-binding the often lower-frequency or varying-demand functions to a pool of memory-constrained GPUs, requiring a joint design of model management (i.e., model swapping and eviction) and request scheduling. Second, previous systems assume a stable model inference latency [32, 47, 58, 65, 67] which, however, does not hold in our setting — model swapping can cause unpredictable performance due to PCIe bandwidth contention [21, 38]. For instance, as show in Table 3, concurrently swapping two models through PCIe increases individual model inference latency compared with running them alone, especially for large models (e.g., Bert-qa).

We propose three policies to address the aforementioned challenges. First, considering that packing many functions together can cause short-term overloading and request queueing, `Torpor` introduces a request prioritization policy to maximize the number of SLO-compliant functions (§5.2). Second, `Torpor` designs a request scheduling and model swapping policy to reduce bandwidth contention across concurrent models, thereby improving overall inference performance (§5.3). Lastly, by leveraging the characteristics of model swapping, `Torpor` proposes an effective model eviction policy to reduce bandwidth footprint in model swapping; combined with the request scheduling policy, `Torpor` minimizes the interference among concurrent model executions (§5.4).

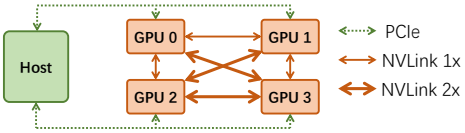


Figure 5: Topology of a 4-GPU worker node in Alibaba Cloud.

5.2 Request Queueing

To maximize the number of SLO-compliant functions, `Torpor` needs to monitor the SLO compliance of individual functions to determine their request executions. Intuitively, `Torpor` prioritizes functions with a higher probability to comply with SLOs. However, realizing this approach requires answering two questions: 1) how to quantify the likelihood of SLO compliance for a function, and 2) how to determine function execution order for improved SLO compliance.

`Torpor` proposes a metric, *required request count* (RRC), to measure the "degree of needed effort" to meet SLOs. RRC represents the expected request number that a function needs to successfully serve in order to satisfy SLOs. Let n be the current number of requests to a function, and m be the number of requests served within deadlines out of n. The RRC of this function is defined as $\frac{pn-m}{1-p}$, where p is the tail percentile specified in SLOs such as 98%. This is derived from the equation: $\frac{m+RRC}{n+RRC} = p$. Smaller RRC values indicate a higher likelihood of SLO attainment. Hence `Torpor` divides functions into high- and low-priority groups based on their RRCs, and prioritizes their requests accordingly. We develop an effective strategy to determine the boundary (i.e., RRC threshold) between the two groups that can dynamically adjust function prioritization based on the current load at a worker, thereby improving the overall SLO compliance (see details in our technical report [62]).

5.3 Scheduling and Model Swapping

While model execution latency is often stable, model swapping can incur unpredictable overhead due to PCIe bandwidth contention [21, 38]. Fig. 5 shows the topology of a worker node in Alibaba Cloud, where each pair of GPUs shares a PCIe switch and GPUs are inter-connected via NVLinks with various bandwidths[6]. The performance slowdown caused by

[6]Despite the presence of other GPU interconnects (e.g., NVSwitch in DGX A100), inter-GPU PCIe bandwidth sharing continues to necessitate interference mitigation.

Algorithm 1 Interference-Aware Request Scheduling

```
1:  function SCHEDULE(req r)
2:      A ← set of available GPUs                    ▷ A ≠ ∅, otherwise queueing r
3:      M ← set of GPUs hosting the target model
4:      if M ≠ ∅ then
5:          G ← M ∩ A
6:          if G ≠ ∅ then
7:              g ← any GPU in G
8:              Execute r on g                       ▷ No swapping
9:          else
10:             (g,m) ← GPU pair with fastest NVLink, g ∈ A, m ∈ M
11:             Execute r on g; Swap model from m    ▷ GPU-to-GPU
12:     else
13:         g ← a GPU whose neighbor is not loading models, g ∈ A
14:         if g not found then
15:             g ← a GPU whose neighbor is loading a light model, g ∈ A
16:         if g not found then
17:             g ← any GPU in A
18:         Execute r on g; Swap model from host     ▷ Host-to-GPU
```

bandwidth contention can vary among models as shown in Table 3, where larger models require more intensive data transmission and exhibit more pronounced performance degradation. Hence we propose interfere-aware scheduling to minimize PCIe contention, thereby reducing request latencies.

Interference-aware scheduling. `Torpor` exploits the direct NVLink connections between GPUs to reduce PCIe contention whenever possible. It prioritizes GPU-to-GPU over host-to-GPU model swapping to enable faster model transmission and avoid interference with concurrent PCIe traffic. When concurrent host-to-GPU swapping is unavoidable, `Torpor` avoids loading bandwidth-intensive models (e.g., Bert-qa in Table 3) simultaneously to minimize the impact of PCIe contention. Therefore, models are categorized as heavy or light based on their bandwidth requirements (see Table 4).

Algorithm 1 shows `Torpor`'s scheduling and swapping mechanisms. `Torpor` first checks whether the target model is loaded on an available GPU, and if so, directly executes it without swapping (line 8). If the model is hosted by busy GPUs, `Torpor` then schedules the request to perform GPU-to-GPU swapping, as the source and target GPUs should have a fast NVLink connection (line 11). Otherwise, `Torpor` resorts to the host-to-GPU swapping and prioritizes target GPUs whose neighbors are idle or running light models to reduce PCIe contention (line 18). Altogether, `Torpor` minimizes the interference and overhead of model swapping for each request, thus providing low inference latency.

5.4 Model Eviction Policy

Model eviction plays a critical role in reducing bandwidth contention and enhancing the overall inference performance, in conjunction with `Torpor`'s request scheduling policy. Unlike traditional cache eviction strategies which primarily aim to minimize the miss rates, `Torpor`'s model eviction policy considers the performance implications of model swapping for different models to facilitate future model loading.

We notice that swapping light models leads to negligible overhead for end-to-end performance compared with heavy ones (Table 3 and Table 4). Therefore, we tend to evict models that have little or no impact on performance when swapping. We employ two mechanisms following this insight. First, `Torpor` manages memory of all GPUs as a pool to globally optimize model placement, which ensures that each model can have up to one replica among GPUs when GPU memory is full. This allows for more efficient model caching and reduces host-to-GPU data transmission. Second, `Torpor` prioritizes light models in eviction, as swapping them leads to negligible or no PCIe bandwidth contention. When only heavy models remain, `Torpor` adopts Least-Recently-Used (LRU) policy to determine their eviction order.

6 Implementation

We have implemented `Torpor` for Alibaba Cloud, one of the world's leading commercial serverless platforms. `Torpor`'s GPU server and GPU client were implemented in 4k and 1.5k lines of C++ code, respectively. Intra-node router and cluster manager were implemented atop the relevant components in Alibaba Cloud. `Torpor`'s late-binding mechanism imposes no intrusive changes to standard cluster management logic, enabling the reuse of Alibaba Cloud's existing manager with minimal changes. In fact, `Torpor` has been successfully deployed in a real-world production environment of Alibaba Cloud (§8). We provide a container image as a function template based on PyTorch, where the original CUDA libraries (e.g., `libcudart.so`) are replaced by our GPU clients to enable GPU remoting. This requires no modification to the PyTorch framework.

7 Evaluation

In this section, we evaluate `Torpor` with the runtime sharing mode using production traces from Alibaba Cloud. We intergrate `Torpor` with runtime isolation mode into Alibaba Cloud's real-world production environment and report the results in §8.

Settings. We deploy `Torpor` at Alibaba Cloud following the realistic production specification of its serverless platform. `Torpor` runs in a cluster with up to 6 workers. Each worker node has 48 vCPU cores, 384 GB memory, and 4 NVIDIA V100 GPUs each with 32 GB memory. We use 8 popular ML models for evaluation, as shown in Table 4, and distribute them across inference functions in a round-robin manner. All functions are warmed up before running the test workloads. We compare `Torpor` against Native execution—the default approach in Alibaba Cloud—and INFless [58], a state-of-the-art serverless inference system.

Metrics. We focus on the ratio of functions meeting SLOs and

Table 4: Various models and their latencies (ms) with GPU remoting and model swapping. Underlined are heavy models where swapping via PCIe slows down the inference (see §5.3).

Model	Native	GPU remoting	Swap-PCIe	Swap-NVLink
DenseNet-169	30	25	27	26
DenseNet-201	36	28	30	30
Inception-v3	19	14	17	16
EfficientNet	17	12	13	13
ResNet-50	11	9	13	11
ResNet-101	20	14	22	16
ResNet-152	25	17	25	20
Bert-qa	42	43	144	45

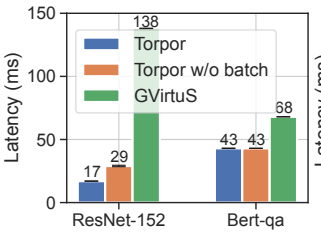


Figure 6: Inference latency with Torpor and other GPU remoting techniques.

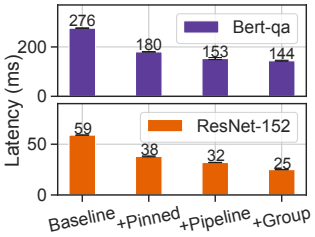


Figure 7: Performance breakdown of Torpor's model swapping via PCIe.

the GPU load in the evaluation. A function complies SLOs if its tail request latency is within a deadline. By default, we use 98^{th} tail latency, and set deadlines for CV models and Bert-qa to 80 ms and 200 ms, respectively. The GPU load is measured by the proportion of time during which the GPU is processing inference requests.

7.1 Overhead of Torpor's Late Binding

Table 4 compares the performance of 8 popular models under Native execution and Torpor with its GPU remoting (§4.2) and model swapping (§4.3). For GPU remoting, Torpor adopts efficient, asynchronous API redirection, leading to comparable performance to Native, or even better for CV models. This is because serving these models requires configuring many cuDNN descriptors where the relevant CUDA APIs are executed on the CPU side and do not require GPU resources; thus, redirecting these APIs effectively distributes CPU-side workloads across functions and the GPU server, enabling functions to access more CPUs and perform parallel computation. Note that, CPU resources are not the bottleneck in this scenario. According to our measurements, the CPU utilization of Torpor (native execution) for ResNet-152 and Bert-qa remains at 27.4% (8.8%) and 17.2% (9.2%), respectively. For model swapping, Torpor supports efficient pipeline execution through PCIe, and leveraing NVLink further improves performance.

Performance of Torpor's GPU remoting. To show the advantage of Torpor's GPU remoting, we compare it with GVirtuS [9, 31], a leading solution among publicly available GPU

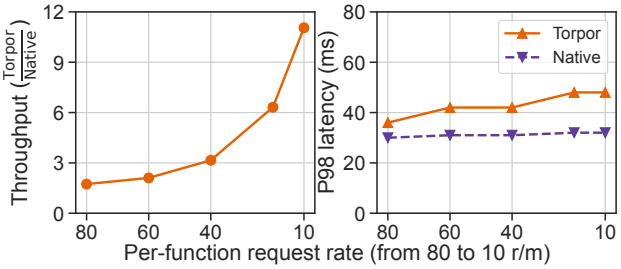


Figure 8: Performance of executing multiple ResNet-152 functions on a single GPU with Torpor's late binding (Torpor) and Native execution. We show the throughput of Torpor normalized to Native (left) and the tail latencies under varying per-function request rates (right).

remoting techniques. GVirtuS adopts a synchronous approach to API redirection. We also evaluate a variant of Torpor that disables call batching in asynchronous API redirection, i.e, "Torpor w/o batch". Fig. 6 shows the inference performance under various approaches. Torpor significantly outperforms GVirtuS and reduces latencies by 88% and 37% for ResNet-152 and Bert-qa, respectively. ResNet-152 triggers a large number of API calls during each inference, leading to high synchronization overhead for GVirtuS. Asynchronous API redirection (Torpor w/o batch) dramatically reduces the latency by 79%; with API call batching, Torpor further reduces the latency by 41%. Compared with ResNet-152, Bert-qa requires less communication in GPU remoting; therefore, the improvement from asynchronous API redirection is less but still quite significant.

Performance breakdown of model swapping. To illustrate how each of Torpor's model swapping designs contributes to performance improvement, we break down the inference performance of ResNet-152 and Bert-qa, as shown in Fig. 7. Specifically, "Baseline" directly performs model swapping and then executes inference; "Pinned" uses a pinned memory pool for improved swapping performance; "Pipeline" overlaps model swapping and execution at the granularity of individual model parameters; "Group" groups parameters for efficient pipeline execution. We note that enabling pinned memory reduces overall latencies by around 35%; parameter-level pipeline execution further reduces the latency by 15%. By grouping model parameters, Torpor achieves up to 22% performance improvement over "Pipeline", especially for ResNet-152 that consists of many small-sized parameters.

7.2 Benefits of Torpor's Late Binding

GPU efficiency for low-frequency functions. With late binding, Torpor substantially reduces per-function memory footprint, thereby enabling the consolidation of many low-frequency functions for improved GPU efficiency. We stress-test its performance by executing multiple ResNet-152 functions on a single GPU, varying request rates between 80

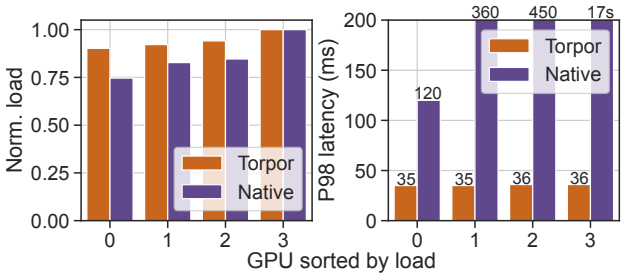


Figure 9: The normalized per-GPU load and the tail request latency with Torpor's late binding (Torpor) and Native.

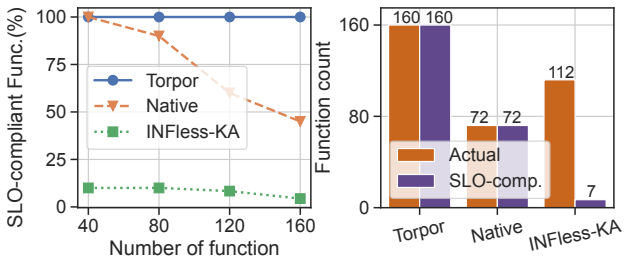


Figure 10: Performance comparison in terms of SLO compliance between Torpor, Native, and INFless-KA.

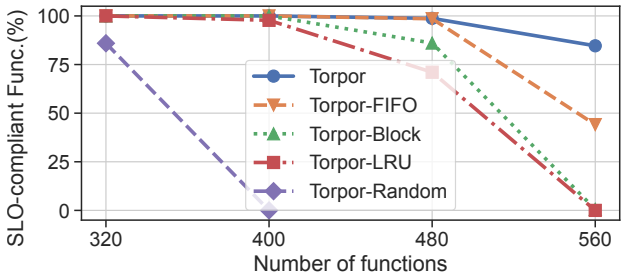


Figure 11: Ratio of SLO-compliant functions with the full Torpor and various different policies.

and 10 requests per minute (r/m)—a typical range of low-frequency functions in production traces (Fig. 2). In this setup, we execute sufficient concurrent functions on a GPU node to saturate GPU memory and ensure a high overall load. Fig. 8 compares normalized throughput and latency under Torpor and Native executions. Native's throughput declines as the per-function request rate decreases, due to its limited capacity to host many functions. In contrast, Torpor leverages host memory to accommodate many more functions, maintaining high throughput with efficient, request-level GPU sharing. For example, Torpor achieves over 10× higher throughput than Native at 10 r/m. Furthermore, even when model swapping is required, Torpor still keeps the tail latency below 50 ms.

Cross-GPU load balancing for high-frequency functions. We run 40 high-frequency ResNet-152 functions on a 4-GPU worker, where the average request rate is around 200 r/m. Fig. 9 shows the normalized per-GPU load and the tail request latency with Torpor and Native. Unlike Native, where GPUs hosting high-frequency functions can easily become overloaded due to bursts of requests, Torpor enables on-demand model migration for efficient load balancing. Therefore, Torpor achieves much less load variance across GPUs compared with Native, as shown in Fig. 9 (left). Moreover, Fig. 9 (right) shows the tail latency of requests executed on each GPU, where Native leads to extremely long tail latency (e.g., multi-seconds) due to high queueing delays. In contrast, Torpor consistantly delivers fast model inference, achieving a tail latency of around 35 ms on all GPUs.

7.3 Torpor at A Node

We next evaluate the performance of Torpor at a node. We use real-world workloads sampled from production traces (Fig. 2), where function request rates range from 5 to 30 r/m.

Performance comparison. We compare Torpor with two baselines, Native execution and INFless [58] — a state-of-the-art serverless inference system. INFless introduces a function keep-alive policy to set the lifespan of individual functions based on historical traces, denoted as INFless-KA. For a fair comparison, we implement the keep-alive policy of INFless (INFless-KA) in the Native system. Fig. 10 (left) shows the ratio of functions meeting SLOs in Torpor, Native, and INFless-KA. Fig. 10 (right) shows the numbers of functions being actually executed and being SLO-compliant, when hosting 160 functions. With fast model swapping, Torpor executes all 160 functions and also meets SLOs for all functions. In contrast, due to limited GPU memory, Native can only execute 72 out of 160 functions. INFless-KA can reclaim GPU memory via cleaning up idle functions, thereby enabling the execution of more functions (i.e., 112 functions) than Native; however, INFless-KA inevitably incurs function cold starts and results in only 7 functions being SLO-compliant.

Benefits of Torpor's policies. To understand the benefits of Torpor's policy designs, we compare the full Torpor with four baselines. 1) *Torpor-FIFO* uses a FIFO policy in request queueing rather than our SLO-aware policy (§5.2). 2) *Torpor-Random* disables our interference-aware scheduling and model swapping (§5.3), and randomly schedules a request to an idle GPU if the target model is not loaded, and then triggers model swapping. 3) *Torpor-LRU* adopts a LRU policy in model eviction rather than prioritizing models according to swapping overheads (§5.4). 4) *Torpor-Block* disables our block management policy (§4.4), and caches the released memory blocks in a single pool; when a new block is required, it directly returns a cached one in the pool if the requested size can be satisfied, otherwise it frees multiple blocks until the required memory space is available.

Fig. 11 shows the ratio of SLO-compliant functions. In particular, Torpor-FIFO is oblivious to SLOs and unable to properly prioritize functions, leading to serious SLO violations when the number of functions is large. Torpor-Block cannot reuse various-sized blocks and forces frequent memory allo-

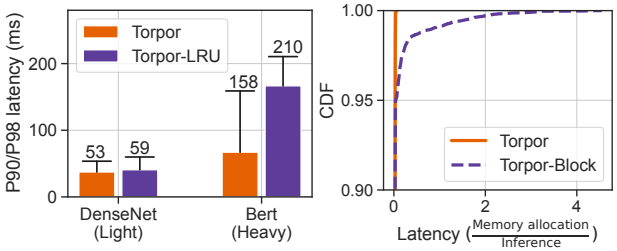


Figure 12: 90^{th} (bars) and 98^{th} (whiskers) tail latencies in `Torpor` and `Torpor`-LRU.

Figure 13: Latency CDF of memory allocation in `Torpor` and `Torpor`-Block.

cation via CUDA APIs, which incurs long delay in block allocation and harms overall performance. `Torpor`-LRU evicts heavy models often, leading to PCIe bandwidth contention during future model swapping. `Torpor`-Random leads to the worst performance due to its inefficient scheduling and model swapping policy, which does not exploit NVLink across GPUs and is oblivious to model heaviness. Compared with these baselines, `Torpor` successfully supports over 80% functions even with 560 functions, maximizing the number of SLO-compliant functions.

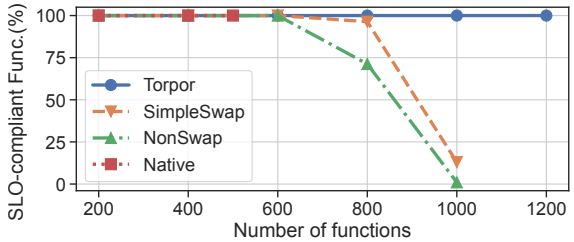
Efficiency of model heaviness. We evaluate the efficiency of model heaviness with `Torpor` and `Torpor`-LRU, using DenseNet-201 and Bert-qa as light and heavy models, respectively. Fig. 12 shows tail latencies of DenseNet-201 and Bert-qa in a mixed workload, where Bert-qa instances account for 60% of overall memory footprint. `Torpor`-LRU is agnostic to model heaviness and can evict heavy models frequently, leading to high tail latencies for Bert-qa that fail to meet its SLOs (200 ms). In contrast, `Torpor` effectively reduces tail latencies of Bert-qa without compromissing performance for DenseNet-201, achieving SLOs for both models.

Efficiency of memory allocation. Fig. 13 shows the distribution of the latencies of per-request memory allocation normalized to inference time, under `Torpor` and `Torpor`-Block. Due to efficient memory allocation and sharing (§4.4), `Torpor` incurs only negligible overhead. In contrast, `Torpor`-Block leads to high allocation overhead (e.g., over 4× than the actual inference time), harming the end-to-end performance.

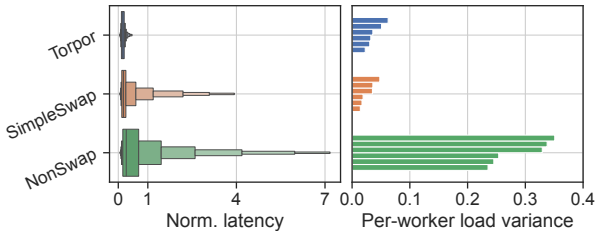
7.4 `Torpor` in A Cluster

We next evaluate `Torpor` in a cluster deployment with 6 GPU worker nodes. As running `Torpor` in a Alibaba Cloud cluster incurs additional system overhead, we set the SLOs for CV models and Bert-qa to 150 ms and 250 ms, respectively.

Baselines. We exclude INFless-KA from our cluster deployment due to its poor performance (see Fig. 10). We use three baselines: 1) *Native* uses native GPU containers bound to specific GPUs, which is the current practice in Alibaba Cloud. 2) *NonSwap* allows GPU remoting similar to `Torpor`, but disables model swapping, reducing memory footprint compared with Native. 3) *SimpleSwap* enables model swapping



(a) Ratio of SLO-compliant functions under `Torpor` and baselines.



(b) Distribution of per-request latencies normalized to deadlines (left) and the per-worker variance of GPU load normalized to the maximum (right), when running 1k functions. Boxes (left) depict the $1/128$, $1/64$, …, $1/2$, …, $63/64$, $127/128$ quantiles.

Figure 14: Cluster evaluation of `Torpor`.

compared with NonSwap. This approach only supports simple strategies as discussed in §7.3, including FIFO request queueing, random scheduling, and LRU model eviction.

Cluster evaluation. Fig. 14 compares `Torpor` with these three baselines. We first show the ratio of SLO-compliant functions under various number of functions. As shown in Fig. 14a, only `Torpor` can consistently meet per-function latency SLOs even with a large number of functions (e.g., over 1000). Native quickly saturates all GPU memory and only supports up to 500 functions, thus low GPU utilization. Compared with Native, NonSwap relaxes the constraint of GPU memory and enables more functions; however, it still fixes the binding between functions and GPUs, and causes GPUs overloaded by requests and leads to long tail latency. Moreover, while SimpleSwap outperforms NonSwap with model swapping, it still suffers from severe SLO violations with a large number of functions (e.g., 1k).

Fig. 14b compares the behaviors of `Torpor`, SimpleSwap, and NonSwap under 1k functions. We show the distribution of per-request latencies normalized to corresponding deadlines (left). In `Torpor`, almost every request can be served within its deadline, leading to a normalized latency less than 1. However, SimpleSwap and NonSwap suffer from long tail latency — over 4× and 7× of the respective deadlines. We also compare the per-worker GPU load of the three systems. For each worker node, we normalize the loads of its four GPUs to the maximum, and calculate the variance. Lower variance indicates better load balancing. Fig. 14b (right) plots the per-worker load variances of three solutions, each with 6 workers in total. Compared with NonSwap, `Torpor` and SimpleSwap

Table 5: Overview of Torpor's pilot production.

Metric	Value	Metric	Value
# of users	> 150	Users' cost savings	70% on avg.
# of GPUs	> 350	GPU savings	65%
# of daily requests	up to 465k		

can effectively balance GPU load across workers with model swapping, thus achieving much less load variance.

8 Torpor in Pilot Production

Torpor has been deployed in a pilot production cluster in Alibaba Cloud for beta testing. In this section, we present the testing results and our observations.

Overview of the pilot production. In the deployment, we employ a cost-efficient billing scheme in which users are charged only 10% of the GPU cost when their functions are inactive and retained in the host memory. This billing scheme aligns with Torpor's late binding mechanism and has attracted a variety of real-world inference workloads to Alibaba Cloud, including image processing, text generation, and image generation. Table 5 provides an overview of our pilot production system and the achieved savings. Currently, Torpor serves over 150 users in a cluster with more than 350 GPUs, handling up to 465k requests everyday. The system achieves an average cost savings of 70% for users compared to the previous approach of Alibaba Cloud that kept functions long-running on GPUs and billed users for the entire GPU time. Moreover, Torpor enables Alibaba Cloud to consolidate various functions for improved GPU utilization, resulting in a 65% reduction in the total number of required GPUs and cost savings for Alibaba Cloud.

Startup latency. Table 1 shows the performance of Torpor in the pilot production across various models, ranging from CNNs such as ResNet to LLMs like Llama. In the production environment, Torpor prioritizes user isolation and manages a dedicated GPU runtime for each function. Hence we also provide a detailed breakdown of the time required for model loading and runtime resumption in Table 1. Compared to cold starts in Alibaba Cloud, Torpor reduces model startup latencies by over an order of magnitude, e.g., cutting Llama2-13B's startup time to 4.4 seconds—a delay that is acceptable considering that generating all output tokens for a query can take tens of seconds [30, 68].

Case study. We next present a case study of a realistic GPU function in our pilot production, which involves text generation from input images. This function is invoked several thousands times per day and follows a request arrival pattern similar to Fig. 1. Without Torpor, this function would need to be kept long-running on a GPU for low inference latency, resulting in high GPU costs. Fig. 15 depict latency distribution and user cost of this function. For confidentiality reasons, we normalize the latency of model (and runtime) loading to the mean of end-to-end times, and the cost to that of long-running GPU functions (i.e., w/o Torpor). The loading duration is consistent, as shown in Fig. 15 (left), which accounts for only ∼30% of the overall latency. Fig. 15 (right) illustrates the user cost, where Torpor reduces the total cost by 84% compared to the previous approach in Alibaba Cloud. Based on this real-world use case, Torpor achieves significant cost savings without compromising end-to-end performance, proving to be an effective solution for serverless inference.

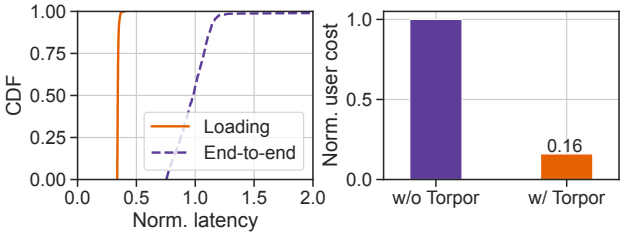


Figure 15: Latency distribution and user cost of a realistic GPU function in Alibaba Cloud.

Discussion. We have identified three challenges that require further investigation based on our experiences with the production cluster. *(1) Extremely infrequent functions:* We have observed that some functions are invoked extremely infrequently (e.g., a few requests per hour). However, these functions still result in substantial user costs due to their host memory usage. To address this, we plan to explore the utilization of cheaper storage for low-demand functions and investigate a multi-tier storage architecture that dynamically adapts to request patterns for improved resource and cost efficiency. *(2) Very large models:* While Torpor's late–binding design is versatile and has been applied to large models (Table 1), it currently does not support model-parallel execution across multiple GPUs. To accommodate very large models spanning multiple GPUs or nodes, it is crucial to enable efficient model partitioning and parallelization that integrates with Torpor's model-swapping mechanism. Therefore, we intend to extend Torpor to leverage various interconnects, including PCIe and NVLink (Fig. 5), for enhanced model parallel and pipeline execution. *(3) Highly bursty workloads:* While Torpor is primarily designed for efficient resource sharing among many low-frequency functions (Fig. 2), some functions may experience highly bursty workloads with short periods of extensive request arrivals. To handle such spikes, Torpor can proactively maintain host-memory-cached model replicas across multiple nodes, mitigating cold starts at the cost of increased memory usage. We plan to explore high-speed inter-node networks (e.g., RDMA) for efficient model loading between nodes, which can reduce memory footprint while maintaining low-latency model startup.

9 Related Work

Serverless Inference. In addition to the serverless platforms discussed in §2.2, there are several recent works on serverless

inference. StreamBox [56] and Dilu [43] support spatial GPU sharing for concurrent model execution; FaaSTube [55] improves data sharing within inference workflows. These works are orthogonal to Torpor and can be integrated for enhanced performance. ServerlessLLM [30] and Medusa [64] propose cold-start optimizations specialized for large language models, which is complementary to Torpor and can be used to accelerate those specific models.

Host-to-GPU data swapping. Many other systems have leveraged host-to-GPU data swapping in general deep learning and GPU workloads [23, 32, 34, 39, 46, 50, 57, 63]. For example, vDNN [46], Salus [63] and SwapAdvisor [34] leverage host memory for deep learning jobs with large GPU memory footprints; Batch-aware [39] and HUVM [23] optimize GPU memory access for general-purpose workloads; POS [35] supports efficient GPU checkpointing and restoring. Compared with Torpor, these systems are not specifically designed for model inference and do not account for its SLO attainment. Inference systems such as PipeSwitch [22] and DeepPlan [36] improve host-to-GPU model loading for fast switching. Unlike Torpor, these systems require detailed model-specific knowledge and do not target meeting model-level SLOs in a shared, multi-tenant serverless environment.

GPU remoting. GPU remoting techniques have been employed in different layers for GPU virtualization [28, 31, 45, 59]. Existing solutions like GVirtuS [31] and rCUDA [28] primarily focus on general-purpose workloads. Torpor applies GPU remoting in serverless inference, leveraging its characteristics for asynchronous, low-latency API redirection.

Spatio-temporal GPU sharing. Existing techniques have investigated the spatial and temporal GPU sharing to improve overall utilization [2, 13–15, 24, 33, 51, 56]. These techniques are orthogonal to Torpor and can be directly applied, which allow partitioning a physical GPU into multiple virtual instances to late-bind more functions.

10 Conclusion

This paper introduces Torpor, a serverless platform for SLO-aware and GPU-efficient model inference. Torpor employs a late binding approach, managing inference functions in host memory and dynamically swapping them to a pool of GPUs upon request arrivals. This approach enables pay-per-GPU-use billing and maximizes resource utilization. Additionally, Torpor proposes request scheduling and model management policies to meet latency SLOs for inference functions while minimizing resource costs. Torpor has been beta released in a large commercial serverless platform, successfully serving up to 465k requests per day and achieving 70% and 65% GPU cost savings for users and the platform, respectively.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Somali Chaterji, for their insightful comments that helped improve this work. We also thank Bohui Wu and Zhexiang Zhang for their help in experiments. This work was supported in part by the Alibaba Innovative Research (AIR) Grant, RGC CRF Grant (Ref. #C6015-23G), RGC GRF Grants (Ref. #16217124 and #16210822), NSFC/RGC CRS Grant (Ref. #CRS_HKUST601/24), and CUHK-Shenzhen Research Grant (UDF01003466).

References

[1] Alibaba Cloud Function Compute. https://www.alibabacloud.com/product/function-compute.

[2] Aliyun cGPU. https://www.alibabacloud.com/help/en/container-service-for-kubernetes/latest/cgpu-overview.

[3] Aliyun Function Compute Billing Scheme. https://www.alibabacloud.com/help/en/function-compute/latest/billing-billing.

[4] Aliyun Function Compute Instance Types and Modes. https://www.alibabacloud.com/help/en/function-compute/latest/instance-types-and-instance-modes.

[5] Amazon SageMaker. https://aws.amazon.com/sagemaker/.

[6] AWS Lambda. https://aws.amazon.com/lambda/.

[7] AWS Lambda Provisioned Concurrency. https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html.

[8] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[9] GVirtuS. https://github.com/gvirtus/GVirtuS.

[10] Llama2. https://www.llama.com/llama2.

[11] Llama3. https://www.llama.com/models/llama-3.

[12] Memory Management on Modern GPU Architectures. https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9727-memory-management-on-modern-gpu-architectures.pdf.

[13] Nvidia Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[14] Nvidia Multi-Process Service. https://docs.nvidia.com/deploy/mps/.

[15] Nvidia Virtual GPU. https://www.nvidia.com/en-us/data-center/virtual-solutions/.

[16] Qwen. https://github.com/QwenLM/Qwen.

[17] ResNet in PyTorch. https://pytorch.org/vision/stable/models/resnet.html.

[18] Stable Diffusion. https://huggingface.co/stable-diffusion-v1-5/stable-diffusion-v1-5.

[19] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *Proc. USENIX NSDI*, 2020.

[20] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. BATCH: Machine learning inference serving on serverless platforms with adaptive batching. In *Proc. ACM/IEEE Supercomputing*, 2020.

[21] Marcelo Amaral, Jordà Polo, David Carrera, Seetharami Seelam, and Malgorzata Steinder. Topology-aware gpu scheduling for learning workloads in cloud environments. In *Proc. ACM SC*, 2017.

[22] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. PipeSwitch: Fast pipelined context switching for deep learning applications. In *Proc. USENIX OSDI*, 2020.

[23] Sangjin Choi, Taeksoo Kim, Jinwoo Jeong, Myeongjae Jeon, Youngjin Kwon, Rachata Ausavarungnirun, and Jeongseob Ahn. Memory harvesting in multi-GPU systems with hierarchical unified virtual memory. In *Proc. USENIX ATC*, 2022.

[24] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on multi-GPU servers with spatio-temporal sharing. In *Proc. USENIX ATC*, 2022.

[25] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *Proc. USENIX NSDI*, 2017.

[26] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[27] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless computing on heterogeneous computers. In *Proc. ACM ASPLOS*, 2022.

[28] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *Proc. IEEE HPCS*, 2010.

[29] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J. Rossbach. Dgsf: Disaggregated gpus for serverless functions. In *Proc. IEEE IPDPS*, 2022.

[30] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Locality-enhanced serverless inference for large language models. In *Proc. USENIX OSDI*, 2024.

[31] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A gpgpu transparent virtualization component for high performance computing clouds. In *Proc. Euro-Par*, 2010.

[32] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *Proc. USENIX OSDI*, 2020.

[33] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *Proc. USENIX OSDI*, 2022.

[34] Chien-Chin Huang, Gu Jin, and Jinyang Li. SwapAdvisor: Pushing deep learning beyond the GPU memory limit via smart swapping. In *Proc. ACM ASPLOS*, 2020.

[35] Zhuobin Huang, Xingda Wei, Yingyi Hao, Rong Chen, Mingcong Han, Jinyu Gu, and Haibo Chen. PARALLELGPUOS: A concurrent OS-level GPU checkpoint and restore system using validated speculation. *arXiv preprint arXiv:2405.12079*, 2024.

[36] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. Fast and efficient model serving using multi-gpus with direct-host-access. In *Proc. ACM EuroSys*, 2023.

[37] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proc. ACM ASPLOS*, 2021.

[38] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters. In *Proc. USENIX OSDI*, 2020.

[39] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. Batch-aware unified memory management in GPUs for irregular workloads. In *Proc. ACM ASPLOS*, 2020.

[40] Kenneth C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, 1965.

[41] Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman. Parity models: erasure-coded resilience for prediction serving systems. In *Proc. ACM SOSP*, 2019.

[42] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the black box of machine learning prediction serving systems. In *Proc. USENIX OSDI*, 2018.

[43] Cunchi Lv, Xiao Shi, Zhengyu Lei, Jinyue Huang, Wenting Tan, Xiaohui Zheng, and Xiaofang Zhao. Dilu: Enabling GPU resourcing-on-demand for serverless DL serving via introspective elasticity. In *Proc. ACM ASPLOS*, 2025.

[44] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based GPU memory management for deep learning. In *Proc. ACM ASPLOS*, 2020.

[45] C. Reaño, A. J. Peña, F. Silla, J. Duato, R. Mayo, and E. S. Quintana-Ortí. Cu2rcu: Towards the complete rcuda remote gpu virtualization and sharing solution. In *Proc. IEEE HiPC*, 2012.

[46] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proc. ACM/IEEE MICRO*, 2016.

[47] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *Proc. USENIX ATC*, 2021.

[48] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proc. USENIX ATC*, 2020.

[49] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *Proc. ACM SOSP*, 2019.

[50] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. High-throughput generative inference of large language models with a single gpu. *arXiv preprint arXiv:2303.06865*, 2023.

[51] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained gpu sharing for ml applications. In *Proc. ACM EuroSys*, 2024.

[52] Huangshi Tian, Suyi Li, Ao Wang, Wei Wang, Tianlong Wu, and Haoran Yang. Owl: Performance-aware scheduling for resource-efficient function-as-a-service cloud. In *Proc. ACM SoCC*, 2022.

[53] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proc. ACM ASPLOS*, 2021.

[54] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *Proc. USENIX ATC*, 2021.

[55] Hao Wu, Junxiao Deng, Minchen Yu, Yue Yu, Yaochen Liu, Hao Fan, Song Wu, and Wei Wang. Faastube: Optimizing gpu-oriented data transfer for serverless computing. *arXiv preprint arXiv:2411.01830*, 2024.

[56] Hao Wu, Yue Yu, Junxiao Deng, Shadi Ibrahim, Song Wu, Hao Fan, Ziyue Cheng, and Hai Jin. StreamBox: A lightweight GPU SandBox for serverless inference workflow. In *Proc. USENIX ATC*, 2024.

[57] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *Proc. USENIX OSDI*, 2020.

[58] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. INFless: a native serverless system for low-latency, high-throughput inference. In *Proc. ACM ASPLOS*, 2022.

[59] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J. Rossbach. AvA: Accelerated virtualization of accelerators. In *Proc. ACM ASPLOS*, 2020.

[60] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Following the data, not the function: Rethinking function orchestration in serverless computing. In *Proc. USENIX NSDI*, 2023.

[61] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *Proc. IEEE ICDCS*, 2021.

[62] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, Haoran Yang, and Yu Ding. Torpor: Gpu-enabled serverless computing for low-latency, resource-efficient inference. *arXiv preprint arXiv:2306.03622*, 2025.

[63] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained GPU sharing primitives for deep learning applications. In *Proc. MLSys*, 2020.

[64] Shaoxun Zeng, Minhui Xie, Shiwei Gao, Youmin Chen, and Youyou Lu. Medusa: Accelerating serverless LLM inference with materialization. In *Proc. ACM ASPLOS*, 2025.

[65] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. In *Proc. USENIX ATC*, 2019.

[66] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: NIMBLE task scheduling for serverless analytics. In *Proc. USENIX NSDI*, 2021.

[67] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *Proc. USENIX NSDI*, 2023.

[68] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

Towards GPU Memory Efficiency for Distributed Training at Scale

Runxiang Cheng[†], Chris Cai[‡], Selman Yilmaz[‡], Rahul Mitra[‡],
Malay Bag[‡], Mrinmoy Ghosh[‡], Tianyin Xu[†]
[†]University of Illinois Urbana-Champaign [‡]Meta Platforms, Inc.

ABSTRACT

The scale of deep learning models has grown tremendously in recent years. State-of-the-art models have reached billions of parameters and terabyte-scale model sizes. Training of these models demands memory bandwidth and capacity that can only be accommodated distributively over hundreds to thousands of GPUs. However, large-scale distributed training suffers from GPU memory inefficiency, such as memory under-utilization and out-of-memory events (OOMs). There is a lack of understanding of actual GPU memory behavior of distributed training on terabyte-size models, which hinders the development of effective solutions to such inefficiency. In this paper, we present a systematic analysis of GPU memory behavior of large-scale distributed training jobs in production at Meta. Our analysis is based on offline training jobs of multi-terabyte Deep Learning Recommendation Models from one of Meta's largest production clusters. We measure GPU memory inefficiency; characterize GPU memory utilization, and provide fine-grained GPU memory usage analysis. We further show how to build on the understanding to develop a practical GPU provisioning system in production.

CCS CONCEPTS

• **Software and its engineering → Cloud computing**; • **Computing methodologies → Machine learning**.

KEYWORDS

Distributed Training, GPU Efficiency, Resource Provisioning.

ACM Reference Format:
Runxiang Cheng, Chris Cai, Selman Yilmaz, Rahul Mitra, Malay Bag, Mrinmoy Ghosh, and Tianyin Xu. 2023. Towards GPU Memory Efficiency for Distributed Training at Scale. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620678.3624661

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

ACM ISBN 979-8-4007-0387-4/23/11.
https://doi.org/10.1145/3620678.3624661

1 INTRODUCTION

The scale of deep learning models (referred to as DNNs) has grown tremendously in recent years. Practitioners and researchers continue to build deeper and larger models to gain better learning performance. Nowadays, deep learning models that power the growth and main revenue stream in large technology companies have billions of parameters, and terabyte-scale model sizes [1, 5, 8, 26, 75].

Training of large-scale models demands significant memory capacity and memory bandwidth, which cannot be accommodated on a single GPU device. Therefore, various distributed training techniques were developed to enable large-scale DNN training on multiple devices [27, 46, 76, 80, 81]. Modern distributed training paradigm splits the memory-intensive model component (via *model parallelism*) and replicates the compute-intensive model component (via *data parallelism*) across GPU devices. These approaches have shown their effectiveness in training large models and increasing the training throughput in practice.

Today, GPU clusters dedicated to large-scale distributed training are common in practice [1, 25, 30, 32]. However, cluster-level GPU memory management for distributed training remains inefficient. Without effective tools, engineers often over- or under-provision GPUs for a training job. Our analysis shows that, in one of the Meta's largest production training clusters, half of the jobs utilize at most 50% of provisioned GPU memory, and 9% of the failed jobs were induced by GPU out-of-memory events (OOMs). GPUs are becoming more scarce due to wide-spread learning demands [11, 22, 47]. For example, OpenAI is heavily GPU-limited at present and GPU shortage is delaying a lot of their short-term plans [22]. Improving GPU memory efficiency in large-scale training is thus important, as training scalability and efficiency are bounded by GPU memory capacity and bandwidth.

Prior work on improving training efficiency mainly focuses on workload scheduling and load balancing [2, 3, 6, 17, 18, 20, 28, 31, 35, 45, 49, 60, 84, 91, 94], GPU memory sharing [38, 43, 56, 91, 95], and model reduction [16, 37, 76, 78, 90]. However, the aforementioned studies often assume requested

amount of GPU memory to be fixed or defined a priori. In practice, there is a significant gap between the amount of memory a training job needs and the amount it requests (§5). Hence, they do not fundamentally address GPU memory inefficiency of training jobs.

A few prior studies present measurement studies of DNN memory usage [6, 19, 38, 91]. However, they are either limited to a single GPU, or only consider data parallelism. We will show in the paper that large-scale distributed training has characteristics distinct from these studied settings. For example, model parallelism has become a norm as model size scales, which changes memory usage assumed by prior work—with data parallelism, memory consumptions are the same across GPUs; however, with model parallelism, memory consumptions differ significantly across GPUs (§6).

We realize that the understanding of GPU memory behavior in large-scale distributed training falls short in the literature. Prior work analyze DNN memory usage [19, 38, 76, 78, 90, 91, 94] based on theoretical modeling, or coarse-grained profiling on small DNNs (with sizes being around 1GB) [14, 23, 82, 85]. There is no public source for researchers and practitioners to understand the actual GPU memory behavior of production distributed training for models in multi-terabyte size and are paralleled on hundreds of GPUs or more [5, 8, 52, 58, 81]. This hinders the research and development of effective GPU memory provisioning, scheduling, and sharing solutions for distributed training at scale.

In this paper, we present a systematic analysis of GPU memory behavior of production training jobs at Meta. Our goal is to fill the knowledge gap and shed light on technical endeavors toward GPU memory efficiency for large-scale distributed training. We further show how to build on the analysis to develop a practical GPU provisioning system to improve GPU memory efficiency for large-scale training jobs in production. Our work is based on Deep Learning Recommendation Model offline training jobs in one of Meta's largest production clusters. Production DLRMs are multi-terabyte in size, consume 50+% of the AI training cycles [1, 24, 51], and serve 60% of the AI inference cycles [21] at Meta.

This paper makes the following main contributions:

- **Measurement of GPU memory inefficiency (§5).** Findings 1-3 show that: half of the production training jobs utilize less than 50% of their GPU memory; 31% of the succeeded jobs had been preempted due to GPU contention, while 9% of the job failures are due to OOM. Our results imply that solutions to GPU memory inefficiency for large-scale distributed training are in eminent demand.
- **Characterization of GPU memory utilization (§6).** Findings 4-6 show characterization of production jobs under mixed parallelisms. Distinct from well-explored data-parallel jobs, memory usage varies significantly across GPUs in jobs with model parallelisms. Memory usages incurred before and in training show distinct patterns, while both constituting the total memory usage in training. We motivate a divide-and-conquer analysis on DNN memory usage, and shed light on effective solutions to GPU memory inefficiency, in large-scale distributed training.
- **Analysis of pre-training GPU memory usage (§7).** Findings 7-9 analyze different segments of GPU memory usage before training, under model parallelism, broadcasting, optimizers, and data parallelism in production. Model parallelism dominates the total pre-training usage. We explore why model parallelisms often balance computation costs while yielding unbalanced model sizes across GPUs. Our results suggest corresponding memory provisioning strategies for different segments of the pre-training usage such as using upper bounds (§7.1) or regression (§7.2).
- **Analysis of training-time GPU memory usage (§8).** Findings 10-13 analyze training-time GPU memory usage and its relations to computation cost balance, batch size, and number of GPUs in production jobs. New memory usage incurred in training varies little across GPUs in a job when the model's computation costs are balanced by parallelism techniques. Our profiling results show that training-time usage can be decomposed as the maximum of the peak memory usages in forward pass (activation), and in backward pass (gradient and autograd).
- **Practical GPU memory provisioning (§9).** We explain the limitations of existing work in distributed training including incomplete memory accounting, substantial profiling overhead or maintenance effort. Driven by the insights of our study, we develop a practical GPU memory provisioning system named AMP, and evaluate it for the most frequently trained models at Meta. AMP can save the number of GPUs by 45% per job on average. We are in the process of deploying AMP in production.

2 BACKGROUND

2.1 Deep Learning Recommendation Model

Deep learning recommendation model (DLRM) is an important class of deep learning models widely used in personalized content ranking and recommendation service [7, 10, 15, 52, 83]. A DLRM trains user data and content and predicts output, such as clickthrough rate (the probability of the user clicking a content). The input data consists of both sparse and dense features: dense features represent continuous data and sparse features represent categorical data.

Like other popular DNNs [5, 14, 23, 52, 81, 89], a DLRM is a sequence of layers, each layer is a dense, high-dimensional tensor. Figure 1 illustrates the canonical DLRM architecture [52, 79, 96]. The model parameters of a DLRM consist of embedding tables, Multi-layer Perceptrons (MLP), feature

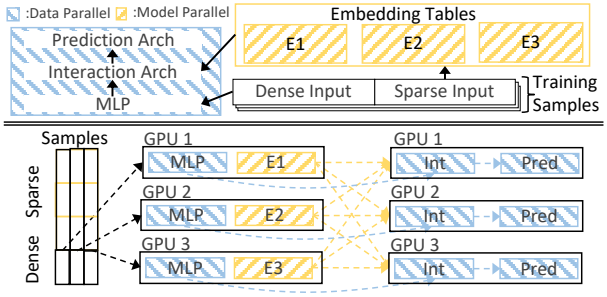


Figure 1: DLRM architecture (top), and its distributed training view (bottom) with data parallelism (blue) and model parallelism (yellow) on 3 GPU trainers.

interaction component, and prediction component. Embedding tables process sparse features into dense representation vectors, and MLP process dense features. The processed features are then fed into the interaction component to produce feature interaction representation, which is then passed into the prediction component to make prediction [52].

The dimension of a sparse feature can often reach billions [79, 96]. So, embedding tables are designed to function as lookup tables that map sparse features into dense, low-dimensional vectors, making them easier for learning. For example, one sparse feature in a training sample could be "*videos clicked in the past 7 days*". The training input of this feature could contain lookup indices to an embedding table that stores the representation vectors of all videos, where each row is a fixed-length vector of a unique video. The looked-up rows are fetched and pooled (e.g., via summation) into a low-dimensional vector that represents the value of the input feature. We refer to all the model parameters (except embedding tables) as *dense parameters*. In a production DLRM, embedding tables are often several terabytes (TBs); dense parameters are hundreds of MBs.

2.2 Distributed Training

Training terabyte-scale DLRMs under high throughput requirements demands significant memory capacity and bandwidth [1, 21, 41, 46, 79]. So, DLRM training simultaneously uses both *model parallelism* to lift the capacity constraint [80, 81] and *data parallelism* to increase throughput [12, 34, 36]. Figure 1 illustrates an example on three GPU training devices, referred to as *trainers*. Model parallelism splits the memory-intensive model component (i.e., embedding tables) across GPU trainers. Data parallelism replicates the compute-intensive model component (i.e., dense parameters) and splits training data across GPU trainers to increase throughput.

Note that the infrastructure challenges of DLRM training in terms of memory capacity and bandwidth also apply to other modern DNN training. Traditionally, DNN training often only needs data parallelism to maximize throughput as models can fit into a single GPU device. However, with the recent advent of large language models and multimodal models [4, 5, 8, 26, 58, 76, 81], terabyte-scale DNNs are becoming popular, making these challenges no longer limited to DLRM training. For example, distributed training for the recent large-scale DNNs must also use model parallelism.

2.2.1 **Training Workflow.** After a training job is configured and submitted to the cluster, it is queued for resources.

Pre-training. The job first undergoes training preparation, which we refer to as the *pre-training* phase. This phase includes model initialization, and several preparation stages:

- "Broadcast": Sharder manifests model parallelism by generating a sharding plan of how to partition the model parallel component (all embedding tables in the model) across all trainers. This stage runs a sharder [87, 88] to generate a sharding plan on the leader trainer, the sharding plan is broadcast globally to other trainers.
- "Sharding": Materializing model parallelism by loading embedding table partitions into trainers' device memory [86].
- "Optimizer": Constructing optimizers for training.
- "DDP": Initializing distributed data parallelism, e.g., replicating dense parameters, across all trainers.

These stages are essential in distributed training. They are often implemented with PyTorch libraries in communication [69], optimization [72], and data parallelism [36, 64].

Training Time. After pre-training phase, the job will start the training phase which runs the distributed training loop. A training loop repeats training iterations hundreds of thousands of times. Like other DNNs, one training iteration of a DLRM is one forward pass followed by one backward pass. Forward pass is computed from the first (input) to the last (output) layer of the model, in which model parameters are accessed to make prediction. Backward pass is computed from the last to the first layer, in which the gradients of the prediction loss with respect to (w.r.t.) the model parameters are computed and used to update the accessed parameters.

A distributed training iteration is as follows. In forward pass (shown in Figure 1), each trainer processes a different set of training samples, every sample has the same features. In each trainer, dense features is processed by the local MLP replica. Meanwhile, sparse features are sent across trainers for embedding table lookup—each trainer fetches and pools rows from local embedding tables and sends to the querying trainers, via an all-to-all communication. Then, in each trainer, the pooled vectors and MLP output are processed by its other local replicated dense parameters to output prediction. In backward pass, gradients of the loss w.r.t. the model parameters will be computed and applied to the parameters. To synchronize the replicated dense parameters, the average gradients are computed and applied to all replicas across

trainers, often through a ring-based or tree-based communication [36]. To update the embedding table, gradients w.r.t. the fetched rows will be sent to the corresponding trainer via another all-to-all communication. Each trainer uses the received gradients to update the fetched rows locally.

2.2.2 **Training Types and Model Placement.** There are two types of training: offline and online training. An offline training job trains a DLRM on large historical data. An online training job recurrently trains an offline-trained DLRM on smaller new data. In production environments, engineers aim to maximize training throughput in offline training—training a large model on as much data as possible per unit time. We focus on offline training jobs as they demand larger memory capacity and bandwidth, hence extensive GPU resources.

During DNN training, the model must reside in device memory [1, 78, 81, 91]. For DLRM, there are several options for placing the embedding tables [1, 79]. The ideal option is to place everything in GPU High Bandwidth Memory (HBM). Since most of the computations during DNN training are done on GPU, this option also eliminates CPU-GPU transfer overhead [78, 91]. Other options leverage Unified Virtual Memory (UVM)—an API from NVIDIA that provides a shared memory address space for host (CPU) and accelerator (GPU) [44]. We can either place all the embedding tables, or the less frequently accessed ones in UVM (i.e., CPU memory). Note that using UVM induces a much lower training throughput, because the memory bandwidth for fetching data from UVM is capped by the interconnect bandwidth of PCIe [46, 79], which is much smaller than HBM bandwidth [53, 55]. So, UVM options are more suitable for online training with lower throughput requirement [79]. In this paper, we focus on large-scale offline training jobs using only HBM. Nonetheless, the memory usage behavior is the same no matter the use of UVM or HBM.

3 DLRM TRAINING AT META

Models. There are hundreds of DLRMs whose offline training jobs are being actively deployed in production at Meta. They differ in model architecture, training objective, serving stage, etc. Each DLRM has an evolving version of the model as a development template. Different jobs training the same DLRM often train variants of the template model. Since many DLRMs have similar architectures, we select three representative DLRMs that have the most distinct model architecture from the top-five most frequently trained DLRMs when qualitative analysis is needed. We refer to them as Model-A, Model-B, and Model-C. Both Model-B and Model-C consist of two sub-models. The sub-models share embedding tables and MLP, with separate feature interaction and prediction components. Model-C is computationally faster than Model-B in training by design. Model-A is often the largest as it consolidates multiple DLRMs. These DLRMs predict clickthrough-rate or click-conversion-rate. Table 1 shows their production model sizes. Each DLRM has 1000+ embedding tables, with total size of 2 to 5TBs. Embedding tables occupy over 99% of the DLRM model size. But, in practice, embedding table size in a DLRM follows a long-tail distribution, e.g., 75% of them are under a couple of hundred MBs.

Table 1: Top-three DLRMs in offline training.

DLRM	Dense Parameters	Embedding Tables	
		#Tables	Size
Model-A	0.46GB	2190	5134GB
Model-B	0.80GB	1037	1988GB
Model-C	0.19GB	1864	4197GB

Table 2: GPU model specifications.

GPU Model	HBM Size	HBM Bandwidth	PCIe
NVIDIA V100	32GB	900GB/s	32GB/s
NVIDIA A100 v1	40GB	1,555GB/s	64GB/s
NVIDIA A100 v2	80GB	1,935GB/s	64GB/s

Hardware. Each machine host in the cluster is equipped with 8 GPU devices (8 trainers), and 1.5TB CPU memory. GPUs on the same host have the same GPU model. Table 2 shows the GPU model specifications.

A training job is highly configurable. ML engineers can configure the number of trainers, training hardware, and model placement options for their jobs (§2). If training hardware is not specified, the system will automatically decide on a GPU model. Each training job only uses one GPU model, i.e., no hardware heterogeneity [31, 49]. Each GPU only dedicates to one job at a time, i.e., no memory sharing due to sharing overhead [38, 43, 91, 95]. Embedding tables are queried from databases based on the model configuration during model initialization. Materialized embedding tables are stored in device memory during training. Model placement optimizations on different layers (GPU, CPU, and SSD) [79, 98] do not directly apply to our setting in this paper (§2.2.2).

Configurations. Batch size and the number of trainers are two training job configuration parameters that have dominant impacts on memory usage of distributed training jobs. They are frequently adjusted by engineers in production. Batch size specifies the number of training samples processed by the model per training iteration—a larger batch size trains the model on more data per unit time to increase throughput, and takes more memory. A larger number of trainers parallels the job on more GPUs, which can also increase throughput, by splitting samples across more trainers in data parallelism (§2). From July to September 2022, the most used batch size for Model-A, Model-B and Model-C offline training jobs is 1024 (89%), 2048 (68%) and 1024 (94%), respectively; the most used number of trainers is 128 for all

three major DLRMs (with occupancy of 90%, 72% and 88%, respectively).; among all DLRMs, the most used number of trainers is 128, and 1024 or 4096 for batch size.

Figure 2 shows the CDF of total GPU memory usage of offline training jobs to their corresponding model size, for the top-five most frequently trained DLRMs in the cluster in November 2022. The total GPU memory usage of a job is computed as the summation of peak GPU memory usage across its trainers. The model sizes of these jobs are multi-terabyte (Table 1). Figure 2 shows the total GPU memory usage in 70% of the jobs are 2-4 times larger than their model size. GPU memory inefficiency in distributed training is easily magnified at this scale.

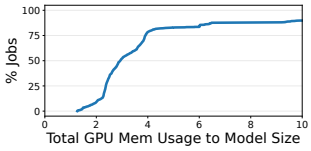


Figure 2: Ratio of total GPU mem usage to model size.

GPU Provisioning Practices. Production ML engineers have three main objectives in DNN training: (1) maximizing training throughput, (2) minimizing the number of trainers for resource efficiency, and (3) avoiding job failures (e.g., due to OOMs). Effective GPU provisioning should meet these three objectives. In this context, GPU memory is a major consideration of GPU provisioning for large-scale distributed training jobs, because training scalability and efficiency are primarily bounded by memory capacity and bandwidth [1, 3, 46, 76, 79]. However, a GPU's memory is limited.

We observe that ML engineers provision GPUs for their training jobs *manually* by manually deriving and setting the number of trainers. If this job fails due to OOM, the engineer must decide on a larger number of trainers for the job, and deploy it again. To reduce the number of failed trials, engineers often fine-tune the default number of trainers for a specific model based on their experience, and use the default value for future training jobs of that model.

The experience-based approach can hardly be accurate or keep up with the evolving models [92, 93]. In practice, engineers tend to over-provision GPUs to avoid job failures due to OOMs, just like any other hardware resources [9, 13, 40, 42, 77]. Our work is motivated by the need for an automatic GPU provisioning tool to accurately estimate the amount of GPU memory needed for a given job in order to improve GPU memory efficiency in production.

4 METHODOLOGY

Our study is based on data of DLRM offline training jobs from one of Meta's largest production clusters. The data range spans from February to December 2022; each part of the study is based on data spans of least 30 consecutive days.

Memory Efficiency Data. Our analysis on GPU memory efficiency (§5) is based on recorded resource monitoring data. The data records hourly-logged GPU memory utilization of all the deployed offline training jobs in the cluster for a limited time span. To analyze OOM events, we collect all the failed jobs due to OOM or sharding errors based on automated log analysis of error messages.

Monitoring. To conduct a more fine-grained analysis of GPU memory utilization (e.g., GPU memory utilization over time and at different stages, §6-8), we add GPU memory monitors [68] to the production training infrastructure to collect per-trainer memory utilization for all the deployed DLRM training jobs. The memory utilization of a GPU trainer is logged every 30 seconds (finest granularity without affecting production SLO) from model initialization to the end of the training loop. For example, to study GPU memory utilization before the training loop (§7), we signpost GPU memory utilization at important stages that occur before the training loop. We collect the per-trainer memory utilization right before and after the execution of each stage.

Profiling. We conduct experiments to profile memory usage over time for tensors computed in the forward and backward pass. We randomly sample 16 DLRM offline training jobs completed in November 2022 in the cluster—four jobs per each of the following DLRM: Model-A, Model-B, Model-C, and DHEN [97]. DHEN is included as a sub-model in Model-B (§3). For all 16 job samples, we profile memory behavior of their models for two training iterations. We empty the CUDA cache in between the two iterations [67]. We use the PyTorch profiler [73], which provides a decomposition of the allocated memory by tensor functionalities. We use profiling results from the second iteration to avoid temporal noise from bootstrapping the profiler and the training loop. For each sample, we control the profiling experiments on three aspects: (1) with or without materialized embedding tables, (2) varying three batch sizes, (3) single GPU versus single CPU. We have 12 ($2 \times 3 \times 2$) experiment runs per sample.

5 GPU MEMORY INEFFICIENCY

Recall from §3 that, without an effective GPU provisioning tool, engineers tend to over-provision GPUs to avoid job failures due to OOMs. While over-provisioning could benefit individual jobs, it leads to significant inefficiency of cluster-level GPU resources. In this section, we quantify the inefficiency in terms of GPU memory.

Finding 1. *50% of the DLRM offline training jobs utilize less than 50% of their provisioned GPU memory in production.*

The peak GPU memory utilization of a job is the peak GPU memory utilization averaged across its trainers, over the job's duration. The peak GPU memory utilization is below 50% for half of all the DLRM offline training jobs. Specifically, the 25th, 50th, 75th and 90th percentiles of peak utilization

are 25%, 43%, 71% and 95%. Under-utilization is more severe on jobs that are configured to use more trainers: less than 10% of the jobs that use more than 150 trainers had reached 50% peak GPU memory utilization. Engineers could intentionally over-provision GPUs for high throughput, e.g., to quickly test model accuracy. We opt for reducing cluster GPU memory inefficiency while meeting throughput requirement.

Finding 2. *31% of the failed DLRM offline training job attempts are due to preemption; 22% of the succeeded offline training jobs have been preempted at least once.*

Over-provisioning directly worsens GPU scarcity in production. Training jobs preempted due to GPU contention against a higher priority job are marked as failure, and will be requeued for resource. 31% of failed DLRM offline training job attempts are due to the job being preempted during execution for a higher-priority job. Further, 22% of the successfully completed DLRM offline training jobs have been preempted and retried at least once. The production cluster thus emits a behavior of *false idleness*: jobs often get preempted due to GPU contention, while majority of the jobs only utilize less than 50% of their provisioned GPU memory at best.

Finding 3. *8.67% of the failed DLRM offline training jobs are caused by GPU OOM in production.*

Although engineers often tend to over-provision GPUs, we still observe a fair amount of job failures due to under-provisioning. A job will fail due to GPU OOM when the memory usage on any one of the trainers exceeds its GPU memory capacity during training. We observe the percentage of GPU OOM in training job failures among Model-A, Model-B, Model-C, and all DLRMs are 15.13%, 16.39%, 6.94%, and 8.67%, respectively. 500,000+ GPU hours from the cluster were wasted within a 90-day range from July to September 2022. We also examine the occurrence of sharder failures due to GPU memory under-provisioning. A sharder fails if it cannot determine a sharding plan because the configured amount of trainers have insufficient memory to store embedding tables. Sharding failures are more lenient than OOM because they occur before training. Sharder failures constitute about 0.5% of the job failures, and wasted about 30,000 GPU hours from February to October 2022.

Implication. GPU memory under-utilization and OOM are prevalent in DNN training in open-source and production settings [19, 29, 30, 32]. We observe that large-scale distributed training simultaneously suffers from these consequences of GPU memory inefficiency. The demand for effective GPU memory provisioning is thus eminent for distributed training at scale, and has not been addressed in practice.

Distributed training jobs in production range from hundreds to thousands of GB in model size, and use tens to hundreds of GPUs in parallel. Accurately provisioning for such a diverse volume of jobs is challenging, because their GPU memory usage could be affected by various factors—parallelism paradigms, feature density, model architectures, etc. Depending on these factors, GPU memory usage differs significantly across jobs, and even across trainers of the same job. Moreover, a training job can carry arbitrary code and configuration changes, making manual solutions untenable. Most prior studies only provide theoretical reasoning on memory usage under limited scenarios (e.g., single-device). Unfortunately, the lack of understanding of actual GPU memory usage behavior of large-scale production workloads hinders solutions to GPU memory inefficiency in practice.

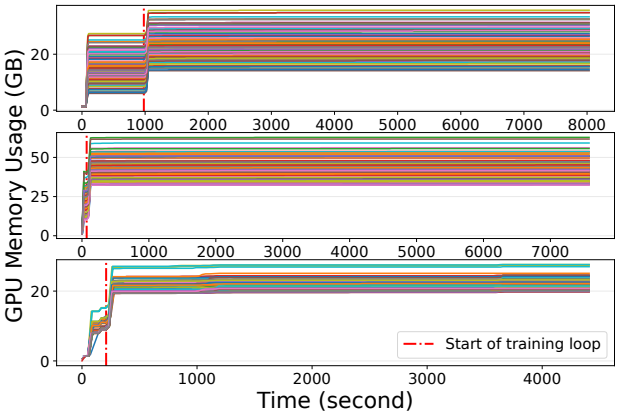


Figure 3: GPU memory utilization from model initialization till training ends for 3 randomly-sampled jobs. Top to bottom: Model-A, Model-B, and Model-C.

6 GPU MEMORY UTILIZATION

In this section, we conduct a fine-grained analysis of GPU memory utilization following §4. We calculate the GPU memory utilization of a trainer every 30 seconds as its *maximum* GPU memory usage within this 30-second window.

Finding 4. *GPU memory usage often varies significantly across trainers within a distributed training job in production.*

Figure 3 shows GPU memory utilization of three randomly sampled offline training jobs, one per our example DLRMs (§3). Each plot has 128 lines, each line shows the GPU memory utilization of a trainer. In each job sample, GPU memory usage varies significantly across trainers. This is common in jobs with model parallelism. On average, GPU memory usage of the most memory-consuming trainer is 47% higher than the average GPU memory usage across trainers in a job. This large variance is mainly caused by embedding tables varying across trainers (§7.3 provides detailed analysis).

Implication. GPU memory provisioning for large-scale distributed training should focus on the peak GPU memory

usage of the most memory-consuming trainer (especially for OOM prevention), instead of the average across trainers [19].

Our finding also shows that training jobs often do not use up the entire memory of every GPU. This presents potential for GPU memory sharing and job co-location for distributed training with model parallelism in practice, especially with the emergence of GPUs with larger HBM size [57]. Trainers with lower memory utilization can share their GPU memory within their GPU memory capacity. It also indicates opportunities for utilizing heterogeneous accelerators for distributed training under model parallelism. To achieve higher GPU memory utilization, GPU trainers with low utilization in a job can use GPU models that have a lower memory capacity during provisioning and scheduling stages. Existing work is limited to single-device or data parallelism-only scenarios, where GPU memory usage across trainers varies little.

Finding 5. *Most trainers stabilize at their peak memory usage early. In over 60% of the jobs: trainers reach their usage peak when the job is completed at most 40%, the most memory-consuming trainer reaches its usage peak when the job is completed at most 20%. See Figure 4.*

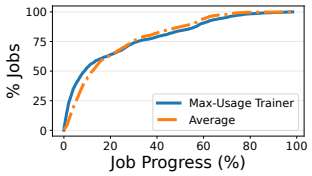


Figure 4: Job progress at 99% peak GPU mem usage.

In Figure 3, the GPU memory usage on each trainer of every job stabilizes quickly with minor fluctuation. Figure 4 shows how long a DLRM offline training job takes for its trainers to reach 99% of their peak GPU memory usage. "Max-Usage Trainer" shows how long the most memory-consuming trainer takes to reach 99% of its peak usage. "Average" shows how long *on average* all the trainers of a job take to reach 99% of their peak usage. Knowing the usage change over time for the most memory-consuming trainer in a distributed training job is important for GPU OOM prevention. In 60% or 80% of the jobs, the average trainers and the most memory-consuming trainer reach peak usage when the job is completed less than 20% or 40%, respectively. The minor bumps after usage stabilized are mainly caused by dynamic caching activities of the tensor allocator from PyTorch, which incrementally builds up a cache of CUDA memory and reassigns it to later allocations during training [59].

Implication. If peak GPU memory usage were highly unstable in large-scale distributed training, memory provisioning, sharing and balancing would have been difficult. We observe this is not the case. Building solutions to GPU memory inefficiency in large-scale distributed training is thus achievable.

Finding 6. *The memory usage incurred before and during the training loop emit distinct patterns. The memory usage incurred before training loop starts continues to constitute the peak overall memory usage during the training loop.*

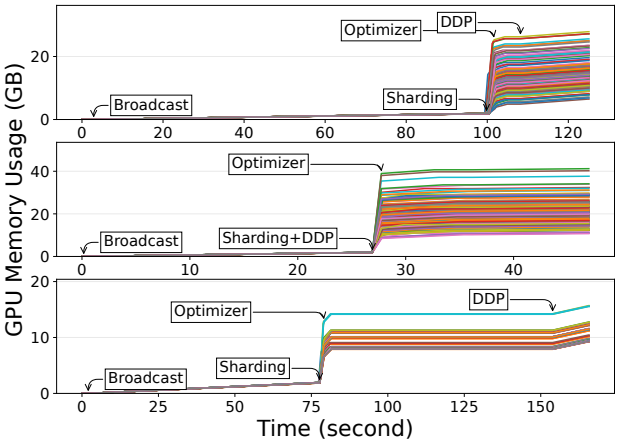


Figure 5: GPU memory usage over time prior to training loop for the same jobs in Figure 3.

We revisit Figure 3 to further analyze the peak overall GPU memory usage. Red vertical line separates the pre-training and training phases in each job (§2.2.1). These two phases have different memory usage patterns. Before training, there is a major memory usage increase, in which the memory usage on each trainer becomes significantly different. This increase is mainly due to model parallelism—embedding tables are partitioned and placed into each trainer's GPU memory. During training, there is another major memory usage increase on each trainer again. But the increased amount is approximately the same across trainers, and quickly stabilizes. This increase occurs as model starts running forward and backward pass with training samples on each trainer. Overall, memory usage incurred before training continues to constitute the total memory usage during training.

Implication. Our result motivates a divide-and-conquer approach to analyze the overall GPU memory usage. Distinct usage patterns before and during training prompt separate analyses of these two phases. And since they both contribute to the overall peak usage throughout training, their analysis results can be aggregated into analysis for the overall usage.

7 PRE-TRAINING GPU MEMORY USAGE

In this section, we study GPU memory usage incurred prior to the training loop (referred to as the *pre-training* phase).

Figure 5 shows GPU memory usage at important pre-training stages for the same jobs in Figure 3. The execution order and the memory usage at each stage could vary by models and jobs, because a training job can carry arbitrary changes from its engineer. From Figure 5, we observe that the memory usages for each stage together constitute the overall peak memory usage in the pre-training phase. Each stage

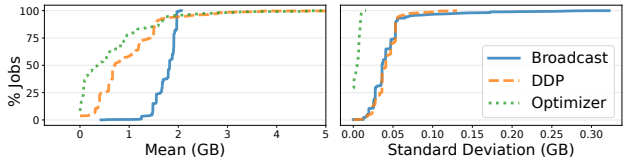


Figure 6: CDF of mean and stdev of GPU mem usage across trainers for "Broadcast", "Optimizer", "DDP".

also shows similar memory usage pattern across jobs. This observation motivates us to analyze each stage separately.

7.1 Broadcasting

Finding 7. *The memory usage for broadcasting in a distributed training job is within constant limit, while it can be positively affected by the number of trainers. See Figure 6 and 7.*

The "Broadcast" stage runs sharder and broadcasts metadata across trainers. Its memory usage is related to the size of the metadata, which includes sharding plan, data transformation information, and model configuration. Because the metadata is used in trainer communication, it constitutes the peak GPU memory usage of each trainer during training. We find metadata to be small so the memory usage for this stage is capped under a certain limit.

Figure 6 shows that the average GPU memory usage across trainers for "Broadcast" is almost the same in various jobs. Figure 7 shows GPU memory usage for "Broadcast" averaged across jobs under different numbers of trainers. "Broadcast" memory usage is overall higher in jobs that use more trainers, because metadata carries information of all trainers.

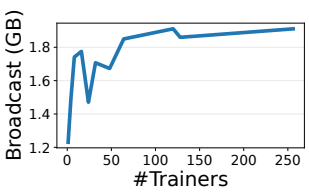


Figure 7: "Broadcast" GPU mem usage vs. #trainers.

Implication. We may account for "Broadcast" memory usage during provisioning by setting a constant upper-bound.

7.2 Optimizer and Data Parallelism

Finding 8. *The memory usages for constructing optimizer(s) and data parallelism both vary little across trainers. They are both linear to the model's dense parameters size. See Figure 8.*

The "Optimizer" stage constructs optimizer objects that will be used to update the model parameters during training [72]. The constructed objects hold the optimizer state (e.g., momentum [74], moments of the gradients [33]), and update the model parameters based on the computed gradients. Figure 8 compares the size of the dense parameters with the GPU memory usage of the "Optimizer" stage for 100 jobs. It shows that the size of the optimizer state is proportional to the size of the parameters it optimizes on, which is the size of the

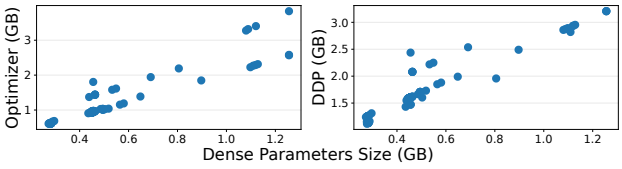


Figure 8: The size of dense parameters versus the max GPU memory usage across trainers for the "Optimizer" and "DDP" stages on 100 randomly-sampled jobs.

model's dense parameters in this case. Figure 6 further shows that the memory usage for "Optimizer" is the same across trainers within a job, as each trainer starts with optimizers that have the same initial state before training starts. Note that trainers here refer to GPUs, not the optimizer.

The "DDP" stage initializes data parallelism using DistributedDataParallel from PyTorch [36], which replicates the dense parameters across trainers, and creates necessary objects for gradient synchronization [64]. As shown in Figure 8, in each training job, the GPU memory usage for initializing DistributedDataParallel is approximately linear to the size of the dense parameters. We can also see from Figure 6 that the GPU memory usage of this stage has little variation across trainers in various jobs.

Implication. The memory usage for data parallelism and optimizer construction in distributed training jobs can be predicted linearly to the size of the model's dense parameters.

7.3 Model Parallelism

Finding 9. *Model parallelism incurs large difference across trainers on their memory usage for storing the model parallel component. In production, the standard deviation is up to 15GB, and the difference is up to 50GB, across trainers within the same distributed DLRM offline training job. See Figure 9.*

Recall from §6 that, GPU memory usage often varies significantly across trainers in a distributed training job. This is mostly due to the size of the model parallel component (e.g., embedding tables in DLRM) vary significantly across trainers under model parallelism. Figure 9 shows that the range and standard deviation of embedding table size across trainers in a job can be up to 50GB and 10GB, respectively. In 68% of the jobs, trainer storing the largest embedding table partition is the most-memory-consuming trainer.

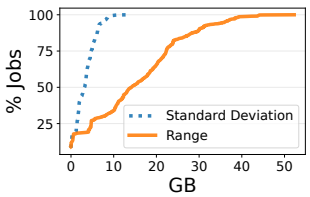


Figure 9: Range and stdev of Emb size across trainers.

In essence, model parallelism techniques often do not balance model size across trainers. In DLRM training specifically, embedding table lookup accounts for half of the total computation and communication cost [96]—they are accessed

across trainers in forward pass; their gradients are applied across trainers in backward pass (§2.2.1). Each embedding table largely differs in table-wise characteristics, e.g., dimension, lookup frequency (times a table is accessed), pooling factor (fetched rows per lookup), coverage (training samples accessing a table). Thus, how to shard terabytes of embedding tables across trainers significantly impacts training throughput. Currently, sharder's primary goal is maximizing training throughput, by partitioning the model such that *computation and communication costs of the partitions are balanced across trainers*. Finding an optimal sharding plan is NP-hard.

In existing sharders, model size (and its balance) is not a primary consideration with the goal of throughput maximization. To generate a sharding plan for DLRM, sharders use heuristic driven algorithms to partition and place embedding tables across trainers. Embedding tables can be partitioned at a mixture of column-, row- and table-wise granularities. Existing sharders [1, 41, 46, 79, 96] use various heuristics (e.g., table characteristics, hardware specifications) to define sophisticated cost functions; the optimization algorithms often have stochastic outputs (e.g., reinforcement learning). Under such design, lookup frequency and pooling factor are more effective as cost to be balanced towards generating a throughput-maximized sharding plan, as they directly represent the access frequency and access volume of an embedding table. But, lookup frequency and pooling factor often do not associate with table size. Figure 10 shows the normalized pooling factor, lookup frequency and size of 10% of the embedding tables in our studied jobs. A larger table does not mean it is used more frequently in training. Consequently, while pooling factor and lookup frequency are more likely to be balanced as part of the cost across trainers by sharders, embedding table size is not.

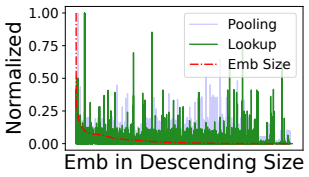


Figure 10: Emb table stats.

Note that some sharders use hierarchical sharding to balance model size across hosts to mitigate heavy inter-host communication. Hierarchical sharding assigns each host a similar load so that the inter-host communication is not bottlenecked by a few hosts that hold much more model parameters than the others. Figure 11 shows an offline training job sample with 128 trainers (16 hosts). Each dot shows the embedding table size on a trainer. We can see the embedding table size is approximately balanced across hosts.

Implication. Model parallelism techniques (i.e., sharders) often behave stochastically, and are too intricate to be modified for other purposes (e.g., provisioning). Instead, we can bypass handling the sharder's complexity while accounting for memory usage from model parallelism, by providing GPU memory provisioning results as input to the sharder to iteratively guide sharding and provisioning decisions. Since computation cost of a model parallel component may not correlate to its size, we believe future techniques on model parallelism should incorporate the goal of balancing the model parallel component size across trainers, which can reduce job-wise memory utilization imbalance in large-scale distributed training, and lead to higher training cluster efficiency.

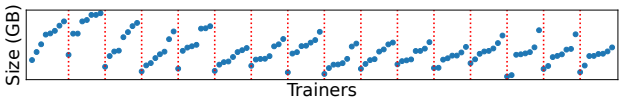


Figure 11: Model partition size per trainer of a random Model-A job with 128 trainers. Red lines separate hosts.

8 TRAINING-TIME GPU MEMORY USAGE

We now study the *new* GPU memory usage incurred during training. Note that pre-training usage (§7) still constitutes the overall GPU memory usage during training (§6). We count the peak training-time usage during job execution.

Finding 10. *Training-time usage varies little across trainers within a job. In 99% of the DLRM offline training jobs in production, the standard deviation is below 2GB. See Figure 12.*

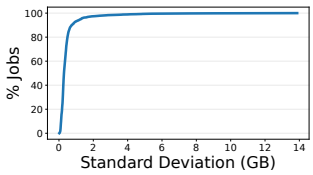


Figure 12: Training-time usage stdev across trainers.

The balance of training-time usage is attributed to the computation cost balance across trainers. Since the training phase only consists of running the distributed training loop, training-time usage is essentially the memory allocated for tensors generated while computing the forward and backward pass in each training iteration (§2.2.1). In DLRM training, the compute-intensive component (dense parameters) are replicated across trainers. So the training-time usage for computing forward/backward pass on the dense parameters are the same across trainers. Meanwhile, the capacity-demanding component (embedding tables) is partitioned across trainers by the sharder. As sharder optimizes computation-cost balance of embedding tables across trainers (§7.3), the training-time usage for embedding table tensor computations (i.e., lookup, pooling, weight update) during forward/backward pass is also opted for being balanced.

Implication. Since trainers in a distributed training job often have the same training-time usage, we can apply the same analysis and techniques to estimate training-time usage across all trainers, rather than tackling it on individual trainers separately during GPU memory provisioning.

Balancing training-time usage across trainers offers convenience for tackling system problems in distributed training that often need memory estimation, e.g., GPU memory provisioning, scheduling, sharing, and balancing. As long as the model parallelism techniques can effectively balance the computation cost across trainers, training-time usage across trainers should be similar within a distributed training job.

8.1 Impact of Configurations

Finding 11. *Training-time usage variance across trainers in a job is primarily due to the variance of computational cost (e.g., lookup frequency and pooling factor in DLRM) across trainers.*

We found lookup frequency and pooling factor to be highly correlated—more frequently accessed embedding tables also have more rows fetched per access. Further, the lookup frequency and pooling factor of an embedding table is often significantly disproportional to its size. A small number of tables have higher access frequency and volume than the others in a DLRM, but are often not the largest in size (§7.3).

A trainer that stores embedding tables with higher lookup frequency and pooling factor should have a higher training-time usage, since more rows will be fetched/updated more frequently during training. Indeed, we observe a 0.30/0.13 Pearson coefficient between lookup frequency/pooling factor and training-time usage at trainer level. Since mixed use of model parallelisms (e.g., row-, column-, table-wise) introduces noise towards aggregating pooling factor and lookup frequency per trainer, we further validated the causality between trainer-wise lookup frequency (and pooling factor) and training-time usage using linear mixed effect model—a standard linear regression model used to determine the causal relationship between multiple variables.

Implication. Under model parallelism, imbalance of model parameter access frequency (e.g., lookup frequency) and access volume (e.g., pooling factor) causes the training-time usage to vary across trainers. Sharder often balances them (§7.3) and achieves a small training-time usage variance. Most often we can omit these factors while provisioning training-time usage. However, more fine-grained solutions should account for their influence to training-time usage, e.g., by iteratively optimizing provisioning decision with sharding decision until both reach convergence. Our results also motivate future work in feature engineering to design learning features with access frequency/volume proportional to their sizes for large-scale DNN, which helps to reduce sharding and load balancing complexity in distributed training.

Finding 12. *Training-time usage and its variance across trainers is linear to batch size. Meanwhile, part of training-time usage comes from communication overhead, which grows by the number of trainers within a constant limit.*

Table 3: Tensor categories.

Category	Description
Parameter	Model parameters used in forward pass, e.g., embedding tables and dense parameters (§2.1).
Gradient	Derivatives of prediction loss w.r.t. the model parameters.
Autograd	Tensors generated in backward pass that are not "Gradient" [62]. They are generally intermediate derivatives from the chain rule or implementation details of `Autograd`, e.g., accumulation buffers [61].
Activation	Tensors generated in forward pass, and is ultimately used to compute the gradient tensors.
Input	Tensors that contribute to forward pass and gradient computation, but do not depend on other tensors.
Temporary	Tensors created and destroyed in a single operator node.
Optimizer	Tensors held in the optimizer state which are used when an optimizer updates the model parameters.
Unknown	Tensors unfit for categories above, or memory allocation not matching a tensor's storage implementation [66].

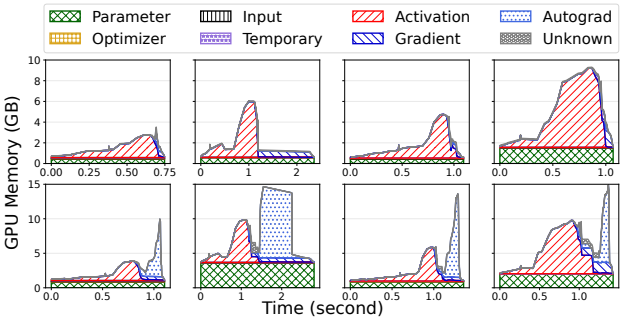


Figure 13: GPU memory usage decomposition by tensor functionality in one forward and backward pass. Each column from left to right shows one job sample for Model-A, Model-B, Model-C and DHEN. Jobs at the second row have embedding tables materialized as part of the model parameters, jobs at the first row do not.

We randomly selected one production offline training job per sample DLRM (§3), and deployed them with different batch sizes and number of trainers. Our experiments show the max, min, average, and variance of training-time usage across trainers are linear to batch size. The slope is different in each job due to model differences. Communication across trainers is frequently performed in each training iteration (§2.2.1), via PyTorch `AllReduce` [36] and NCCL [54]. Our experiments show communication overhead can grow from 1.5GB to 2GB per trainer when the same job uses more trainers.

Implication. We can provision training-time usage linear to batch size in distributed training jobs [19, 39, 78, 94]. The communication usage can be provisioned with a constant.

8.2 Decomposing Training-Time Usage

We further decompose training-time usage by profiling the memory usage of tensors computed in the forward and backward pass over time. We categorize the allocated memory

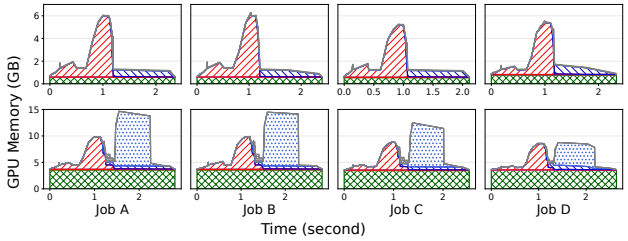


Figure 14: Four Model-B jobs under the same batch size. First/second row is without/with embedding tables.

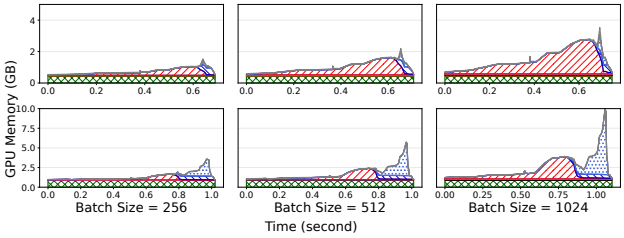


Figure 15: A Model-A job on batch size 256, 512 and 1024. First/second row is without/with embedding tables.

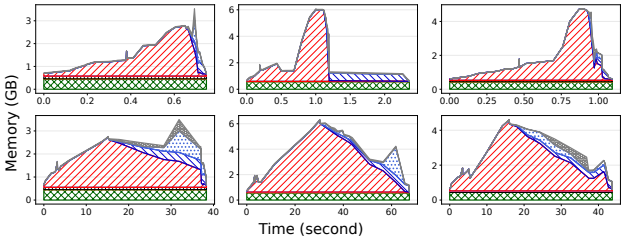


Figure 16: Three jobs without embedding tables running on GPU (first row), versus on CPU (second row). Left to right: Model-A, Model-B, Model-C.

by tensor functionality (see Table 3). Figure 13 presents the profiled GPU memory of four jobs, one per DLRM. All jobs use batch size 1024. Each color pattern indicates memory allocated for a tensor category.

Finding 13. *The peak training-time usage in each training iteration is dominated by the maximum of the peak forward pass memory usage and the peak backward pass memory usage.*

Figure 13 shows similar training-time usage pattern across different DLRMs. Activation, gradient and autograd tensors dominate the peak memory usage in training [38, 78, 90, 91]. In forward pass, activation tensors accumulate, whose memory usage surges quickly near the end of forward pass. This surge is mostly caused by large prediction head. In backward pass, autograd and gradient tensors w.r.t. the previously-computed activation tensors and parameters are being computed. Activation tensors are released after their corresponding gradients are computed. Gradient tensors are released after being applied to update parameters. Autograd tensors are not released until after the end gradients are derived [61]. Model parameters and input are kept until after training ends; memory usages of optimizer, temporary and unknown tensors are small. From Figure 13, the peak training-time usage equals to the maximum of the peak usage in forward pass and the peak usage in backward pass.

Training-time usage vastly differs across models, but varies little across jobs for the same model type. Figure 13 shows that each model has a different training-time usage peak and allocation trajectory, because model architecture governs the amount and size of tensors that will be traversed per training iteration. However, jobs for the same model type show similar pattern in Figure 14 (training jobs of the same model type often have small architectural changes).

In addition, we draw the following observations:

- In Figure 13 (first row), without materializing embedding tables, the memory bottleneck lies in caching and computation of the data parallel component. There is only one memory usage summit per training iteration, similar to profiling results on canonical DNNs [38, 91, 95]. In Figure 13 (second row), embedding tables are materialized in backward and forward pass. The memory bottleneck lies in caching and computation of the model parallel component. They largely increase the memory usage in backward pass, since many autograd tensors are held until gradients of the fetched table rows are derived. With embedding tables, there are two memory usage summits per iteration.
- The usage decomposition in Figure 15 shows that training-time usage is overall linear to batch size because both the peak usage for activation, and the peak usage for gradient and autograd, are linear to batch size.
- Figure 16 shows the peak training-time usage between GPU and CPU to be the same. But the usage accumulation trajectory differs, as activation and autograd tensors are held longer due to slower tensor computation on CPU.

Implication. Finding 13 offers a way to provision a model's training-time usage under given batch size. Because training-time usage pattern varies little across production jobs of the same model type, we can extend the provisioning system on a model basis rather than job basis. Since peak training-time usage is largely hardware-independent, we can omit hardware difference when provisioning training-time usage.

Our results also suggest that computation of the model parallel component incurs high memory usage in backward pass, resulting in multiple adjacent memory usage summits per training iteration. Existing memory sharing, or scheduling algorithms for DNN training jobs typically assume one isolated summit per iteration, with enough time in between to overlap multiple jobs on a single device [38, 91, 95]. Our results motivate future work on these directions to further account for this factor in model parallelism.

9 GPU MEMORY PROVISIONING

Recall from §3 that, manual GPU provisioning by engineers can hardly be accurate or keep up with evolving models, which leads to GPU memory inefficiency in large-scale distributed training (§5). In this section, we describe our experience of devising an effective GPU memory provisioning system for distributed training in production at Meta. Our system aims to achieve the following goals:

- *Accurate* provisioning of minimum GPU memory to meet throughput requirement, while avoiding OOM;
- *Automatic* provisioning without manual configuration or maintenance effort;
- *Generally applicable* to training jobs of different models;
- *Efficient* with little runtime overhead onto training.

9.1 Limitations of Existing Techniques

We started by exploring the viability of adopting existing work—a number of prior studies have presented approaches for estimating GPU memory usage for DNNs [2, 19, 39, 84, 94]. However, we find it challenging to adopt them in our production systems. From a high level, existing DNN memory usage estimation techniques can be categorized as follows:

9.1.1 **Modeling.** Several scheduling techniques estimate DNN memory usage with coarse-grained modeling derived from theoretical reasoning [2, 94]. They estimate a DNN's memory usage as the size sum of its parameters, input, activation and gradient tensors. More fine-grained estimation techniques [19, 84] define memory cost function (MCF) per operator. They define an operator's MCF as the sum of its weight, output and ephemeral tensor sizes (input, parameters, activation, gradient and temporary tensors in §8.2). These techniques statically traverse DNN graph, get memory cost at each operator by summing the sizes of its alive tensors according to its MCF (an alive tensor is one used by any current or descendent operator), then output the max memory cost across nodes as the estimated memory usage.

Limitations. Existing models miss important factors of the DNN memory usage. Coarse-grained models [2, 94] miss important components from the learning framework and training system, e.g., broadcast, optimizer (§7), and autograd (§8). Fine-grained models (i.e., MCFs) are static, and are derived by theoretically reasoning the relationship between the parameters and memory usage of each operator [19, 84]. However, operators in production are diverse and customized. Some operators have complex definitions, with stochastic behavior, e.g., the memory usage relation of embedding table operators varies by learning features [46]. Hence, hardcoding operator-wise models are brittle. Also, existing models cannot account for autograd tensors—their memory usage is dominant in backward pass and is unknown before execution (§8.2).

9.1.2 **Profiling.** Several scheduling techniques obtain memory usage of DNN jobs by performing fine-grained profiling on the jobs during training [18, 31, 35, 91]. They profile either the entire job, or smaller sample jobs of the actual job before it is scheduled, often at second or millisecond granularity.

Limitations. Profiling the training loop adds non-negligible overhead to training efficiency at production scale. A training iteration usually takes less than a second, job- or GPU-wise profiling at second granularity slows down the training loop for the vast volume of jobs that need profiling in the cluster. From our experience, profiling smaller sample jobs cannot provide accurate memory usage statistics for provisioning due to various shrinkage to the large-size model and training configurations of the actual job.

9.1.3 **Testing.** One technique [39] leverages test-case DNN generation and neural network formal specifications to estimate a model's memory usage. For a particular DNN (e.g., VGG [82]), it generates test-case DNNs by combinations of sample parameter inputs to each operator in the DNN. It then selects valid test-cases with formal specifications of the DNN architecture, obtain memory usage of each test-case via profiling or static analysis, and learns a polynomial regression on these test pairs to predict memory usage.

Limitations. With sophistical production model architectures [97], generating comparable test-cases is hard. Estimating memory usage of these test-cases also demand extra endeavors, i.e., profiling or static analysis [39]. Further, developing formal specifications of production models for test-case validation requires manual effort, and is difficult to keep up with the evolving models.

9.2 AMP: A Practical Provisioning System

We developed a practical GPU memory provisioning system called AMP for production DLRM training at Meta. AMP is an analytical approach built with open-source PyTorch.

9.2.1 **Design and Implementation.** AMP runs at model initialization. To provision GPU memory for a training job, AMP automatically estimates its model's peak memory usage given throughput requirement specified by engineers (batch size), and configures its number of trainers before training.

Design. Peak memory usage on a trainer is constituted by storing the model parallel component, and the other memory usage. Shown in Figure 17, AMP first estimates the *other memory usages* as the *reserved memory* of each trainer, and feeds it as input to the sharder. Sharder generates a sharding plan for model parallelism using the remaining memory (original capacity minus reserved memory) on each trainer. In this way, AMP bypasses interfering sharder's stochastic optimizations (§7.3). If sharder fails, AMP provisions more trainers from a small default value until sharder succeeds.

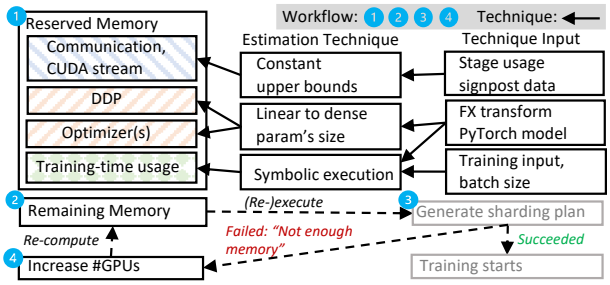


Figure 17: Provisioning system AMP first estimates the reserved memory, then iteratively provisions GPUs if remaining memory (memory capacity minus reserved memory) cannot fit the model parallel component.

AMP automatically estimates the reserved memory via divide-and-conquer ("Technique" in Figure 17). Peak memory usage accounted by the reserved memory is constituted by usages from pre-training stages, and training-time usage (§6). AMP estimates each of them separately:

- Peak memory usages for communication and CUDA stream are accounted by constant upper-bounds (§7.1, §8.1).
- Peak memory usages for "DDP" and "Optimizer" are estimated linear to the model's dense parameters size (§7.2).
- Training-time usage is estimated by symbolically executing the model under FX transformation [70] with training sample batches. AMP finds and compares the memory usage peaks in forward and backward pass via symbolic execution, and uses the higher peak as the estimate (§8).

Implementation. The upper-bounds for communication and CUDA stream usage are updated periodically by querying memory usage signposts at the start and end of the pre-training stages of recent historical job samples.

During model initialization of a job, we transform the model into a `torch.fx.GraphModule` instance, which is a graph intermediate representation of the model where each node represents a callsite to entities such as operator [70, 71]. We calculate the size of dense parameters on the transformed model to estimate memory usage for "DDP" and "Optimizer".

To estimate training-time usage, we symbolically execute the transformed model with raw training input batches. We use `ShapeProp` to execute the graph node-by-node with the given arguments, and get the output tensor's metadata (e.g., shape, data type, `require_grad`) of each node [65]. In the forward traversal, we compute the size-sum of output tensors that have their `require_grad` set to true as the *forward usage estimate*. This estimates the peak memory usage for activation. In the reversed traversal, we compute the maximum size-sum of output tensors used downstream at any given node in the graph as the *backward usage estimate*. This estimates the peak memory usage for gradient related tensors. The final estimated training-time usage is the maximum of

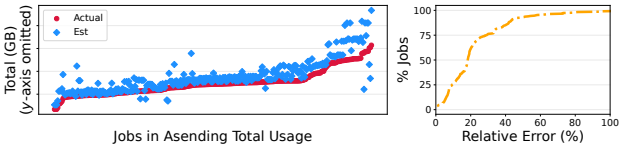


Figure 18: Actual versus estimated total GPU memory usage of 300 offline training jobs for top-five DLRMs.

these two entities (§8.2). Symbolically executing the model graph with original batch size (e.g., 1024, 2048) can cause OOM. So we compute estimates from 10 micro batch sizes (e.g., 13, 43), and use them to predict the final estimate (Finding 12). The estimation process takes a few seconds.

Summary. AMP provisions GPU memory by estimating and tuning the job configuration before training. It is thus easy to be incorporated into AutoML frameworks, e.g., for parameter auto-tuning, based on our experience in production.

AMP is independent of specific profilers, it only adds a one-time overhead to the job before training starts. AMP regularly queries memory usage signposts of pre-training stages from recent job samples (§3) to update upper-bound estimations with near-zero overhead. In comparison, profiling memory usage of a training job adds constant overhead to the job. AMP can account for memory usage from all important tensor categories missed by coarse-grained models (§8.2) with symbolic execution. AMP also does not need sophisticated models (i.e., operator-wise MCFs), because it symbolically executes each operator with training input and gathers their respective tensor sizes to compute usage peaks (§8.2). AMP requires no DNN generation and verification.

From our experience, AMP can be extended to other model types. If the model type has a new custom tensor type (which is uncommon), we need to add it to the tensor size-sum calculation when estimating training-time usage. Other usages differ little by model type. If a model type uses a new optimizer, we'll need to understand the optimizer as in §7.2.

AMP does not consider training-time usage variance across trainers. In practice, this variation is mostly very small (Figure 12), but it can be larger if computation cost is unbalanced (Finding 11). We can potentially address this issue by integrating sharding logic into provisioning. AMP also does not consider CUDA caching behavior which has minor impacts on total memory usage (§6).

9.2.2 **Evaluation.** We evaluate AMP in production for top-five DLRMs (§3). Figure 18 shows relative error ($\frac{100(Est-Actual)}{Actual}$) between the estimated and actual total GPU memory usage of the most memory-consuming trainer on 300 randomly sampled production offline training jobs in December 2022. The actual usage is obtained via monitoring (§4). For these jobs, the 75th percentile of symbolic execution cost is 2.2 seconds. The left plot compares actual and estimated usages,

where jobs are sorted in ascending actual usage. AMP is generally accurate, and over-estimates at the distribution tails. The right plot shows the estimate is below 25% over the actual max peak usage in 70% of the jobs. In 3% of the jobs, AMP under-estimated. In another 3%, the estimated usage exceeds the GPU memory capacity. In this case, user configurations are used as fallback. On the evaluated jobs, AMP can reduce user-configured number of GPUs by 45%, and save 1 million GBs of HBM.

10 DISCUSSION

In this section, we discuss the generalizability and limitations of our findings from §5-8, and threats to validity of this paper.

Our findings do not assume any characteristics of specific DLRMs. Embedding tables and other model components in a DLRM are essentially dense tensors, similar to what other major, large-scale DNN are composed of (§2.1). In DLRM, lookup frequency and pooling factor of the embedding tables represent the communication and computation costs of the dense tensors that are model-parallelized. Many of our findings should generalize to other DNNs beyond DLRMs.

Findings 1-3 study GPU memory inefficiency in production scale, which we believe is true for other DNNs [19, 29, 30, 32]. Findings 4-8 study memory usage governed by training components (training loop, broadcasting, optimization, data parallelism) whose implementations should generalize beyond specific model architectures [64, 69, 72]. Since our study is based on PyTorch models and tooling, these results could be specific to PyTorch and NCCL [54, 59, 62, 63, 73].

Findings 9-10 could be limited by the underlying model parallelisms. DLRM mostly uses intra-layer parallelisms [46], which shard each operator in the model into chunks [76, 81]. Orthogonal to intra-layer, inter-layer parallelisms (e.g., pipeline parallelism) shard the model into groups of operators [27, 48]. Some techniques use both [50, 99]. Under inter-layer parallelisms, memory usage could still differ across GPUs due to sharding, but the training-time usage may vary across GPUs, because each GPU holds a different group of operators and the computation cost may vary across GPUs.

Findings 11-12 study the effect of computation cost balance, batch size, and number of trainers in large-scale distributed training. We believe their relations to GPU memory usage are generalizable. Finding 13 generalizes beyond DLRM from our experience, though concrete memory usage shape would differ in different DNN architectures.

11 OTHER RELATED WORK

Prior studies on DNN training cluster focus on workload characteristics, cluster throughput and training failures [1, 6, 25, 30]. Our work complements them with in-depth characterization of DNN training memory usage (§5-6).

DNN memory management techniques aim to reduce model memory footprint (the amount of memory a model needs) to improve training throughput [16, 37, 76, 78, 90]. They offer insights on DNN memory usage with theoretical reasoning and static analysis. We aim to reduce the amount of memory a job requests to improve GPU memory efficiency, with a more fine-grained analysis at production scale.

Memory sharing for concurrent training [38, 43, 56, 91, 95] leverages the cyclic memory usage pattern of DNN to enable a GPU's memory shared temporally (time slicing) or spatially by multiple training jobs on single-device or data-parallel scenarios. We study memory usage of large-scale production DNN training jobs under mixed parallelisms. Our results show distinct characteristics of model-parallel jobs, and potentials for future work in memory sharing (§6, §7.3).

DNN job scheduling and balancing techniques assume requested amount of GPU memory of a training job to be fixed or predefined, and focus on maximizing training cluster throughput [2, 3, 6, 17, 18, 18, 20, 28, 31, 35, 45, 49, 60, 91, 94]. Our work focuses on proactively reducing the requested amount of GPU memory of a job. We analyze GPU memory efficiency of production jobs under mixed parallelisms to shed light on future work along distributed training at scale.

12 CONCLUSION

As the size of deep learning models grows at terabyte scale, large-scale distributed training is becoming a norm. This paper presents a systematic analysis of GPU memory behavior of large-scale distributed training jobs in production at Meta. We measure GPU memory inefficiency, characterize GPU memory utilization, and provide fine-grained analysis on GPU memory usage of production jobs. Our study revealed over a dozen findings with concrete implications. We further build on the analysis to develop a practical GPU provisioning system to improve GPU memory efficiency in production. We believe future research can leverage our study to improve GPU memory efficiency in distributed training.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Venkata Vamsikrishna Meduri, for their valuable comments. We thank Xiaodong Wang, Menglu Yu, Taylor Robie, Srinath Mandalapu, Mehmet Yunt, and other Meta colleagues for their helpful discussions and infrastructure support. We thank Gangmuk Lim, Chirag Shetty, and Darko Marinov for discussions on early drafts. Runxiang Cheng is supported in part by NSF grant CCF-1763788. Tianyin Xu is supported in part by NSF grants CNS-2145295 and CNS-1956007.

REFERENCES

[1] Acun, B., Murphy, M., Wang, X., Nie, J., Wu, C.-J., and Hazelwood,

K. Understanding training efficiency of deep learning recommendation models at scale. In *HPCA* (2021).

[2] Albahar, H., Dongare, S., Du, Y., Zhao, N., Paul, A. K., and Butt, A. R. Schedtune: A heterogeneity-aware gpu scheduler for deep learning. In *CCGrid* (2022).

[3] Athlur, S., Saran, N., Sivathanu, M., Ramjee, R., and Kwatra, N. Varuna: scalable, low-cost training of massive deep learning models. In *EuroSys* (2022).

[4] Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M. S., Bohg, J., Bosselut, A., Brunskill, E., et al. On the opportunities and risks of foundation models. *arXiv:2108.07258* (2021).

[5] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *NeurIPS* (2020).

[6] Chen, Z., Quan, W., Wen, M., Fang, J., Yu, J., Zhang, C., and Luo, L. Deep learning research and development platform: Characterizing and scheduling with qos guarantees on gpu clusters. *TPDS* (2019).

[7] Cheng, H.-T., Koc, L., Harmsen, J., Shaked, T., Chandra, T., Aradhye, H., Anderson, G., Corrado, G., Chai, W., Ispir, M., et al. Wide & deep learning for recommender systems. In *DLRS* (2016).

[8] Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *arXiv:2204.02311* (2022).

[9] Cortez, E., Bonde, A., Muzio, A., Russinovich, M., Fontoura, M., and Bianchini, R. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *SOSP* (2017).

[10] Covington, P., Adams, J., and Sargin, E. Deep neural networks for youtube recommendations. In *RecSys* (2016).

[11] Davis, W., and Lawler, R. Nvidia became a $1 trillion company thanks to the AI boom. https://www.theverge.com/2023/5/30/23742123/nvidia-stock-ai-gpu-1-trillion-market-cap-price-value, 2023.

[12] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., et al. Large scale distributed deep networks. *NeurIPS* (2012).

[13] Delimitrou, C., and Kozyrakis, C. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices* (2014).

[14] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv:1810.04805* (2018).

[15] Elkahky, A. M., Song, Y., and He, X. A multi-view deep learning approach for cross domain user modeling in recommendation systems. In *WWW* (2015).

[16] Fang, J., Zhu, Z., Li, S., Su, H., Yu, Y., Zhou, J., and You, Y. Parallel training of pre-trained models via chunk-based dynamic memory management. *TPDS* (2022).

[17] Gao, W., Hu, Q., Ye, Z., Sun, P., Wang, X., Luo, Y., Zhang, T., and Wen, Y. Deep learning workload scheduling in gpu datacenters: Taxonomy, challenges and vision. *arXiv:2205.11913* (2022).

[18] Gao, W., Ye, Z., Sun, P., Wen, Y., and Zhang, T. Chronus: A novel deadline-aware scheduler for deep learning training jobs. In *SoCC* (2021).

[19] Gao, Y., Liu, Y., Zhang, H., Li, Z., Zhu, Y., Lin, H., and Yang, M. Estimating gpu memory consumption of deep learning models. In *ESEC/FSE* (2020).

[20] Gu, J., Chowdhury, M., Shin, K. G., Zhu, Y., Jeon, M., Qian, J., Liu, H. H., and Guo, C. Tiresias: A gpu cluster manager for distributed deep learning. In *NSDI* (2019).

[21] Gupta, U., Wu, C.-J., Wang, X., Naumov, M., Reagen, B., Brooks, D., Cottel, B., Hazelwood, K., Hempstead, M., Jia, B., et al. The architectural implications of facebook's dnn-based personalized recommendation. In *HPCA* (2020).

[22] Habib, R. OpenAI's plans according to Sam Altman. https://website-754fwhahs-humanloopml.vercel.app/blog/open_ai_talk?utm_source=bensbites&utm_medium=newsletter&utm_campaign=openai-s-roadmap/, 2023.

[23] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *CVPR* (2016).

[24] Hsia, S., Gupta, U., Wilkening, M., Wu, C.-J., Wei, G.-Y., and Brooks, D. Cross-stack workload characterization of deep recommendation systems. In *IISWC* (2020).

[25] Hu, Q., Sun, P., Yan, S., Wen, Y., and Zhang, T. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In *SC* (2021).

[26] Huang, S., Dong, L., Wang, W., Hao, Y., Singhal, S., Ma, S., Lv, T., Cui, L., Mohammed, O. K., Liu, Q., et al. Language is not all you need: Aligning perception with language models. *arXiv:2302.14045* (2023).

[27] Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *NeurIPS* (2019).

[28] Hwang, C., Kim, T., Kim, S., Shin, J., and Park, K. Elastic resource sharing for distributed deep learning. In *NSDI* (2021).

[29] Islam, M. J., Nguyen, G., Pan, R., and Rajan, H. A comprehensive study on deep learning bug characteristics. In *ESEC/FSE* (2019).

[30] Jeon, M., Venkataraman, S., Phanishayee, A., Qian, J., Xiao, W., and Yang, F. Analysis of large-scale multi-tenant gpu clusters for dnn training workloads. In *ATC* (2019).

[31] Jia, X., Jiang, L., Wang, A., Xiao, W., Shi, Z., Zhang, J., Li, X., Chen, L., Li, Y., Zheng, Z., et al. Whale: Efficient giant model training over heterogeneous gpus. In *ATC* (2022).

[32] Jiang, Y., Zhu, Y., Lan, C., Yi, B., Cui, Y., and Guo, C. A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters. In *OSDI* (2020).

[33] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *arXiv:1412.6980* (2014).

[34] Krizhevsky, A. One weird trick for parallelizing convolutional neural networks. *arXiv:1404.5997* (2014).

[35] Le, T. N., Sun, X., Chowdhury, M., and Liu, Z. Allox: compute allocation in hybrid clusters. In *EuroSys* (2020).

[36] Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv:2006.15704* (2020).

[37] Li, Y., Phanishayee, A., Murray, D., Tarnawski, J., and Kim, N. S. Harmony: Overcoming the hurdles of gpu memory capacity to train massive dnn models on commodity servers. *arXiv:2202.01306* (2022).

[38] Lim, G., Ahn, J., Xiao, W., Kwon, Y., and Jeon, M. Zico: Efficient gpu memory sharing for concurrent dnn training. In *ATC* (2021).

[39] Liu, H., Liu, S., Wen, C., and Wong, W. E. Tbem: Testing-based gpu-memory consumption estimation for deep learning. *IEEE Access* (2022).

[40] Liu, Q., and Yu, Z. The elasticity and plasticity in semi-containerized co-locating cloud workload: a view from alibaba trace. In *SoCC* (2018).

[41] Lui, M., Yetim, Y., Ozkan, O., Zhao, Z., Tsai, S.-Y., Wu, C.-J., and Hempstead, M. Understanding capacity-driven scale-out neural recommendation inference. In *ISPASS* (2021).

[42] Luo, S., Xu, H., Lu, C., Ye, K., Xu, G., Zhang, L., Ding, Y., He, J., and Xu, C. Characterizing microservice dependency and performance: Alibaba trace analysis. In *SoCC* (2021).

[43] Mahajan, K., Balasubramanian, A., Singhvi, A., Venkataraman, S., Akella, A., Phanishayee, A., and Chawla, S. Themis: Fair and efficient gpu cluster scheduling. In *NSDI* (2020).

[44] Mark Harris. Unified Memory for CUDA Beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/, 2017.
[45] Mohan, J., Phanishayee, A., Kulkarni, J., and Chidambaram, V. Looking beyond gpus for dnn scheduling on multi-tenant clusters. In *OSDI* (2022).
[46] Mudigere, D., Hao, Y., Huang, J., Jia, Z., Tulloch, A., Sridharan, S., Liu, X., Ozdal, M., Nie, J., Park, J., et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *ISCA* (2022).
[47] Mujtaba, H. NVIDIA AI GPU Demand Blows Up, Chip Prices Increase By 40% & Stock Shortages Expected Till December. https://wccftech.com/nvidia-ai-gpu-demand-blows-up-chip-prices-increase-40-percent-stock-shortages-till-december/, 2023.
[48] Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: Generalized pipeline parallelism for dnn training. In *SOSP* (2019).
[49] Narayanan, D., Santhanam, K., Kazhamiaka, F., Phanishayee, A., and Zaharia, M. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *OSDI* (2020).
[50] Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *SC* (2021).
[51] Naumov, M., Kim, J., Mudigere, D., Sridharan, S., Wang, X., Zhao, W., Yilmaz, S., Kim, C., Yuen, H., Ozdal, M., et al. Deep learning training in facebook data centers: Design of scale-up and scale-out systems. *arXiv:2003.09518* (2020).
[52] Naumov, M., Mudigere, D., Shi, H.-J. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C.-J., Azzolini, A. G., et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv:1906.00091* (2019).
[53] NVIDIA. NVIDIA V100 Tensor Core GPU Datasheet. https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf/, 2018.
[54] NVIDIA. NVIDIA Collective Communication Library (NCCL). https://developer.nvidia.com/nccl, 2019.
[55] NVIDIA. NVIDIA A100 Tensor Core GPU Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf/, 2020.
[56] NVIDIA. CUDA Multi-Process Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2022.
[57] NVIDIA. NVIDIA GH200 Grace Hopper Superchip. https://resources.nvidia.com/en-us-grace-cpu/grace-hopper-superchip, 2023.
[58] OpenAI. GPT-4 Technical Report. https://cdn.openai.com/papers/gpt-4.pdf, 2023.
[59] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *NeurIPS* (2019).
[60] Peng, Y., Bao, Y., Chen, Y., Wu, C., and Guo, C. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *EuroSys* (2018).
[61] Autograd mechanics. https://pytorch.org/docs/stable/notes/autograd.html, 2023.
[62] Automatic differentiation package. https://pytorch.org/docs/stable/autograd.html, 2023.
[63] CUDA semantics. https://pytorch.org/docs/stable/notes/cuda.html, 2023.
[64] DistributedDataParallel. https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html, 2023.
[65] Shape propagation. https://github.com/pytorch/pytorch/blob/master/torch/fx/passes/shape_prop.py, 2023.
[66] StorageImpl. https://github.com/pytorch/pytorch/blob/master/c10/core/StorageImpl.h, 2023.
[67] torch.cuda.empty_cache. https://pytorch.org/docs/stable/generated/torch.cuda.empty_cache.html, 2023.
[68] torch.cuda.list_gpu_processes. https://pytorch.org/docs/stable/generated/torch.cuda.list_gpu_processes.html, 2023.
[69] torch.distributed.broadcast_object_list. https://pytorch.org/docs/stable/distributed.html#torch.distributed.broadcast_object_list, 2023.
[70] torch.fx. https://pytorch.org/docs/stable/fx.html, 2023.
[71] torch.fx.Node. https://pytorch.org/docs/stable/fx.html#torch.fx.Node, 2023.
[72] torch.optim.Optimizer. https://pytorch.org/docs/stable/optim.html#torch.optim.Optimizer, 2023.
[73] torch.profiler. https://pytorch.org/docs/stable/profiler.html, 2023.
[74] Qian, N. On the momentum term in gradient descent learning algorithms. *Neural Networks* (1999).
[75] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *JMLR* (2020).
[76] Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models. In *SC* (2020).
[77] Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H., and Kozuch, M. A. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC* (2012).
[78] Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A., and Keckler, S. W. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *MICRO* (2016).
[79] Sethi, G., Acun, B., Agarwal, N., Kozyrakis, C., Trippel, C., and Wu, C.-J. Recshard: statistical feature-based memory optimization for industry-scale neural recommendation. In *ASPLOS* (2022).
[80] Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., et al. Mesh-tensorflow: Deep learning for supercomputers. *NeurIPS* (2018).
[81] Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv:1909.08053* (2019).
[82] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556* (2014).
[83] Steck, H., Baltrunas, L., Elahi, E., Liang, D., Raimond, Y., and Basilico, J. Deep learning for recommender systems: A netflix case study. *AI Magazine* (2021).
[84] Sun, Q., Liu, Y., Yang, H., Zhang, R., Dun, M., Li, M., Liu, X., Xiao, W., Li, Y., Luan, Z., et al. Cognn: efficient scheduling for concurrent gnn training on gpus. In *SC* (2022).
[85] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *CVPR* (2016).
[86] DistributedModelParallel. https://pytorch.org/torchrec/torchrec.distributed.html#torchrec.distributed.model_parallel.DistributedModelParallel, 2023.
[87] EmbeddingShardingPlanner. https://pytorch.org/torchrec/torchrec.distributed.planner.html#torchrec.distributed.planner.planners.EmbeddingShardingPlanner, 2023.
[88] TorchRec. https://github.com/pytorch/torchrec, 2023.
[89] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *NeurIPS* (2017).
[90] Wang, L., Ye, J., Zhao, Y., Wu, W., Li, A., Song, S. L., Xu, Z., and Kraska, T. Superneurons: Dynamic gpu memory management for training deep neural networks. In *PPoPP* (2018).
[91] Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., et al. Gandiva:

Introspective cluster scheduling for deep learning. In *OSDI* (2018).

[92] Xu, T., Zhang, J., Huang, P., Zheng, J., Sheng, T., Yuan, D., Zhou, Y., and Pasupathy, S. Do not blame users for misconfigurations. In *SOSP* (2013).

[93] Xu, T., and Zhou, Y. Systems approaches to tackling configuration errors: A survey. *CSUR* (2015).

[94] Yeung, G., Borowiec, D., Yang, R., Friday, A., Harper, R., and Garraghan, P. Horus: Interference-aware and prediction-based scheduling in deep learning systems. *TPDS* (2021).

[95] Yu, P., and Chowdhury, M. Fine-grained gpu sharing primitives for deep learning applications. *NeurIPS* (2020).

[96] Zha, D., Feng, L., Tan, Q., Liu, Z., Lai, K.-H., Bhushanam, B., Tian, Y., Kejariwal, A., and Hu, X. Dreamshard: Generalizable embedding table placement for recommender systems. *arXiv:2210.02023* (2022).

[97] Zhang, B., Luo, L., Liu, X., Li, J., Chen, Z., Zhang, W., Wei, X., Hao, Y., Tsang, M., Wang, W., et al. Dhen: A deep and hierarchical ensemble network for large-scale click-through rate prediction. *arXiv:2203.11014* (2022).

[98] Zhao, W., Xie, D., Jia, R., Qian, Y., Ding, R., Sun, M., and Li, P. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. *MLSys* (2020).

[99] Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *OSDI* (2022).

LithOS: An Operating System for Efficient Machine Learning on GPUs

Patrick H. Coppock, Brian Zhang, Eliot H. Solomon, Vasilis Kypriotis, Leon Yang[†], Bikash Sharma[†], Dan Schatzberg[†], Todd C. Mowry, and Dimitrios Skarlatos
Carnegie Mellon University [†]Meta

Abstract

The rapid growth of machine learning (ML) has made GPUs indispensable in datacenters and underscores the urgency of improving their efficiency. However, balancing diverse model demands with high utilization remains a fundamental challenge. Transparent, fine-grained GPU resource management that maximizes utilization, energy efficiency, and isolation requires an OS approach. This paper introduces *LithOS*, a first step towards a GPU OS.

LithOS includes the following new abstractions and mechanisms for efficient GPU management: (i) a novel *TPC Scheduler* that supports spatial scheduling at the granularity of individual TPCs, unlocking efficient TPC stealing between workloads; (ii) a transparent *kernel atomizer* to reduce head-of-line blocking and allow dynamic resource reallocation mid-execution; (iii) a lightweight *hardware right-sizing* mechanism that dynamically determines the minimal TPC resources needed per atom; and (iv) a transparent *power management* mechanism that reduces power consumption based upon in-flight work characteristics.

We build LithOS in Rust and evaluate its performance across a broad set of deep learning environments, comparing it to state-of-the-art solutions from NVIDIA and prior research. For inference stacking, LithOS reduces tail latencies by 13× compared to MPS; compared to the best-performing SotA, it reduces tail latencies by 4× while improving aggregate goodput by 1.3×. Furthermore, in hybrid inference-training stacking, LithOS reduces tail latencies by 4.7× compared to MPS; compared to the best-performing SotA, it reduces tail latencies by 1.18× while improving aggregate throughput by 1.35×. Finally, for a modest performance hit under 4%, LithOS's hardware right-sizing provides a quarter of GPU capacity savings on average, while for a 7% hit, LithOS's transparent power management delivers a quarter of GPU total energy savings on average. Overall, LithOS transparently increases GPU efficiency, establishing a foundation for future OS research on GPUs.

SOSP '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1870-0/2025/10
https://doi.org/10.1145/3731569.3764818

ACM Reference Format:
Patrick H. Coppock, Brian Zhang, Eliot H. Solomon, Vasilis Kypriotis, Leon Yang, Bikash Sharma, Dan Schatzberg, Todd C. Mowry, and Dimitrios Skarlatos. 2025. LithOS: An Operating System for Efficient Machine Learning on GPUs. In *ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3731569.3764818

1 Introduction

The rise of ML workloads has driven massive GPU deployments in datacenters. Yet, despite concerns over power and supply constraints, utilization remains low—public reports cite just 52% at Microsoft [30] and 10% at Alibaba [65]. Our analysis of production services at Meta also reveals that utilization can be low. In inference services, utilization can be below 30% depending on model and service characteristics. For training, Llama 3 achieves a GPU utilization of around 40% [25]. Figure 1 shows normalized GPU utilization metrics over a period of a week. Given the high monetary cost and rising power demands—now exceeding 1,000 W per GPU [36, 41]—this is unsustainable.

It is challenging to achieve high utilization without GPU sharing. While dedicating a GPU to a single workload leads to high performance, individual workloads often fail to keep the GPU fully utilized: GPU cores idle on communication stalls, low batch sizes result in insufficient parallelism, dynamic request loads lead to overprovisioning, and so on [24, 27, 65]. As GPUs become more powerful with increasing Streaming Multiprocessor (SM) counts and memory bandwidth [13, 41], achieving high utilization will become more challenging.

One potential approach to GPU sharing is collocating *latency-critical* (LC) tasks for which performance is of utmost importance with *best-effort* (BE) tasks that lack hard deadlines. However, existing systems do not offer a practical solution for prioritizing LC tasks over BE tasks when they contend for resources. Many approaches lack transparency, rendering them incompatible with large parts of the ML software stack [2, 18, 19, 26, 27, 29, 39, 42, 45, 51, 54, 59]. For instance, some are tied to specific versions of frameworks like PyTorch or TVM that are no longer maintained [2, 19, 26, 59, 65]. Other solutions like TGS [64] or Clockwork [26] fall short of achieving high GPU utilization due to limited temporal scheduling that cannot execute multiple models in parallel. Spatial scheduling solutions, including NVIDIA's MPS [12]

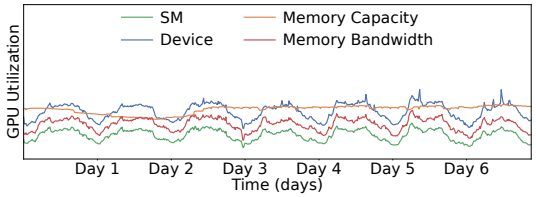


Figure 1. *GPU utilization metrics over a week in production inference services at Meta.*

and MIG [15] or research efforts like REEF [27], Orion [59], and others [19], enable parallel execution of multiple applications. However, they are too coarse-grained, scheduling entire inference requests, training batches, or DNN operators, resulting in low utilization and head-of-line (HoL) blocking [2, 9, 19, 21, 27, 37, 39, 42, 51, 59, 60, 64, 66]. Efficient multitenant scheduling on GPUs has remained elusive.

Beyond collocation mechanisms, datacenter GPU management must move past static provisioning to address inefficiencies without sacrificing performance or transparency. Current systems overlook the changing characteristics of deep learning workloads—such as fluctuating compute intensity and parallelism across models and execution phases—leaving GPUs underutilized even as they consume significant power. Bridging this gap requires approaches that adapt resource allocation and power consumption to the fine-grained behavior of ML workloads.

This utilization crisis is in stark contrast with the situation for CPUs, where time-sharing operating systems allocate tasks to cores via inexpensive context switches, providing isolation, resource allocation, power management, and transparency. The extreme data-parallel nature of GPUs imposes different trade-offs than do CPUs, but also exposes the limitations of current abstractions built around compilers, frameworks, and drivers. To transparently improve utilization and efficiency, we believe that GPUs must evolve toward an operating system model—one that brings first-class support for control, isolation, and resource management.

1.1 Our Approach: An Operating System for GPUs

To address datacenter GPU efficiency challenges, we introduce LithOS, which brings an efficient operating system approach to deep learning on GPUs. LithOS is fully transparent to the ML stack, allowing seamless integration without requiring modifications to models, runtimes, or frameworks. LithOS moves the bulk of GPU scheduling from proprietary drivers and hardware into software, allowing, for the first time, fine-grained temporal and spatial scheduling of ML workloads. LithOS operates at the granularity of individual kernel thread blocks that are dynamically mapped onto the GPU's Texture Processing Clusters (TPCs). To achieve this, LithOS introduces novel abstractions and mechanisms that decouple kernel work submission from thread block execution on GPUs, enabling intelligent scheduling decisions, resource allocation, and power management.

First, LithOS introduces a novel fine-grained *TPC Scheduler* that asynchronously determines the compute unit allocation and submission time for each piece of work. It enables precise control at the granularity of individual TPCs, providing strong isolation between workloads. The scheduler is guided toward efficient scheduling decisions by an online kernel latency predictor and incorporates a technique called *TPC Stealing* to improve GPU utilization.

To address the absence of hardware preemption, LithOS introduces *kernel atomization*, which transparently partitions kernels into schedulable *atoms*—subsets of thread blocks—without compiler, runtime, source, or PTX changes. Atomization reduces head-of-line blocking, mitigates interference, and allows TPC reconfiguration mid-execution, providing flexibility that is unavailable for monolithic kernels. Building on this foundation, LithOS introduces a dynamic *hardware right-sizing* mechanism that uses lightweight modeling to determine the minimal TPC resources required for each kernel and its atoms, saving significant capacity. Finally, LithOS presents a fine-grained *power management* mechanism that adjusts the GPU's frequency in response to the characteristics of in-flight work, saving substantial energy.

We implement LithOS in Rust and evaluate its performance across a broad set of deep learning environments, comparing it to state-of-the-art solutions from NVIDIA and prior research. For inference stacking, LithOS reduces tail latencies by 13× compared to MPS; compared to the best-performing SotA, it reduces tail latencies by 4× while improving aggregate goodput by 1.3×. Furthermore, in hybrid inference-training stacking, LithOS reduces tail latencies by 4.7× compared to MPS; compared to the best-performing SotA, it reduces tail latencies by 1.18× while improving aggregate throughput by 1.35×. Finally, for a modest performance hit under 4%, LithOS's hardware right-sizing provides a quarter of GPU capacity savings on average, while for a 7% hit, LithOS's transparent power management delivers a quarter of a GPU total energy savings on average. Overall, LithOS transparently increases GPU efficiency, establishing a foundation for future OS research on GPUs.

This paper makes the following contributions:

- A comprehensive study of inference services at Meta, highlighting the behavior of production ML models and the challenges of GPU underutilization.
- A fine-grained spatial *TPC Scheduler* that dynamically allocates TPCs using *TPC Stealing* to boost utilization.
- A transparent *Kernel Atomizer* that independently schedules sets of kernel thread blocks, unlocking efficiency.
- A dynamic *hardware right-sizing* mechanism that optimizes TPC allocations for significant capacity savings.
- A transparent *power management* mechanism that adjusts frequency based on kernel scaling to save energy.
- The design of LithOS, a step towards an OS for GPUs.
- The evaluation of LithOS across ML environments.

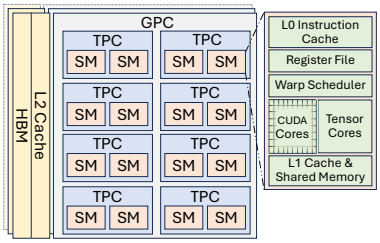


Figure 2. *GPU Architecture.*

2 Background and Related Work

In this section, we first present a brief background on NVIDIA GPU architectures and then cover related work.

2.1 A Brief Background on GPUs

GPU Architecture. Figure 2 depicts a typical GPU architecture using NVIDIA's terminology. Each GPU consists of several Graphics Processing Clusters (GPCs). Each GPC is a collection of multiple Texture Processing Clusters (TPCs), and each TPC includes a small number of Streaming Multiprocessors (SMs). Each SM is composed of tens of cores. For example, NVIDIA's H100 [13] includes 8 GPCs, 9 TPCs per GPC, 2 SMs per TPC, and 128 cores per SM.

GPU Programming. GPU applications are composed of kernels that execute specific operators (e.g., convolution). A kernel defines its resources—thread blocks, threads, registers, and shared memory—at launch time. Programmers divide a kernel's work among the thread blocks. Each thread block executes on an SM and consists of multiple SIMD threads.

GPU Streams. CUDA streams enable concurrent execution of independent tasks, similar to CPU threads. Stream work is executed in FIFO order. Some CUDA calls are asynchronous, while others wait for all previous tasks to finish.

2.2 Related Work

Cooperative multitenancy. Cooperative scheduling involves tenants coordinating to share resources, typically at the ML framework level, with all models running in the same process [2, 18, 19, 26, 27, 29, 39, 42, 45, 51, 54, 59]. These approaches require custom ML frameworks and are hence limited by their inability to support arbitrary applications. Some also rely on extensive offline profiling [27, 59] or kernel source modifications [27, 42], which are impractical at scale. Finally, any non-cooperating tenant invalidates guarantees made by the runtime, making adoption difficult in practice.

Transparent multitenancy. Transparent GPU sharing supports unmodified applications through native mechanisms such as time slicing, MPS [12], and MIG [15], or their combinations [44, 61]. By contrast, most prior software solutions require application or framework changes. TGS [64] is one exception, enabling transparent sharing across containers. In practice, however, uncooperative tasks and limited application-specific knowledge make transparent multitasking especially challenging.

Temporal multitenancy. Temporal multitenancy dedicates the entire GPU to a single task at a time via native time slicing or software scheduling. Some approaches work at the level of entire inference requests (e.g., Clipper [18], Nexus [54], TensorFlow-Serving [45], Clockwork [26], and INFaaS [51]), while others schedule kernels (e.g., PipeSwitch [2], AntMan [65], Gemini [7], KubeShare [67], and TGS [64]). *Time slicing* is NVIDIA's default temporal multitenancy solution. It shares the GPU in a round-robin fashion, giving each task exclusive access for several milliseconds. These methods execute only one job at a time, leading to low utilization.

Spatial multitenancy. Spatial multitenancy typically builds on MIG or MPS to enable multiple applications to run concurrently on a GPU and improve utilization. *MPS* multiplexes multiple GPU contexts onto one, allowing multiple tasks to use the GPU concurrently. This can yield greater throughput but leads to performance interference. *MIG* partitions the GPU's compute and memory resources along GPC boundaries, providing strong hardware isolation. However, the coarse granularity of its partitioning and steep reconfiguration overheads (>5s [63]) can leave resources idle. These problems exist even at datacenter scale. As shown in our study, hardware-based multitenancy is workable for Meta's production use cases. However, the fluctuations in Figure 1 also exist at finer granularities, necessitating overprovisioning and leaving capacity on the table. Dynamic reconfiguration is currently too slow to be a viable remedy for this.

Like temporal systems, existing spatial sharing systems are coarse-grained, operating at the level of inference requests or kernels. Their goal is to protect latency-critical (LC) applications by restricting kernels launched by other jobs [19, 59] or limiting GPU resources allocated to best-effort tasks, as seen in systems like REEF [27], MuxFlow [39], PTask [52], and others [9, 21, 34, 37, 42, 60, 66]. However, the coarseness of these approaches limits control over GPU resources, often leading to HoL blocking, low utilization, and interference. Figure 3 highlights the challenges of spatial sharing. In Figure 3(a), a single workload runs on the GPU, issuing two requests with five total kernels. This results in fast kernel completion for *A* and *B* but leaves much of the GPU underutilized. When MPS enables concurrent execution of multiple tasks in Figure 3(b), utilization is improved, but the original task's requests face significant delays. Overall, prior works have tackled some multitenant ML scheduling challenges but fail to offer a complete, transparent solution. Importantly, prior temporal and spatial strategies operate at a coarse granularity, limit utilization, and cause HoL blocking, which interferes with collocated workloads.

Right-sizing. Prior efforts have explored GPU job right-sizing to improve resource efficiency. However, these approaches often rely on hardware modifications [10, 72], lack transparency to application software [8, 9, 21, 35, 71], and depend on offline profiling [9–11, 21, 35, 71]. Crucially, most existing solutions operate at the granularity of entire jobs,

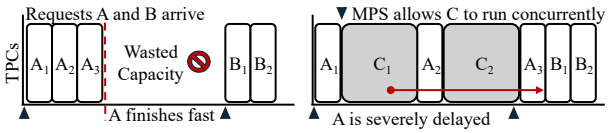


Figure 3. *GPU timeline showcasing the pitfalls of MPS.*

which limits their ability to fully exploit the benefits of fine-grained right-sizing and can lead to suboptimal performance.

Dynamic Voltage Frequency Scaling. Recent efforts [31, 47, 48, 58, 69] have applied dynamic voltage frequency scaling (DVFS) to minimize the power consumption of GPUs with a particular focus on LLM inference clusters [31, 58]. Such approaches are based on extensive offline profiling across several input lengths and train dedicated output length predictors, failing to provide a transparent mechanism. Prior work on DVFS operates at a coarser granularity, observing the performance of the whole inference request and missing finer optimization opportunities.

3 Motivation

In this section, we showcase a detailed study of production GPU infrastructure challenges and opportunities.

3.1 Understanding GPU Utilization in Datacenters

To understand GPU utilization in datacenters, we analyze a subset of inference services at Meta, which serve deep learning models across its fleet. At Meta, inference services rely in part on NVIDIA H100 GPU nodes. Each node has 8 GPUs, which can be further partitioned via software and hardware-based multitenancy into different container shapes. The production service performs offline analysis of each model, assigning models to hardware partitions for deployment. The goal is to meet tight SLAs on tail response times for each model. In Figure 1, we show normalized GPU compute and memory utilization over a week. Device utilization in production services can range between under 25% and higher than 60%. As expected, SM utilization is lower than device utilization, with lows of under 15%. In terms of memory, there is bandwidth room for multitenancy, with a fifth of the bandwidth being utilized on the lower end. Memory capacity utilization behaves similarly, leaving room for multitenancy, with utilization being steady as models are kept loaded in GPU memory to meet tight SLAs. These SLAs also enforce small batch sizes, preventing full GPU resource saturation even at high request loads. Finally, the characteristics of individual models can lead to low utilization of GPU resources. For example, memory-intensive models can lead to SM utilization plummeting. As a result, multi-model stacking can enable high utilization.

Inference Traffic. To investigate low GPU utilization, we first examine inference traffic. Figure 4 shows the mean-normalized requests per second (RPS) over a week, revealing a diurnal pattern. RPS can scale by 2.2× between minimum and maximum traffic, closely correlating with the GPU utilization trends shown in Figure 1. Next, we analyze model request frequencies. We sample thirteen of the most commonly used models and plot in Figure 5 the normalized frequency of inference requests over the same week. The distribution's variance is significant, with the most popular model *A* receiving several hundred times more requests than the least popular model *M*. Over-provisioning GPUs for such a wide request distribution can lead to underutilization, particularly for less popular models.

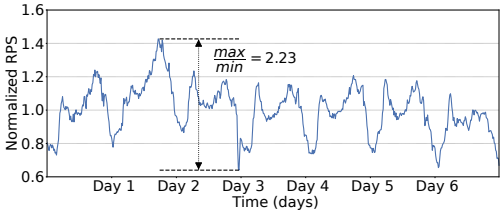


Figure 4. *Mean normalized traffic.*

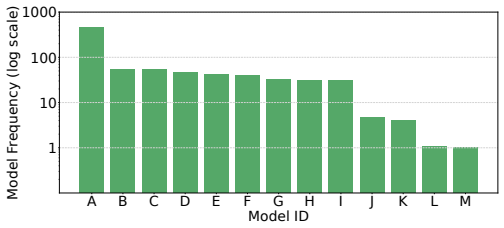


Figure 5. *Model frequency distribution.*

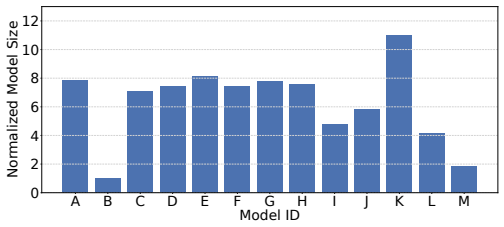


Figure 6. *Model size distribution.*

Model Sizes. To better understand GPU utilization, we examine the sizes of the most commonly used models based on weights, parameters, and embeddings. As shown in Figure 6, model sizes vary significantly, with a more than a 10× difference between the largest and smallest models. Half are relatively large, while the rest are smaller. Both large and small models are frequently used: for example, the smallest model *B* has usage comparable to larger models *E* and *G*. This highlights the opportunity to collocate models of different sizes while meeting each of their service-level agreements.

GPU Sharing Limitations and Takeaways. Despite the urgent need to improve GPU utilization, datacenters often rely on limited GPU sharing or hardware approaches like MIG due to requirements for compatibility and transparency within the ML software stack. Non-transparent solutions that require framework or application changes for multitenancy are impractical at scale, given the complexity of maintaining

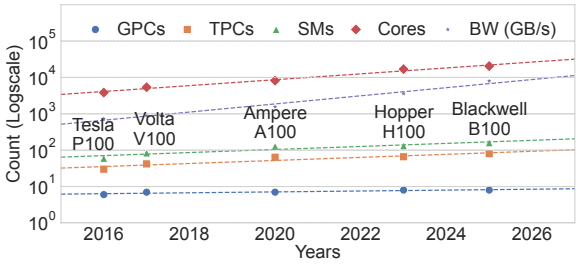


Figure 7. *NVIDIA GPU trends over a decade.*

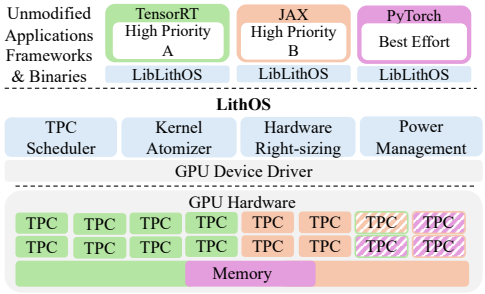


Figure 8. *LithOS architecture overview.*

multiple ML frameworks, runtimes, and compilers. Importantly, in the rapidly evolving ML space, transparent solutions help avoid the risk of locking infrastructure into rigid, outdated designs. Based on these insights, we design LithOS as a fully transparent OS for efficient ML multitenancy.

4 Abstractions, Interfaces, and Principles for a GPU Operating System

LithOS is built on a set of abstractions, interfaces, and principles that define how a GPU operating system should manage resources. These abstractions identify the right granularity of control, the interfaces expose flexible yet predictable knobs, and the principles guide how LithOS balances efficiency, fairness, and robustness under multitenancy.

4.1 Resources and Isolation

Scheduling Granularity. GPU cores and memory bandwidth have grown by orders of magnitude, yet GPC counts have remained nearly flat (six in P100 to eight in B100) as shown in Figure 7. With multi-die designs [22], coarse GPC-level partitioning (e.g., MIG) will waste even more resources. Conversely, intra-SM control is best handled by hardware and compilers; OS intervention would break transparency and portability. LithOS instead adopts the TPC as its scheduling abstraction. TPCs provide finer-grained control than GPCs while remaining transparent to application-level optimizations. Although current APIs do not expose TPC/SM control, LithOS leverages reverse-engineering to manage them, and we argue that native support is feasible for future hardware. *Principle: Manage resources at the finest granularity where the OS is effective while preserving transparency.*

Resource Allocation. The one-application-per-GPU model ignores that kernels scale differently: some saturate with few resources while others benefit from many. LithOS allocates TPCs to kernels based on runtime scaling behavior. Its interface allows applications or administrators to specify tolerable performance loss, enabling right-sizing while maintaining predictability. *Principle: Expose simple performance-tolerance knobs while hiding hardware complexity.*

Power Management. Today's GPUs enforce device-wide DVFS, assuming a single workload. In multitenant settings, this is inefficient: memory-bound kernels saturate bandwidth early, while compute-bound kernels benefit from high frequencies. LithOS virtualizes frequency, letting each workload appear to run at its preferred setting while the OS adjusts DVFS policies for energy efficiency. The same tolerance interface enables transparent performance/power trade-offs, and future per-SM DVFS would further amplify these gains. *Principle: Virtualize frequency to preserve the illusion of dedicated control while optimizing system-wide power gains.*

Security and Fault Isolation. NVIDIA solutions lie at extremes: MIG offers strong but coarse GPC isolation; MPS offers flexible sharing with no protection. LithOS adopts TPC-level isolation, combining flexibility with protection. Each application executes in its own address space with hardware-enforced memory isolation. For faults, LithOS interposes on common GPU errors [39], terminating only the faulty process. LithOS reinitializes the driver in case of unrecoverable errors. LithOS targets multi-tenant production environments with shared GPU infrastructure. *Principle: Enforce isolation at the finest practical granularity, and ensure local faults degrade gracefully.*

4.2 Closing the Gap

These abstractions define LithOS's philosophy: virtualize resources at the right granularity, expose predictable interfaces, and ensure robustness under multitenancy. A key challenge is bridging the gap between CPU and GPU OS design. LithOS leverages OS principles to reimagine GPUs, transforming them from single-model devices into fully virtualized multitenant platforms. LithOS provides proven CPU OS principles to GPU realities, transparently unifying GPU management. This enables higher utilization, strong guarantees, and a foundation for future GPU OS research.

5 LithOS Design

We propose LithOS, an OS designed to address GPU inefficiencies in datacenters. LithOS operates transparently across the ML stack, enabling efficient machine learning on GPUs.

5.1 Architecture Overview

Figure 8 presents the architecture of LithOS. It runs on CPU cores and interposes at the driver level, providing a dynamically linked library, LibLithOS, that mimics the native CUDA library. As a GPU operating system, LithOS maintains a

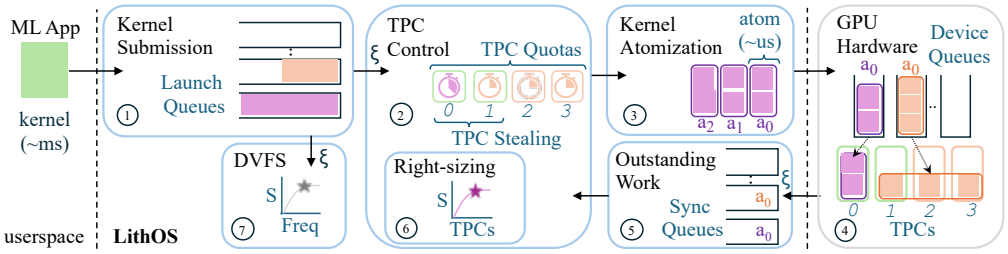


Figure 9. *LithOS operations overview.*

system-wide view of GPU state across applications with varying priorities, enabling efficient scheduling and management. Applications follow the CUDA programming model and submit kernels to LithOS, which decouples submission from GPU execution. This transparently shifts scheduling control from the driver and hardware to the LithOS layer. The *TPC Scheduler* manages resources at the granularity of individual TPCs, unlocking *TPC Stealing* opportunities. Idle TPCs are lent to other tasks, improving utilization.

LithOS also introduces the *Kernel Atomizer*, which—without access to application source or PTX code—transparently breaks kernels into smaller thread block chunks called *atoms*. This enables finer-grained GPU scheduling and reduces head-of-line (HoL) blocking. Building on fine-grained control, LithOS supports dynamic hardware right-sizing, using lightweight models to reduce TPC allocations for individual kernels and atoms, yielding substantial capacity savings. Finally, LithOS applies transparent fine-grained DVFS, adjusting GPU frequency based on in-flight work to save energy. Together, these mechanisms enable intelligent scheduling policies that maximize GPU utilization and efficiency across diverse ML workloads. The rest of this section details how these mechanisms operate and interact, referencing Figure 9.

5.2 Interface with Userspace

Kernel Submission. Applications interact with LithOS via *launch queues* that buffer work (Figure 9, Step ①), giving LithOS full control over when work is dispatched to the GPU. This is important because, once submitted, a kernel's priority or resources cannot be changed, nor can it be rescheduled. Eagerly dispatching work can lead to sub-optimal scheduling. LithOS therefore defers dispatch to minimize outstanding work on the GPU. A launch queue is created when an application creates a stream via `cuStreamCreate`. On asynchronous CUDA calls like `cuLaunchKernel`, LithOS enqueues the kernel and returns control to the application.

Compute Quotas. LithOS lets users and system administrators enforce GPU limits through TPC quotas (Figure 9, Step ②), guaranteeing each application a specified number of TPCs when runnable work exists. TPCs are managed analogously to CPU cores, enabling fine-grained GPU control. A lightweight TPC scheduler then coordinates launch queues and quotas to maximize utilization and efficiency.

5.3 TPC Scheduler

LithOS introduces a novel scheduler that operates at the granularity of individual TPCs, offering several advantages. TPC-level control enables fine-grained GPU resource management. Unlike static partitioning schemes like MIG, LithOS supports dynamic, on-the-fly TPC allocation, allowing kernels to run on different TPCs without reconfiguration overhead. This flexibility maximizes utilization without coarse partitioning or slow reallocation. Kernels are scheduled on assigned TPCs, ensuring guaranteed resources for high-priority applications. However, as shown in Section 3, fixed allocations often leave TPCs idle due to traffic patterns or model variability. To address this, LithOS employs dynamic scheduling and TPC Stealing to reassign idle resources. We believe that TPC scheduling lays the foundation for evolving GPU policies, much as CPU scheduling matured over time [23, 46].

Operation. At a high level, the TPC Scheduler uses dispatcher threads to monitor launch queues (Figure 9, Step ①) and submit work to the GPU. A key goal is to keep the GPU busy while maintaining scheduling flexibility. The scheduler faces two main challenges: varying kernel durations and balancing flexibility with GPU starvation. To address the former, it applies Kernel Atomization (Figure 9, Step ③, Section 5.4) to split long-running kernels into smaller thread block chunks called *atoms*. To address the latter, it tracks outstanding work via sync queues (Figure 9, Step ⑤), throttling submissions until the backlog drops below a tunable threshold. We use a 100μs limit, sufficient to cover host-device communication latency. A dedicated Tracker thread monitors task completion and updates the scheduler state.

TPC Stealing. To improve work conservation, the scheduler dynamically reassigns underutilized TPCs across applications. In Figure 10(a), static allocation leads to idle TPCs. In Figure 10(b), stealing allows A_1 to borrow TPCs from an idle workload, reducing waste. However, this may cause head-of-line (HoL) blocking from priority inversion if a new request B is delayed by C_2 occupying the stolen TPCs. To mitigate this, the scheduler adopts a layered strategy. It maintains per-TPC timers informed by a latency prediction module, estimating kernel (and atom) durations at submission time. These timers help avoid stealing from long-running TPCs. As tasks complete, sync queues are cleared and timers updated, potentially refining predictions (Section 5.7). LithOS also

limits outstanding atoms and uses lower hardware stream priorities for work on stolen TPCs. Combined with kernel atomization, these mechanisms boost utilization while minimizing interference.

5.4 Kernel Atomizer

At the core of LithOS lies the *Kernel Atomizer*. The Kernel Atomizer transforms kernels into small chunks called *atoms*, each containing a subset of the grid's thread blocks (Figure 9, Step ③). Importantly, the Kernel Atomizer operates without any access to source or PTX code, making it fully transparent to the entire ML software stack. This allows LithOS to dispatch work at thread-block rather than kernel granularity. This is a critical requirement for an OS targeting GPUs, as the execution time of kernels can vary wildly from a few microseconds to tens of milliseconds.

Impact of Kernel Scheduling on Latency. To illustrate the need for kernel atomization, Figure 11 presents P_{99} kernel latencies across various training and inference workloads. Figure 11(a) shows how P_{99} latency increases with larger training batch sizes. Since the typical batch size for each model varies, we normalize by plotting memory usage at each size. Most models quickly produce long-running kernels lasting several milliseconds; DLRM [40] stands out with kernel latencies exceeding 30 ms. While training workloads are the major culprit, in Figure 11(b) we see that large language model (LLM) inference based on a trace from Microsoft Azure [58] containing small (S), medium (M), and large (L) prompt lengths can also produce several-millisecond-long kernels for large prompts. Given that models can have very tight SLO constraints (in the low tens of milliseconds), we guide the design of LithOS toward a finer-grained scheduling unit that mitigates head-of-line blocking effects.

Operation. When a long-running kernel is about to be scheduled, LithOS predicts the duration of the kernel given its TPC assignment using the predictor module (detailed in Section 5.7). LithOS then computes the number of atoms into which to split the kernel by dividing the predicted kernel duration by a tunable parameter called the `atom_duration`. If this parameter is set too low, an atomized kernel may actually take longer to complete. Limits of 250-500 μs are effective. Crucially, LithOS is able to transparently chunk kernels into atoms at runtime. Atoms are then submitted to the GPU and can be scheduled on the TPCs dictated by the TPC Scheduler (Figure 9, ④). As a result, LithOS resolves a major challenge faced by prior works that operate higher in the stack: the Kernel Atomizer works on applications written in any framework, that use any libraries (including closed-source ones like cuDNN), and are built with any compiler.

To understand the benefits of scheduling at atom granularity, we return to Figure 10(b). Stealing improves the schedule but does not eliminate HoL blocking and wasted capacity. By dividing the kernels into atoms, work can be packed more tightly, as in Figure 10(c), and TPC allocations can be dynamically adjusted throughout a kernel's execution. Now, B_1 is no longer blocked by C_2, as stealing is disabled for the latter's subsequent atoms $\hat{C}_2$ once request B is submitted.

Algorithm 1 Prelude Kernel Pseudocode.

```
kernel fn prelude(*args):
    let atom : *const AtomMetadata = AtomMetadataAddr as _
    let block_idx = blockIdx.z * gridDim.y * gridDim.x
                  + blockIdx.y * gridDim.x
                  + blockIdx.x
    if atom->block_idx_lo <= block_idx < atom->block_idx_hi:
        atom->kernel_entrypoint(*args)
```

To demonstrate how LithOS's Kernel Atomizer operates, we consider a `Conv` kernel with a grid dimension of {8,8,1}, resulting in 64 blocks with `block_idx` ranging from 0 to 63. Instead of launching the `Conv` kernel directly, LithOS invokes a `Prelude` kernel, which calls into the original kernel using the same launch configuration. The prelude kernel is shown in Algorithm 1. At a high level, it checks whether `block_idx` falls within a specified range—calling `Conv` if so, or exiting early otherwise. For example, to partition the grid into 2 atoms, the kernel atomizer launches the prelude twice with block index ranges [0,32) and [32,64). Using this technique, LithOS can divide the kernel into up to 64 atoms. By specifying non-overlapping block ranges, the atomizer ensures each block is executed once, maintaining correctness.

Atomization Considerations. Kernels launch with an explicit set of resources; thus, the kernel atomizer ensures that the `Prelude` kernel uses the same set of resources as the original `Conv` kernel. Furthermore, the `Prelude` kernel needs to know the entry point to the `Conv` kernel. The Kernel Atomizer passes this information to the `Prelude` kernel in an `AtomMetadata` struct as seen in Algorithm 1.

Performance Optimizations. LithOS continuously monitors the effectiveness of the Kernel Atomizer to enhance performance. First, to avoid the overhead introduced by additional code in the `Prelude` kernel for kernels with many short threads, LithOS may disable atomization for such kernels. Additionally, for kernels with a large number of thread blocks, the Kernel Atomizer dynamically adjusts the `atom_duration` parameter to control its aggressiveness. This minimizes the performance penalty due to the increased thread block traffic from early-exiting threads.

5.5 Right-Sizing Hardware Resources

LithOS's ability to schedule at the TPC level unlocks new opportunities for fine-grained GPU right-sizing. Figure 12 highlights this potential by plotting kernel speedups as a function of allocated TPCs for representative workloads (Section 7). The selected kernels collectively account for 99% of total execution time, with color gradients indicating each kernel's relative contribution. For Llama inference, general matrix multiplication (GEMM) and multihead attention kernels exhibit diminishing returns, while the kernel responsible

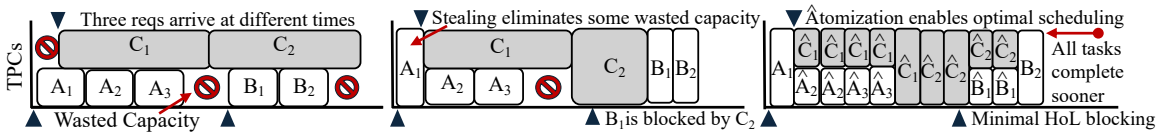


Figure 10. *GPU timeline for two workloads showcasing (a) TPC Scheduling, (B) Stealing, and (C) Atomization.*

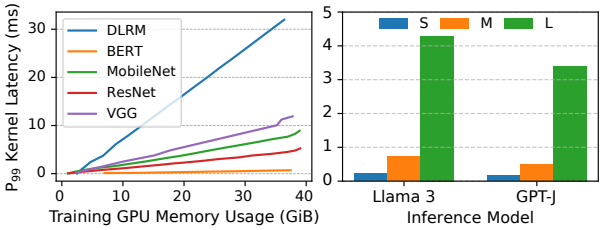


Figure 11. *(a) P_{99} kernel latency at different training batch sizes normalized to memory usage. (b) P_{99} kernel latency for different inference prompt sequence lengths for LLMs.*

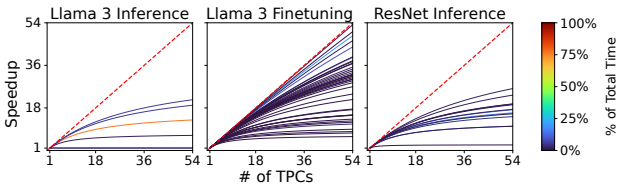


Figure 12. *LithOS's interpolated TPC scaling curves.*

for applying the token frequency penalty does not scale. The results show that whole-model right-sizing is suboptimal—there is no single TPC configuration that fits all kernels. Instead, a substantial opportunity lies in right-sizing at the kernel level. First, individual kernels exhibit diverse scaling behaviors: some scale linearly, while others show diminishing returns. Second, the extent to which execution time is distributed across many kernels varies from workload to workload—highlighting the need for adaptive, per-kernel scheduling to fully optimize GPU resource consumption.

Modeling Kernel Scaling. LithOS introduces on-the-fly TPC right-sizing at the granularity of kernels (Figure 9, Step ⑥). The atoms of a given kernel inherit its allocated TPCs, as they exhibit the same scaling behavior as the kernel itself. To this end, LithOS introduces a model-based approach that interpolates the scaling of individual kernels based on two points: the latencies of a kernel running with all TPCs and just one TPC. It then fits a curve of the form

$$l = \frac{m}{t} + b$$

to these points, where l is the predicted latency, t is the corresponding number of TPCs, and m and b are constants. Note that the form of this curve is consistent with Amdahl's law for parallel speedup. Intuitively, b can be thought of as how long it takes for a single one of the kernel's thread blocks to complete on a single SM, and m quantifies the extent to which a kernel can take advantage of parallel processors.

Filtering Outliers. We find that, in practice, this simple model accurately captures the scaling behavior of most deep learning kernels. However, a small number of outlier kernels—typically those with very short runtimes—deviate from the model, as they fail to benefit from large TPC allocations and are inherently harder to model. To handle these cases, we introduce a *filtering* heuristic based on a kernel's thread block occupancy. Specifically, we estimate the number of TPCs a kernel can effectively utilize by dividing its total number of thread blocks by the occupancy per TPC—that is, the number of thread blocks a single TPC can execute concurrently. LithOS already tracks thread blocks per kernel as part of atomization, while occupancy can be queried from the driver API [43]. This heuristic provides an intuitive upper bound on useful TPC allocations per kernel, helping avoid overprovisioning even for difficult-to-model kernels.

Operation. When a kernel is submitted to LithOS, the dispatch thread first applies the filtering heuristic to estimate an upper bound on the number of TPCs the kernel can effectively utilize. If this estimate is lower than the job's allocated TPCs, the kernel is launched using the estimated bound. Otherwise, the dispatch thread leverages the learned scaling model to determine the minimum number of TPCs that would increase the kernel's latency by, at most, a multiplicative factor k that we call the *latency slip parameter*. This tunable parameter allows users and administrators to intuitively configure LithOS, for example, by specifying that up to 10% performance degradation is acceptable. Overall, LithOS enables highly efficient fine-grained right-sizing, while its modeling and scaling techniques offer a robust and accurate solution—as we will see in Section 8.2.

Supporting Hardware-Aware Optimizations. The right-sizing approach of LithOS is orthogonal to, and naturally complements, hardware-aware optimizations performed at the framework and compiler layers. These optimizations typically focus on tailoring kernel implementations to intra-SM architectural features (such as Tensor Cores), warp-level techniques, and memory hierarchy tuning. In contrast, LithOS operates at the inter-SM level by managing kernel-to-TPC allocations. Because these domains are independent, hardware-aware optimizations can seamlessly coexist with LithOS.

5.6 Transparent Power Management

LithOS is well-positioned to enable transparent and efficient power management via DVFS. Just as right-sizing lets LithOS adapt resource allocation based on kernel scalability across

TPCs, DVFS enables vertical scaling through frequency adjustment. Figure 13 shows how kernels from various workloads respond to frequency scaling. Many exhibit predictable behavior, creating opportunities for energy savings with bounded performance impact. To achieve efficient DVFS, LithOS must address two key challenges. First, current GPUs support relatively slow frequency switching (~50 ms). While future architectures may reduce this latency [16], DVFS remains impractical for models with very short kernels. Thus, LithOS must consider the cumulative impact of scaling across kernel sequences. Second, although many kernels scale linearly with frequency, enabling significant energy savings, LithOS must balance these gains against increased latency.

Modeling Frequency Scaling. LithOS introduces a transparent sequence-based kernel frequency scaling model that guides DVFS (Figure 9, Step ⑦). Similarly to right-sizing, the atoms of a given kernel inherit its frequency target, as they exhibit the same scaling behavior as the kernel itself. Specifically, each kernel is assigned a weight w, the ratio of its total runtime to the cumulative runtime of all the kernels in a particular stream. Then, LithOS approximates each kernel's relative slowdown as proportional to the fractional drop in frequency based on a first-order Taylor approximation:

$$k = \frac{lat(f_{th})}{lat(f_{max})} - 1 = s \cdot (\frac{f_{max}}{f_{th}} - 1)$$

where $lat(f)$ is the kernel's latency at frequency f. Specifically, f_{max} is the maximum frequency, and f_{th} is one of the device's supported frequencies. Each kernel's sensitivity is

$$s = \frac{k}{\frac{f_{max}}{f_{th}} - 1}$$

and the aggregate sensitivity S across all kernels is equal to $\sum w * s$. Similarly, the total slowdown is equal to

$$S \cdot (\frac{f_{max}}{f_{final}} - 1) \leq k$$

and thus the final frequency that LithOS assigns to the workload is $f_{final} = \frac{f_{max}}{1+\frac{k}{S}}$. Intuitively, compute-bound kernels whose slowdown scales linearly with frequency reduction skew the final frequency closer to the maximum according to their sensitivity, while memory-bound kernels whose slowdown is frequency-insensitive shift the final frequency to lower levels depending on their weight.

Operation. Similar to right-sizing, LithOS uses a multiplicative factor k, the *latency slip parameter*, to guide DVFS decisions. At runtime, this parameter is used to evaluate the scaling model and select a target frequency. Due to the high latency of switching, LithOS adopts a conservative strategy and extends its learning period to avoid unnecessary transitions. Initially, LithOS collects per-kernel metadata at maximum frequency, forcing unseen kernels to run at max frequency. At first, a kernel is assumed to scale linearly,

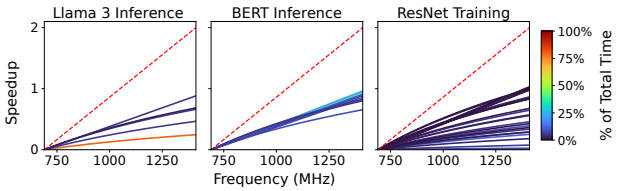


Figure 13. *LithOS's interpolated frequency scaling curves.*

and its frequency is reduced based on the configured k. Depending on the observed performance, LithOS either further lowers the frequency or stops after confirming linear behavior. Over time, it fits the collected data to the scaling model, enabling more informed and efficient DVFS decisions.

5.7 Online Latency Prediction

The latency prediction module learns the execution time of kernels, enabling the optimizations carried out by all of LithOS's components. In particular, it enhances TPC Stealing by estimating the duration of outstanding tasks and guides the number of atoms the Kernel Atomizer splits each kernel into. It further assists right-sizing and DVFS by providing the latencies that are used to calculate speedups based on TPC and frequency scaling. This obviates the need for extensive offline profiling, which is impractical for a transparent OS.

Latency prediction operates separately for independent launch queues, allowing LithOS to dynamically adapt to the behavior of different applications. During execution, the module records kernel latencies and refines its predictions. Each kernel's latency varies based on the allocated TPCs, the GPU frequency, and the granularity at which it is atomized; therefore, the prediction module continuously monitors these conditions. In the case where such metadata are not available for a specific atom, the prediction module is conservative, assuming optimal linear scaling.

One pitfall in achieving accurate kernel latency prediction is assuming a given kernel always has the same latency. In practice, duration can depend on launch parameters and inputs. For instance, a single `Conv` kernel function may be used across model layers with varying tensor sizes. This necessitates that the latency prediction module track operators rather than kernel functions. By recording explicit synchronization events, we can determine the start and end of a batch. We associate kernel launches with an ordinal index k, referring to the k^{th} kernel after the start of a batch. This uniquely identifies operator nodes in the model's data flow graph (DFG), despite LithOS lacking explicit access to this higher-level information. This additional ordinal index is sufficient to identify model operators and make accurate latency predictions.

6 Implementation

We implement a prototype of LithOS targeting NVIDIA GPUs in ~5000 lines of Rust, excluding macro-generated code for

interposing the entire CUDA Driver API. The LithOS prototype supports Ampere and Hopper architectures and applications running natively or in containers. To enable concurrent execution across GPU contexts, we build on top of MPS.

Interposition Architecture. LithOS is fully transparent to applications, supporting the diverse ML ecosystem and full GPU stack. It interposes at the CUDA Driver API—the lowest common denominator—so applications interact with LithOS rather than the driver, while preserving CUDA call semantics. This ensures generality and transparency at the OS level, enabling unmodified ML frameworks and libraries such as PyTorch, TensorFlow, JAX, TensorRT, and cuDNN. LithOS implements only a small subset of Driver APIs (e.g., `cuLaunchKernel`); our toolchain auto-generates the rest. Unlike prior CUDA API interposition systems [68], LithOS avoids complex cross-address-space marshaling, streamlining support for new CUDA versions and long-term OS maintenance.

TPCs and Atomization. For TPC mappings, we extend prior reverse-engineering work `libsmctrl` [4] and add support for Hopper, including handling its new TPC masking layout. `libsmctrl` is an interface that exposes TPC mappings without additional logic. In LithOS, we reimplement functionality to identify TPCs through the Queue MetaData (QMD) data structure and enable dynamic TPC allocation on kernel launch. Furthermore, on Hopper GPUs, NVIDIA introduced Thread Block Clusters—a new scheduling abstraction. We reverse-engineer these mappings and ensure atoms are always multiples of the cluster size. We verified functionality across NVIDIA datacenter GPUs: Ampere (A30, A100), Hopper (H100), and Ada Lovelace (L4). On top of the TPC mapping interface, the core implementation of LithOS relies on a set of scheduling mechanisms responsible for work submission, TPC scheduling, stealing, hardware right-sizing, atomization, outstanding work monitoring, and power management. We believe that future GPU drivers can expose these APIs to simplify the implementation of LithOS.

For kernel atomization, we inject `Prelude` logic by modifying the QMD struct used to launch kernels [4, 17]. To allocate appropriate resources, LithOS first launches the original kernel, allowing the CUDA Driver to configure the environment. We then patch the QMD's program address to point to the `Prelude`. As a result, execution begins at the `Prelude` while retaining the original kernel's resources. Due to space limits, we defer low-level reverse-engineering details to a separate technical report. The QMD reverse-engineering effort is minimal and often completed within days of a new architecture.

Special Kernels. There are a few cases of kernels that may require special attention from LithOS. To extend atomization for CUDA Graphs, LithOS can interpose graph creation APIs and atomize graphs into subgraphs, ensuring correct execution ordering. Furthermore, some kernels comprise thread blocks that synchronize with each other, e.g., with `grid_group::sync()`. These kernels require a certain number of SMs during execution. LithOS can simply return the number of allocated SMs for the `CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT` in `cuDeviceGetAttribute`. Furthermore, for special kernels that involve cross-block synchronization or persistent kernels, LithOS disables stealing and atomization.

Model	Mem. (GiB)	Batch Size	Latency (ms)
VGG-19 [56]	17.4	120	291
ResNet-50 [28]	18.4	184	281
MobileNetV2 [53]	18.4	216	254
DLRM [40]	6.7	32768	74
BERT-Large [20]	17.3	20	159
Llama 3 Finetuning	32.0	4	690

Table 1. *Training model parameters.*

7 Experimental Setup and Methodology

Testbed. Experiments were conducted on a `1x A100 (SXM4)` Lambda Labs GPU instance with 30 CPU cores and 216 GB of host memory. The A100 GPU has 108 SMs and 40 GB of memory. The server was configured with Ubuntu 22.04, CUDA 12.6, Rust 1.83.0-nightly, Python 3.10, PyTorch 2.3, TensorRT 10.1, TensorRT-LLM 0.11.0, and Triton 24.07.

Baselines. We compare LithOS to all four NVIDIA GPU sharing methods: *Time slicing*, *MPS*, stream *Priority*, and *MIG*. We further compare against SOTA prior work across the spectrum of transparent solutions *TGS* [64], application modifications *REEF* [27], and both application modifications and offline profiling *Orion* [59]. We used the open-source TGS directly but had to reimplement Orion and REEF using our own interposition infrastructure since the available code was tied to specific CUDA drivers and software stacks. We extend REEF and Orion to handle multiple HP apps in a straightforward manner. For REEF, BE kernels are not launched if *any* HP app is running. For Orion, BE kernels are not launched if they contend with *any* HP kernel.

Models and Configurations. All high-priority inference tasks run on NVIDIA's Triton Inference Server with dynamic batching [14]. RetinaNet runs on ONNX Runtime, while the other served models run on NVIDIA's TensorRT and TensorRT-LLM backends. We choose three representative vision models (RetinaNet [38], YOLOv4 [5], and ResNet-50 v1.5 [28]) and three language models (Llama 3 8B [25], GPT-J 6B [62], and BERT-Large [20]) as inference workloads. For large language models, we use a Microsoft Azure trace [58]. For the best effort training tasks, we use three vision models, ResNet-50, MobileNetV2, VGG-19, and two language models, DLRM and BERT-Large, as listed. The training batch size is adjusted to use at most half of the GPU DRAM to keep all models in memory when stacking. The best effort training task runs continuously. More details are in Tables 1 and 2.

Latency Constraints. For workloads which require a latency SLO, we use latency constraints from the MLPerf datacenter inference benchmark [50] (Table 2). This collection of results is an industry standard for evaluating the performance of inference servers, and the constraints vary from 2.3×–7.4× of baseline end-to-end request latency.

Model	Framework	Load (rps)	Constraint (ms)
ResNet [28]	TensorRT	1000	15
RetinaNet [38]	ONNX Runtime	9	100
Llama 3 [25]	TensorRT-LLM	0.5	2000
GPT-J [62]	TensorRT-LLM	0.5	2000
BERT [20]	TensorRT	30	130

Table 2. *Inference services for inference-only multitenancy.*

8 Evaluation

Our evaluation answers the following questions:

1. Does LithOS improve performance for different multitenancy environments and SOTA prior works?
2. What are the capacity savings due to LithOS's hardware right-sizing?
3. What are the energy savings of LithOS's DVFS?
4. How do different LithOS features affect performance?

8.1 Performance in Multitenant Environments

In the following experiments, we disable the right-sizing and power management features of LithOS to provide an apples-to-apples comparison to other systems in terms of scheduling efficiency alone. We evaluate these features afterwards.

Inference-only Multitenancy. We evaluate LithOS in a multitenant environment with three inference applications: two high-priority (HP) and one best-effort (BE). This is a realistic stacking scenario as current GPUs can satisfactorily fit two HP models, while the BE task utilizes the remaining resources. The first HP app, *HP A*, has a latency-oriented SLO: percentage of requests executed within a latency constraint. The second, *HP B*, has a throughput-oriented SLO: attained throughput as a percentage of the case where it executes alone. These vary according to the model (Table 2).

The BE and HP B models are chosen from Llama 3, GPT-J, and BERT. For HP A, we add ResNet and RetinaNet. We run all possible combinations. HP apps follow Poisson load and run on the Triton inference server, while BE apps execute in a closed loop. All model latencies are measured end-to-end.

We compare LithOS against all configurations. For systems that support partitioning, HP A and HP B are isolated on partitions of 75% and 25%, respectively. MIG's limited partitioning configurations cannot support a 25%-75% split, so we use a 3/7-4/7 split instead. MIG and Limits cannot support a BE app, but only apps with provisioned resources; therefore, the BE does not run on these systems. There is no way to isolate multiple latency-sensitive applications on systems like Priority, REEF, TGS, and Orion. For these, we set both HP apps to high priority and the BE to low priority.

Figure 14 compares all systems across two dimensions: SLO attainment and throughput. "SLO" of 100% means both HPs reach 100% attainment. "Throughput" of 1 means that the throughput achieved is as much as if any of the apps had the entire device. Unsurprisingly, MPS sets the bar for throughput. MPS's fine-grained, intra-SM stacking ensures

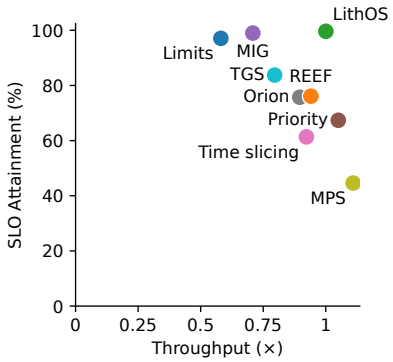


Figure 14. *SLO Attainment and throughput by system.*

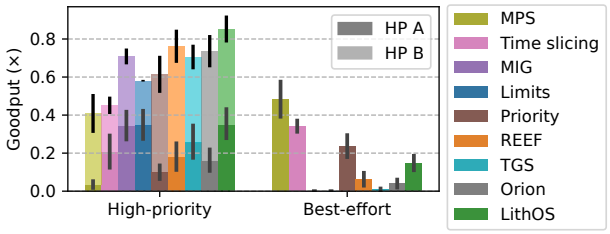


Figure 15. *Inference-only multitenancy: Goodput by app.*

device resources are maximally utilized, and it allows more throughput when stacking than any application could have alone; hence, it achieves a throughput of 1.11. MPS's throughput comes at the cost of SLO attainment, at 45%. MIG and thread limits both successfully meet SLOs. This is expected, as each system minimizes interference by devoting resources to individual apps. However, the partitions are not fully utilized without a BE app. As a result, aggregate throughput drops to 0.58 and 0.71 for thread limits and MIG, respectively. Without isolating HP apps, priority-only systems cannot attain SLOs, with TGS leading at 84%. LithOS provides the best of both worlds, as it provides spatial isolation like MIG with an SLO attainment of 100% and a throughput of 1.

Where do LithOS's benefits come from? Figure 15 shows LithOS consistently leading in goodput (throughput excluding HP A requests that violate SLOs) for the HP apps while still allowing significant (0.15) BE throughput. While the partitioning systems match LithOS in HP A goodput, they lack in HP B goodput: MIG at 0.37 vs. LithOS at 0.50. They also cannot support any BE throughput, while LithOS allows HP apps to steal unused resources from each other and further support BE throughput. No SOTA system can perform effectively across all requirements. Specifically, Orion outperforms in latency-sensitive throughput, TGS in HP throughput, and REEF in best effort. Only LithOS provides the best HP throughput while sustaining high BE throughput.

Diving deeper, we next look into the latencies of the HP A app in Figure 16. The figure shows the P_{99} latencies for each model averaged across all combinations. Latencies diverge

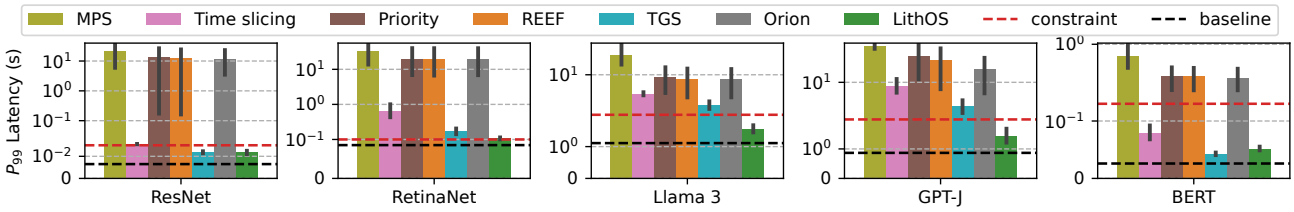


Figure 16. *Inference stacking multitenancy: HP A tail latencies by model.*

in many cases, with only LithOS and the partitioning systems limiting latencies to the constraints. MPS is the worst-performing with respect to latencies; LithOS's are 13× better. LithOS reduces latencies by 4× compared to Orion. This is expected as Orion cannot handle multiple HP apps. TGS limits latencies much more effectively than Orion and REEF, but LithOS still improves over it by 1.2×. Overall, LithOS provides a robust solution for inference stacking.

Hybrid Inference/Training Multitenancy. In the first scenario, we modeled a realistic multitenant scenario where two primary HP inference jobs are placed on a GPU, and a BE task makes use of the remaining idle resources. Next, we evaluate an alternative realistic scenario that is commonly examined by prior work. In this experiment, we stack an HP inference app that has a latency-oriented SLO with a training BE app. Similar to the inference-stacking experiment, resources unused by the sensitive inference app should be donated to the best-effort training job. At the same time, service latency must not increase. We choose the inference model from the set: Llama 3 8B, GPT-J 6B, BERT-Large, RetinaNet, and YOLOv4. We choose the training model from those listed in Table 1. We run all model combinations, and our client creates Poisson loads. Load parameters are chosen to keep GPU utilization around 80% for the HP app.

Figure 17 shows the P_{99} HP latency and aggregate throughput, averaged across all training models. HP throughput is normalized to the load before being added to the BE throughput, which is normalized to the case where the BE model runs alone on the device. Latencies are also normalized to the HP running alone on the device. MPS yields latencies 5.83× the ideal case, and its service throughput is the lowest at 60%. Time slicing fares better as it enables the long-running kernels of the best-effort models to be preempted, guaranteeing the service approximately 50% of the GPU time. MIG performs similarly to time slicing by allocating 50% of the GPU to the service spatially rather than temporally. However, both methods fail to sustain peak HP throughput. Stream priority also falls short, leading to a 2.89× increase in service latency and service throughput as low as 68%.

Both TGS and REEF also struggle to maintain low service latencies. TGS has an average inference latency of 1.41× the ideal, and REEF is 2.89×. TGS's poor performance stems from its adaptive rate control mechanism, which assumes a constant work arrival rate. This assumption is invalid for inference services, which have unpredictable load patterns. REEF fails to sufficiently throttle the training model, allowing tail latencies to reach 8.93×. In contrast, LithOS maintains a tail latency within 20% of the ideal. On average, this is 2.34× and 1.18× over REEF and TGS, respectively. Compared to the native MPS solution, LithOS reduces latency by up to 13.54× and 4.7× on average. LithOS maintains service throughput within 1% of load in the worst case. LithOS improves training throughput by an average of 34× and aggregate throughput by 1.35× vs. TGS. In total, LithOS improves aggregate throughput 1.23×–1.57× with an average of 1.38×.

8.2 Kernel-SM Right-Sizing

Capacity Savings. Figure 18 shows the capacity savings due to right-sizing with LithOS. We compute savings by comparing the time-weighted average of TPC utilization before and after right-sizing. LithOS provides excellent savings of up to 51%, and a mean of 26% across all workloads. We expect that in future GPU architectures with an increased number of TPCs, the fine-grained right-sizing approach of LithOS will provide even greater savings.

Latency and Throughput Cost. With a latency slip parameter of 1.1, the performance cost of right-sizing in terms of P_{99} and throughput is modest. The mean increase in P_{99} and decrease in throughput are both 4%. Our latency slip parameter is conservative because not all of the end-to-end execution time of each inference or training iteration is spent inside a GPU kernel; this does not impede tuning in practice.

Accuracy. To quantify the accuracy of our prediction technique, we compute the kernel-execution-time weighted average of the R^2 values for the curves we fit (i.e., for kernels where the possible TPCs value exceeds the threshold). Across all evaluated workloads, the average R^2 values range from 0.92 (Llama finetuning) to 0.99 (RetinaNet inference), indicating that our linear models are sufficiently accurate. Future work can explore more involved modeling to leverage LithOS to extend right-sizing to even more diverse GPU workloads.

8.3 Kernel-Dependent DVFS

In this experiment, we compare the energy consumption of the LithOS DVFS mechanism to the default settings of the GPU, for a variety of inference and training jobs with high GPU utilization; the baseline runs mostly under the maximum GPU frequency (1410 MHz). We run experiments for a

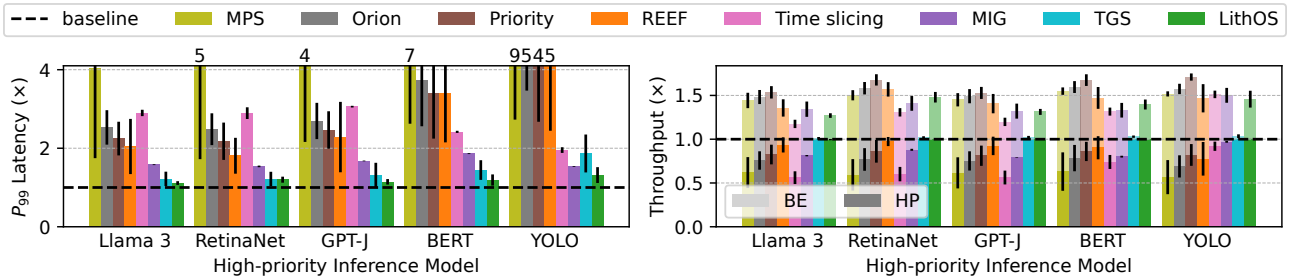


Figure 17. *Hybrid multitenancy: (a) P_{99} service latency and (b) aggregate throughput.*

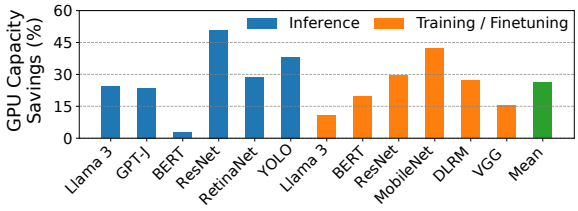


Figure 18. *Hardware right-sizing GPU capacity savings.*

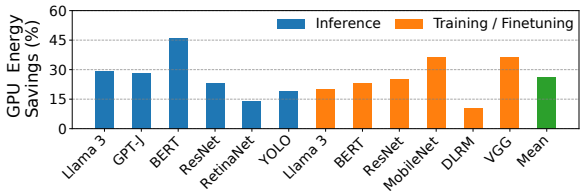


Figure 19. *Power management GPU energy savings.*

fixed number of requests or training epochs for a fair energy comparison. Energy is calculated as the average power consumption multiplied by the time required for the experiment to complete. We measure power with `nvidia-smi` every 100 ms, the smallest granularity at which the tool operates.

Energy Savings. Figure 19 shows the energy savings of LithOS's DVFS mechanism across different inference and training workloads. We define energy savings as the difference between executing the workload at default frequency and under LithOS's DVFS policy. LithOS provides significant energy savings of up to 46%, and a mean of 26% across all workloads without offline profiling requirements.

Performance Cost. The slip parameter for this experiment was set at 1.1, and the mean increase in P_{99} latency is 7%. The minimal increase in P_{99} latency demonstrates that LithOS's DVFS policy is inherently conservative. It respects latency constraints across workloads while transparently providing substantial energy savings. Finer-grained frequency control could unlock additional energy savings.

8.4 Ablation and Case Studies

Multi-tenancy Breakdown. Figure 20 presents a performance analysis for inference-training as explored in Figure 17. Enabling the TPC scheduler improves HP tail latencies to 1.38× of ideal by throttling BE work, while maintaining ideal HP throughput. Kernel Atomization offers additional gains, reducing tail latencies to an average of 1.19× and up to 1.55×, by splitting long BE kernels and improving TPC Stealing. Because of space limitations, we plot only latencies. Kernel Atomization does introduce a 10% throughput overhead, as LithOS prioritizes HP workloads by reducing BE throughput. Overall, each of LithOS's features plays a crucial role in optimizing end-to-end performance.

Kernel Atomization. To highlight the challenges of scheduling long-running kernels, we collocate an HP BERT inference workload with either a BE VGG training or a BE Llama 3 inference. In Figure 21, we vary (a) the batch size of the BE training job and (b) the sequence length of the BE inference job and measure the P_{95} latency of the HP inference job. LithOS outperforms REEF by 6.5× and 3.9× in (a) and (b), respectively. Unlike REEF, which simply throttles BE work, LithOS accounts for kernel durations, which can vary significantly. To understand the impact of Kernel Atomization, we further evaluate LithOS with Kernel Atomization disabled. Kernel Atomization provides an improvement of 2× and 1.3× in (a) and (b), respectively. As described in Figure 11, kernel durations grow with training batch size and inference input sequence length. As Kernel Atomization allows LithOS to schedule at thread block granularity, HoL blocking is minimized. Consequently, the HP tail latency for the full LithOS system is within 14% (or 1 ms) or 7% (or 0.45 ms) of ideal for even the largest batch size or sequence length, respectively. Without atomization, noisy neighbors with large batch sizes or long sequence lengths can substantially degrade the performance of latency-critical tasks.

Latency Prediction Module. Next, we evaluate the accuracy of the latency prediction module of LithOS that enhances the TPC Scheduler and the Kernel Atomizer. We record the predicted atom latencies and compare them with the corresponding CUDA events, treating absolute errors greater than 50 μs as mispredictions. Overall, we find very low misprediction rates of just 0.9% and 0.38% for the HP workloads in inference-inference and inference-training environments, respectively. Additionally, the prediction error tails are small with P_{99}s of 49 μs and 31 μs. Misprediction rates for the BE workloads are higher at 14% and 11% for inference-inference and inference-training, respectively.

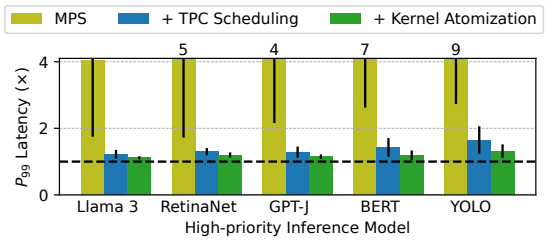


Figure 20. *Breakdown of LithOS features for inf-train.*

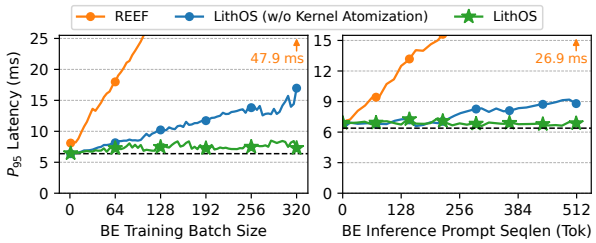


Figure 21. *P_{95} latency of HP inference collocated with varied (a) batch sizes training and (b) sequence lengths inference.*

This is acceptable as BE work is frequently preempted by HP work and has lower priority for GPU resources. Future work can explore more complex modeling to reduce error rates; however, our evaluation shows that the existing predictor's accuracy allows for sufficient performance isolation in practice.

Overheads. The interposition and control logic of LithOS impose modest overhead. By measuring inference models without multitenancy against the vanilla NVIDIA driver, LithOS adds an overhead of only 4%. Atomization adds less than 1%. For comparison, the overhead of TGS and REEF is close to 2%, while Orion stands at 6%. In general, we believe this overhead is small and can be further optimized away.

Memory Contention. From our study of production services, memory capacity contention is not a concern as models are kept in GPU memory. In our evaluation, bandwidth contention can be a concern for some workload combinations. Using MIG and thread limits, we estimate that bandwidth isolation would yield 4–13% performance gains for contention-heavy cases. Compute isolation is significantly more critical, yielding more than an order of magnitude improvement.

9 Discussion

Other GPU Resources. This work focuses on compute and power, but the same principles extend to other resources. Prior systems target GPU memory [1, 3, 6, 32, 64], bandwidth [29, 70], PCIe [33, 70], SSDs [3, 49], and networking [57]. Others interpose at higher layers, via custom drivers [34, 52] or CUDA APIs [7, 67]. GPUfs [55] is the closest to an OS-like design, providing file-system extensions. LithOS complements these efforts, offering a foundation to virtualize and manage additional GPU resources.

Driver and Hardware Support. LithOS demonstrates what is possible with today's hardware, but additional driver and architectural support would unlock further gains: kernel-to-SM assignment, preemption, cache and memory partitioning, NUMA-style placement, fine-grained (sub-ms) DVFS, per-SM power control, and richer context management. Similar capabilities are standard in CPUs and will be increasingly essential as GPUs scale, integrate multiple dies (e.g., Blackwell), and grow more heterogeneous. Open-source drivers will be critical for enabling efficient OS-level control.

While LithOS targets TPC-level scheduling, emerging heterogeneity within SMs (e.g., tensor cores) highlights opportunities for intra-SM resource management. Hardware support here could enable even finer-grained efficiency.

Lessons Learned. A central lesson is that both spatial and temporal partitioning are required for efficient GPU multitenancy. Without dedicated resources, latency-critical tasks suffer interference, as in MPS, while without time-sharing, utilization drops, as in MIG. Fine-grained control is equally crucial: TPC scheduling allows GPUs to be sliced into many more virtual devices than MIG, improving packing and utilization, while kernel-level atomization enables fast switching to high-priority tasks, strengthening isolation. Power management also emerged as a key challenge. Device-wide DVFS proves effective for today's relatively well-behaved kernels, but future workloads are more diverse and input-dependent, demanding finer-grained mechanisms that adapt at sub-ms timescales, distinguish between compute, caches, and memory, and apply power controls spatially. Finally, our experience showed that the CUDA Driver API forms a stable "narrow waist" for interposition. By intercepting only a handful of calls, LithOS remains lightweight, portable across driver versions, and easy to retarget from Ampere to Hopper, suggesting this is a robust control point for OS research.

LithOS opens a new direction for GPU operating systems. By coupling OS design with forthcoming hardware extensions, future ML systems can deliver stronger isolation, higher utilization, and significant energy savings.

10 Conclusion

This paper introduced LithOS, a first step towards an operating system for efficient machine learning on GPUs. LithOS operates transparently to the entire ML stack; through mechanisms like TPC Scheduling, Kernel Atomization, hardware right-sizing, and power management, LithOS significantly improves GPU efficiency while laying the foundation for future OS research on GPUs.

Acknowledgments

This work was funded in part by NSF grants CNS-2239311, CCF-2217016, a Meta Faculty Award, and a Wilton E. Scott Institute Faculty Award. We thank the anonymous reviewers for all of their valuable feedback.

References

[1] Georgios Alexopoulos and Dimitris Mitropoulos. 2024. nvshare: Practical GPU Sharing without Memory Size Constraints. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 16–20.

[2] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 499–514.

[3] Joshua Bakita and James H. Anderson. 2022. Enabling GPU Memory Oversubscription via Transparent Paging to an NVMe SSD. In *2022 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 370–382.

[4] Joshua Bakita and James H. Anderson. 2023. Hardware Compute Partitioning on NVIDIA GPUs. In *Proceedings of the 29th IEEE Real-Time and Embedded Technology and Applications Symposium*. 54–66.

[5] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. 2020. YOLOv4: Optimal Speed and Accuracy of Object Detection. arXiv:2004.10934 [cs.CV] https://arxiv.org/abs/2004.10934

[6] Chia-Hao Chang, Jihoon Han, Anand Sivasubramaniam, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-Mei Hwu. 2024. GMT: GPU Orchestrated Memory Tiering for the Big Data Era. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 464–478. doi:10.1145/3620666.3651353

[7] Hung-Hsin Chen, En-Te Lin, Yu-Min Chou, and Jerry Chou. 2023. Gemini: Enabling Multi-Tenant GPU Sharing Based on Kernel Burst Estimation. *IEEE Transactions on Cloud Computing* 11, 1 (2023), 854–867. doi:10.1109/TCC.2021.3119205

[8] Qichen Chen, Hyerin Chung, Yongseok Son, Yoonhee Kim, and Heon Young Yeom. 2021. smCompactor: a workload-aware fine-grained resource management framework for GPGPUs. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing* (Virtual Event, Republic of Korea) *(SAC '21)*. Association for Computing Machinery, New York, NY, USA, 1147–1155. doi:10.1145/3412841.3441989

[9] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 199–216. https://www.usenix.org/conference/atc22/presentation/choi-seungbeom

[10] Marcus Chow, Ali Jahanshahi, and Daniel Wong. 2023. KRISP: Enabling Kernel-wise RIght-sizing for Spatial Partitioned GPU Inference Servers. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 624–637. doi:10.1109/HPCA56546.2023.10071121

[11] Marcus Chow and Daniel Wong. 2024. CoFRIS: Coordinated Frequency and Resource Scaling for GPU Inference Servers. In *Proceedings of the 14th International Green and Sustainable Computing Conference* (Toronto, ON, Canada) *(IGSC '23)*. Association for Computing Machinery, New York, NY, USA, 45–51. doi:10.1145/3634769.3634808

[12] NVIDIA Corporation. [n. d.]. Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html. Accessed: April 14, 2025.

[13] NVIDIA Corporation. 2023. *NVIDIA H100 Tensor Core GPU Architecture*. Technical Report. NVIDIA Corporation, Santa Clara, CA.

[14] NVIDIA Corporation. 2024. Triton Inference Server. https://developer.nvidia.com/triton-inference-server. Accessed: May 8, 2024.

[15] NVIDIA Corporation. 2025. NVIDIA Multi-Instance GPU User Guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html. Accessed: April 14, 2025.

[16] NVIDIA Corporation. 2025. NVIDIA RTX Blackwell GPU Architecture. https://images.nvidia.com/aem-dam/Solutions/geforce/blackwell/nvidia-rtx-blackwell-gpu-architecture.pdf.

[17] NVIDIA Corporation. 2025. Open GPU documentation. https://github.com/NVIDIA/open-gpu-doc.

[18] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 613–627. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

[19] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2021. Enable simultaneous DNN services based on deterministic operator overlap and precise latency prediction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 15, 15 pages. doi:10.1145/3458817.3476143

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL] https://arxiv.org/abs/1810.04805

[21] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. 2020. GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 492–506. doi:10.1145/3419111.3421284

[22] Benj Edwards. 2025. Nvidia announces "Rubin Ultra" and "Feynman" AI chips for 2027 and 2028. https://arstechnica.com/ai/2025/03/nvidia-announces-rubin-ultra-and-feynman-ai-chips-for-2027-and-2028/

[23] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 281–297. https://www.usenix.org/conference/osdi20/presentation/fried

[24] Yanjie Gao, Yichen He, Xinze Li, Bo Zhao, Haoxiang Lin, Yoyo Liang, Jing Zhong, Hongyu Zhang, Jingzhou Wang, Yonghua Zeng, et al. 2024. An Empirical Study on Low GPU Utilization of Deep Learning Jobs. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[25] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, and Ahmad Al-Dahle et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[26] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 443–462. https://www.usenix.org/conference/osdi20/presentation/gujarati

[27] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 539–558. https://www.usenix.org/conference/osdi22/presentation/han

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV] https://arxiv.org/abs/1512.03385

[29] Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Rajkumar. 2019. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 29–41.

[30] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, unjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19)*. USENIX Association, USA, 947–960.

[31] Andreas Kosmas Kakolyris, Dimosthenis Masouros, Petros Vavaroutsos, Sotirios Xydis, and Dimitrios Soudris. 2024. SLO-aware GPU Frequency Scaling for Energy Efficient LLM Inference Serving. arXiv:2408.05235 [cs.DC] https://arxiv.org/abs/2408.05235

[32] Woosung Kang, Jinkyu Lee, Youngmoon Lee, Sangeun Oh, Kilho Lee, and Hoon Sung Chwa. 2024. RT-Swap: Addressing GPU Memory Bottlenecks for Real-Time Multi-DNN Inference. In *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 373–385.

[33] Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Ragunathan Rajkumar. 2011. RGEM: A responsive GPGPU execution model for runtime engines. In *2011 IEEE 32nd Real-Time Systems Symposium*. IEEE, 57–66.

[34] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. 2011. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Portland, OR) *(USENIXATC'11)*. USENIX Association, USA, 2.

[35] Yunseong Kim, Yujeong Choi, and Minsoo Rhu. 2022. PARIS and ELSA: an elastic scheduling algorithm for reconfigurable multi-GPU inference servers. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 607–612. doi:10.1145/3489517.3530510

[36] Beth Kindig. 2024. AI power consumption: Rapidly becoming mission-critical. https://www.forbes.com/sites/bethkindig/2024/06/20/ai-power-consumption-rapidly-becoming-mission-critical/

[37] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. 2022. MISO: Exploiting Multi-Instance GPU Capability on Multi-Tenant GPU Clusters. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 173–189. doi:10.1145/3542929.3563510

[38] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2018. Focal Loss for Dense Object Detection. arXiv:1708.02002 [cs.CV] https://arxiv.org/abs/1708.02002

[39] Xuanzhe Liu, Yihao Zhao, Shufan Liu, Xiang Li, Yibo Zhu, Xin Liu, and Xin Jin. 2024. MuxFlow: efficient GPU sharing in production-level clusters with more than 10000 GPUs. *Science China Information Sciences* 67, 12 (2024), 222101. doi:10.1007/s11432-024-4227-2

[40] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. arXiv:1906.00091 [cs.IR] https://arxiv.org/abs/1906.00091

[41] Microsoft Network. 2024. Dell exec reveals Nvidia has a 1,000 watt GPU in the works. https://www.msn.com/en-us/lifestyle/other/dell-exec-reveals-nvidia-has-a-1-000-watt-gpu-in-the-works/ar-BB1jlE8f Accessed: June 24, 2024.

[42] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. 2023. Paella: Low-latency Model Serving with Software-defined GPU Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 595–610. doi:10.1145/3600006.3613163

[43] NVIDIA Corporation. [n. d.]. *NVIDIA CUDA Driver API Documentation: Occupancy*. NVIDIA Corporation. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__OCCUPANCY.html

[44] NVIDIA Corporation. 2024. *Virtual GPU Software User Guide (v13.0)*. NVIDIA Corporation. https://docs.nvidia.com/vgpu/13.0/grid-vgpu-user-guide/index.html Version 13.0, Accessed: 2025-08-28.

[45] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. arXiv:1712.06139 [cs.DC]

[46] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[47] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Brijesh Warrier, Nithish Mahalingam, and Ricardo Bianchini. 2024. Characterizing Power Management Opportunities for LLMs in the Cloud. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 207–222. doi:10.1145/3620666.3651329

[48] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. 2024. Power-aware Deep Learning Model Serving with μ-Serve. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 75–93. https://www.usenix.org/conference/atc24/presentation/qiu

[49] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, C. J. Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. 2023. GPU-Initiated On-Demand High-Throughput Storage Access in the BaM System Architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 325–339. doi:10.1145/3575693.3575748

[50] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 2019. MLPerf Inference Benchmark. arXiv:1911.02549 [cs.LG]

[51] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 397–411. https://www.usenix.org/conference/atc21/presentation/romero

[52] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 233–248. doi:10.1145/2043556.2043579

[53] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[54] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association

for Computing Machinery, New York, NY, USA, 322–337. doi:10.1145/3341301.3359658

[55] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2014. GPUfs: Integrating a file system with GPUs. *ACM Trans. Comput. Syst.* 32, 1, Article 1 (Feb. 2014), 31 pages. doi:10.1145/2553081

[56] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556 [cs.CV] https://arxiv.org/abs/1409.1556

[57] Athinagoras Skiadopoulos, Zhiqiang Xie, Mark Zhao, Qizhe Cai, Saksham Agarwal, Jacob Adelmann, David Ahern, Carlo Contavalli, Michael Goldflam, Vitaly Mayatskikh, Raghu Raja, Daniel Walton, Rachit Agarwal, Shrijeet Mukherjee, and Christos Kozyrakis. 2024. High-throughput and Flexible Host Networking for Accelerated Computing. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 405–423. https://www.usenix.org/conference/osdi24/presentation/skiadopoulos

[58] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2024. DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency. arXiv:2408.00741 [cs.AI] https://arxiv.org/abs/2408.00741

[59] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 1075–1092. doi:10.1145/3627703.3629578

[60] Cheng Tan, Zhichao Li, Jian Zhang, Yu Cao, Sikai Qi, Zherui Liu, Yibo Zhu, and Chuanxiong Guo. 2021. Serving DNN Models with Multi-Instance GPUs: A Case of the Reconfigurable Machine Scheduling Problem. arXiv:2109.11067 [cs.DC]

[61] VMware. 2020. *SHARING GPUS IN MACHINE LEARNING ENVIRONMENTS*. VMware. https://www.vmware.com/docs/vmware-ai-ml-rama

[62] Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax.

[63] Tianyu Wang, Sheng Li, Bingyao Li, Yue Dai, Ao Li, Geng Yuan, Yufei Ding, Youtao Zhang, and Xulong Tang. 2024. Improving GPU Multi-Tenancy Through Dynamic Multi-Instance GPU Reconfiguration. *arXiv preprint arXiv:2407.13126* (2024).

[64] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. 2023. Transparent GPU Sharing in Container Clouds for Deep Learning Workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 69–85. https://www.usenix.org/conference/nsdi23/presentation/wu

[65] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 30, 16 pages.

[66] Fei Xu, Jianian Xu, Jiabin Chen, Li Chen, Ruitao Shang, Zhi Zhou, and Fangming Liu. 2023. iGniter: Interference-Aware GPU Resource Provisioning for Predictable DNN Inference in the Cloud. *IEEE Transactions on Parallel and Distributed Systems* 34, 3 (2023), 812–827. doi:10.1109/TPDS.2022.3232715

[67] Ting-An Yeh, Hung-Hsin Chen, and Jerry Chou. 2020. KubeShare: A Framework to Manage GPUs as First-Class and Shared Resources in Container Cloud. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing* (Stockholm, Sweden) *(HPDC '20)*. Association for Computing Machinery, New York, NY, USA, 173–184. doi:10.1145/3369583.3392679

[68] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J Rossbach. 2020. AvA: Accelerated virtualization of accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 807–825.

[69] Yijia Zhang, Qiang Wang, Zhe Lin, Pengxiang Xu, and Bingqiang Wang. 2024. Improving GPU Energy Efficiency through an Application-transparent Frequency Scaling Policy with Performance Assurance. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 769–785. doi:10.1145/3627703.3629584

[70] Yongkang Zhang, Haoxuan Yu, Chenxia Han, Cheng Wang, Baotong Lu, Yang Li, Xiaowen Chu, and Huaicheng Li. 2024. Missile: Fine-Grained, Hardware-Level GPU Resource Isolation for Multi-Tenant DNN Inference. *arXiv preprint arXiv:2407.13996* (2024).

[71] Yongkang Zhang, Haoxuan Yu, Chenxia Han, Cheng Wang, Baotong Lu, Yunzhe Li, Zhifeng Jiang, Yang Li, Xiaowen Chu, and Huaicheng Li. 2025. SGDRC: Software-Defined Dynamic Resource Control for Concurrent DNN Inference on NVIDIA GPUs. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Las Vegas, NV, USA) *(PPoPP '25)*. Association for Computing Machinery, New York, NY, USA, 267–281. doi:10.1145/3710848.3710863

[72] Xia Zhao, Magnus Jahre, and Lieven Eeckhout. 2020. HSM: A Hybrid Slowdown Model for Multitasking GPUs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1371–1385. doi:10.1145/3373376.3378457

Mercury: Unlocking Multi-GPU Operator Optimization for LLMs via Remote Memory Scheduling

Yue Guan[1], Xinwei Qiang[1], Zaifeng Pan[1], Daniels Johnson[2], Yuanwei Fang[2], Keren Zhou[3,4], Yuke Wang[5], Wanlu Li[1], Yufei Ding[1,2], Adnan Aziz[2]
[1]University of California, San Diego, [2]Meta,
[3]George Mason University, [4]OpenAI, [5]Rice University
[1]{yueguan, x1qiang, zapan, wal019, yufeiding}@ucsd.edu
[2]{danielsjohnson, fywkevin, adnanaziz}@meta.com
[3]kzhou6@gmu.edu, [5]yuke.wang@rice.edu

Abstract

In this paper, we propose Mercury, a multi-GPU operator compiler based on a loop-based intermediate representation, CommIR. At the core of Mercury is an abstraction that treats remote GPU memory as an explicitly managed extension of the memory hierarchy, expanding the available storage and communication resources beyond local HBM. This unified view enables the compiler to reason holistically about data placement and inter-device communication, unlocking a vastly larger design space that encompasses and extends beyond existing manual strategies. As a result, Mercury is able to automatically reproduce the performance of hand-optimized baselines like RingAttention and Ulysses, and in some configurations, even discovers more effective strategies that manual designs have overlooked. Our implementation is open-sourced at https://github.com/ChandlerGuan/mercury_artifact.

ACM Reference Format:
Yue Guan[1], Xinwei Qiang[1], Zaifeng Pan[1], Daniels Johnson[2], Yuanwei Fang[2],, Keren Zhou[3,4], Yuke Wang[5], Wanlu Li[1], Yufei Ding[1,2], Adnan Aziz[2], [1]University of California, San Diego, [2]Meta,, [3]George Mason University, [4]OpenAI, [5]Rice University, [1]{yueguan, x1qiang, zapan, wal019, yufeiding}@ucsd.edu, [2]{danielsjohnson, fywkevin, adnanaziz}@meta.com, [3]kzhou6@gmu.edu, [5]yuke.wang@rice.edu, . 2025. Mercury: Unlocking Multi-GPU Operator Optimization for LLMs via Remote Memory Scheduling. In *ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3731569.3764798

SOSP '25, October 13–16, 2025, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1870-0/25/10
https://doi.org/10.1145/3731569.3764798

1 Introduction

As large language models (LLMs)[54] scale up in both model size and input sequence length, the compute and memory demands of individual operators, especially *attention*[48] and *general matrix multiplication (GEMM)*[32], have grown beyond the capacity of a single GPU[28]. Modern attention operators[17], with many heads and long contexts (e.g., 32K tokens), can require hundreds of gigabytes of memory; the KV cache alone for Llama-3 70B consumes 282GB, far exceeding the 80GB HBM of an NVIDIA H100 GPU[30]. Multi-GPU operator design is thus not only a performance optimization, but also a fundamental requirement for training and inferring large-scale models.

Optimizing multi-GPU operators for LLMs remains a highly manual and labor-intensive process. In the past two years alone, over twenty papers (e.g., [2, 6, 15, 18, 20, 27, 44, 51, 52]) have proposed different hand-tuned designs for just two operators, attention and linear, underscoring both the difficulty and importance of this problem. These manual optimizations are often tightly coupled to specific hardware and model configurations, with performance depending on factors such as GPU memory size, the number of GPUs, interconnect topology, and operator-specific parameters like head count and sequence length. This explodes the design space, making it difficult to port these optimizations to new settings, and the sheer number of possible configurations makes it infeasible to manually tune each one [10, 53]. On the other hand, with the advent of advanced hardware, like NVIDIA's B100 GPUs interconnected with NVLink72 [33], the hand-tuning methods become increasingly impractical.

This motivates the need for an automated and adaptive compiler for multi-GPU operators, one that not only reduces engineering effort but also unlocks a broader optimization space through proper abstractions, enabling the discovery of solutions that match or even outperform expert-tuned implementations across diverse hardware and workloads. Yet, existing multi-GPU compilers [55] remain insufficient for optimizing LLM operators. Academic compilers [7, 61] have yet to uncover a design space that encompasses recent hand-optimized multi-GPU operator designs [15], and

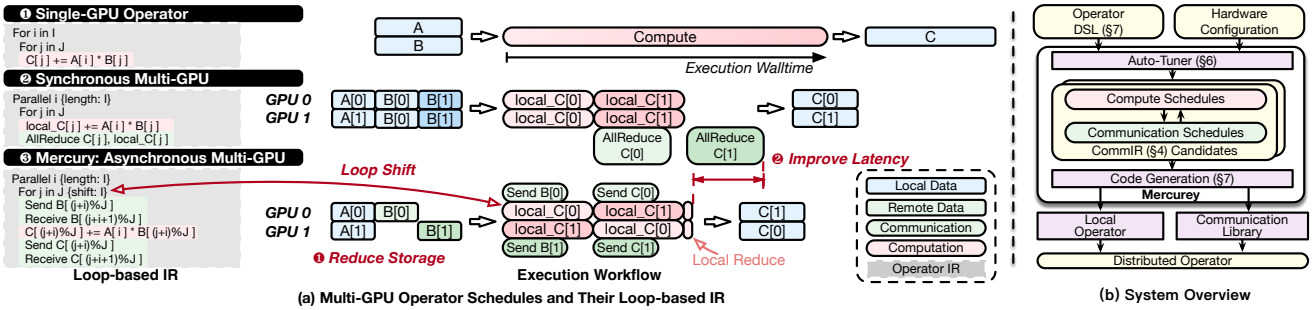


Figure 1. (a) Motivating example of multi-GPU operators with remote memory access. (b) Overview of Mercury.

are often unavailable or impractical to evaluate. Meanwhile, industrial systems like `torch.compile` [5] offer only simple multi-GPU support and consistently underperform compared to manual implementations [15, 18].

We observe that a fundamental reason for the performance gap in existing multi-GPU operator compilers lies in their restrictive assumption of a local-memory-centric execution model. That is, compilers assume that all input data must be fully available in each GPU's local memory before computation can proceed. As a result, inter-device communication is largely treated as a mechanism for exchanging intermediate results between operators. This assumption leads current compilers to default to execution models that fully duplicate shared inputs and enforce identical, temporally synchronized computation across devices—a pattern we refer to as the *synchronous schedule*. As illustrated in Fig. 1-❷, GPU0 computes over outer loop I on $A[0]$ and GPU1 over $A[1]$, but both follow the same inner loop structure over J. Shared input B is fully replicated across devices and accessed in a fixed order, e.g., $B[0]$ is used first, followed by $B[1]$, in exact synchrony across all GPUs. This duplication not only wastes valuable local HBM capacity but also hinders GPU optimizations such as deeper tiling. This restricts the compiler's ability to explore more flexible execution and communication patterns, particularly those that take advantage of remote memory as a shared data storage resource in the memory hierarchy, which may reduce the local memory footprint and reduce overall latency by enabling better compute-communication overlap.

To address this limitation, our approach is grounded in the insight that remote GPU memory can be treated as a first-class, schedulable layer in the memory hierarchy—on par with local HBM. In this abstraction, inter-device communication is used as a means to access shared input across devices rather than just exchanging intermediate results between kernels. This abstraction thus unlocks many new schedules. One representative example is the *shifted asynchronous schedule*, where devices stagger their access to shared data, allowing it to reside on the larger remote GPU memory pool aggregated by all GPUs and be transferred when needed by the local device. As shown in Fig. 1-❸, offsetting the inner J loop timeline by GPU I induces such a shifted computation and communication pattern. This temporal decoupling allows shared input data to be reused across devices, reducing local memory pressure and enabling other compiler optimizations (e.g., large tiling sizes). We note that this schedule with asynchronous temporal shift also forms the foundation of recent hand-optimized multi-GPU operators [27, 52], but prior implementations are rigid and not designed to integrate with compiler frameworks.

Translating such remote-memory-aware execution strategies into a general compiler infrastructure presents several key challenges. First, existing compiler abstractions lack a unified view of compute, memory, and communication, treating remote memory as an external mechanism rather than an integral part of the scheduling space, making it difficult to express structured patterns involving remote reuse or cross-device scheduling within the loop hierarchy. Second, not all remote memory accesses patterns are equivalent; those supported by collective primitives (e.g., *AllGather*, *ReduceScatter*) often outperform arbitrary P2P remote memory access patterns [45]. Thus, the compiler must not only express fine-grained memory sharing but also be capable of automatically generating efficient collective primitives when appropriate.

To address these challenges, we design Mercury, a compiler for optimizing multi-GPU operators. Fig. 1(b) shows an overview of the Mercury architecture. Mercury takes as input a tensor-level operator written in a Python-embedded domain-specific language (DSL) and generates an optimized execution plan across multiple GPUs, both within a single node and across multiple nodes. We remark that Mercury acts as a middle layer in the compiler stack: it connects to the higher-level computation graph optimization [61] to support global decisions such as operator fusion and intra-operator resharding, and to lower-level tensor compilers [8, 46] and libraries [35] for intra-GPU kernel optimization and code generation. This separation of concerns allows Mercury to focus exclusively on multi-GPU scheduling. Concretely, Mercury advances tensor compilers by introducing a loop-based IR that treats remote GPU memory as first-class and unifies

compute, memory, and communication within the schedule—enabling asynchronous (shifted) and collective patterns that explore this joint inter- and intra-GPU space to synthesize efficient collectives rather than relying on fixed templates. We will discuss the comparison with existing tensor compilers in Sec. 9.1.

At the core of Mercury is CommIR (§4), a loop-based intermediate representation (IR) that extends traditional loop-based IRs [16, 36]. CommIR introduces structured transformation primitives, `parallelize`, `shift`, `shard`, and `replicate`, which unify support for both standard intra-GPU tiling and advanced inter-GPU scheduling patterns, such as asynchronous shifts and collective communication primitives. These primitives express all known hand-optimized multi-GPU strategies and explore a significantly larger design space by allowing parallelism and shift transformations across arbitrary loop dimensions, as well as flexible hybridization of collective patterns that manual efforts have yet to explore (§5).

Mercury implements the CommIR and adopts an auto-tuning process for the optimal schedule (§6). Unlike prior systems that rely on hardcoded templates, Mercury lowers the CommIR candidates to local operator and communication kernels automatically to explore a larger design space (§7). Because the transformations are structured, Mercury can automatically synthesize communication plans, e.g., generating a ring-style pass by applying a `shift` over the loop dimension, without requiring custom kernel logic.

We evaluate Mercury across a range of LLM operators with varying context lengths, hardware platforms, and network configurations, demonstrating consistent performance improvement (§8). Our compiler outperforms state-of-the-art (SOTA) hand-optimized designs like USP[15] and Ulysses[20], averaging 1.56× speedup. Compared with model-level 3D-parallel [26], Mercury achieves up to 1.62× performance improvement for real LLM workloads.

In summary, this work makes the following contributions:

- We introduce CommIR, a novel loop-based IR that treats remote GPU memory as an explicit extension of the memory hierarchy and unifies computation, memory, and communication winthin a single scheduling abstraction.
- We build a modular compiler, Mercury, that automatically generates efficient multi-GPU operators through a set of communication-driven transformation passes.
- Our evaluation shows that Mercury outperforms state-of-the-art hand-tuned LLM libraries across diverse operators and hardware platforms.

2 Background

In this section, we first introduce the common settings for modern multi-node multi-GPU systems and their remote memory access interfaces. We then discuss representative multi-GPU operator designs with these interfaces.

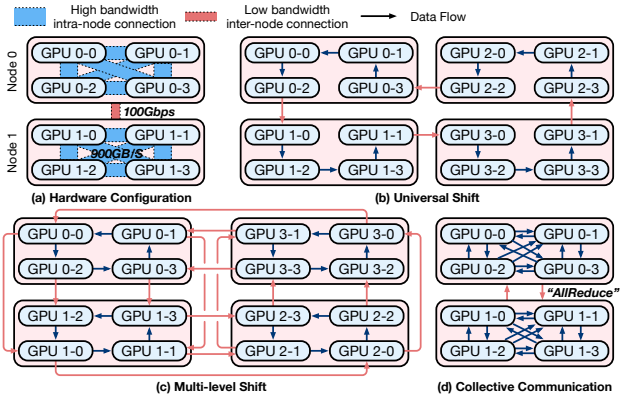


Figure 2. Multi-GPU interconnection and access patterns.

2.1 Multi-GPU Systems

To understand the design of multi-GPU operators, we first examine their interconnect topologies and access interfaces.

Multi-GPU Interconnection. To meet the massive requirement of modern LLM workloads, the systems are usually organized in a multi-node multi-GPU topology with different bandwidth as shown in Fig. 2-(a). Building on diverse interconnection hardware (such as Ethernet or PCIe), the communication bandwidth varies dramatically at different levels. For example, the intra-node bidirectional connection over NVLink mesh provides a 900 GB/s bandwidth, which is merely 3× slower than the local HBM bandwidth. The inter-node communication over high-specification RDMA over Converged Ethernet (RoCE) [49] or InfiniBand [42] delivers a much slower bandwidth, for example, 100 Gbps. Over this physical interconnection hardware, the remote memory access between GPU devices is delivered through point-to-point (P2P) or collective communication interfaces.

P2P Access. P2P communication provides fine-grained control of data transferring directly between GPU pairs, enabling flexible and overlapped communication patterns ideal for irregular or pipeline-parallel workloads. However, the P2P communication requires non-trivial scheduling across multiple GPU devices, making it difficult to schedule. For example, Fig. 2-(b) and (c) demonstrate two patterns that pass the intermediate results asynchronously. The multi-level shift pattern in (c) groups the intra- and inter-node connection together while the universal shift launches P2P communication universally. This flexibility makes it well-suited for optimizing bandwidth usage in hierarchical systems.

Collective Access. Modern vendor-provided libraries [1, 34], on the other hand, offer highly optimized implementations of collective remote memory access launched synchronously by a group of GPUs. These libraries deliver programming interfaces such as *AllReduce*, *Broadcast*, and *AllGather* as shown in Fig. 2-(d). The underlying implementation of these collective communication are aware of the

Projects	Parallel Dimension* Head	Query	Context	Collective	Schedule	Topology Adaptivity
Synchronous Operators						
Context Parallel [25]	○	◐	○	○	No	○
Ulysess [20]	◐	○	○	●	No	○
TreeAtten [44]	○	○	◐	●	No	●
Asynchronous Operators						
RingAtten [27]	○	●	○	○	No	○
USP [15]	◐	●	○	●	Template	●
LoongTrain [18]	◐	●	○	●	Template	●
Automatic Approaches						
Alpa [61]	◐	◐	◐	●	Template	●
Centauri [7]	◐	◐	◐	●	Template	○
CoCoNet [21]	◐	◐	◐	●	Auto	○
Mercury	●	●	●	●	Auto	●
	*◐ synchronous, ● asynchronous					

Table 1. Comparison of multi-GPU attention operators.

physical hierarchy and dynamically select optimized algorithms (e.g., ring, tree, or hybrid) based on the bandwidth and latency characteristics. For instance, NCCL exploits the NVLink mesh for high-throughput intra-node communication while using pipelined protocols over RoCE or InfiniBand for inter-node transfers. Collective communication provides a clean abstraction for inter-GPU coordination and ensures performance portability across a range of heterogeneous systems through low-level, hardware-aware scheduling.

2.2 Distributed Operators

With the aforementioned remote memory access interfaces, many parallelism strategies and supporting operators are studied. We will discuss the common operators in LLMs and their multi-GPU implementations in the following.

Operators in LLMs. Modern LLMs are fundamentally built upon attention mechanisms and linear layers. Multi-Head Attention (MHA) [48] enables models to capture diverse contextual relationships by attending to different parts of the input sequence. Variants like Multi-Query Attention (MQA) [41] and Grouped-Query Attention (GQA) [3] optimize inference efficiency and memory usage by sharing key and value (KV) activation across attention heads [3, 41]. The calculation of attention involves a four-level loop structure: the *batch*, *head*, *query*, and *context* dimensions. Another part is the linear layer, typically implemented as General Matrix-Matrix Multiplication (GEMM) [14] operations. Efficient distribution and execution of these operators across multiple GPUs are crucial for scaling LLM training and inference.

Synchronous Operators. With these well-defined collective communication libraries, many parallel strategies are proposed addressing the variety of operators, device configurations, and deployment scenarios. The basic asynchronous designs can be regarded as parallelizing the loop axis at a specific dimension. To elaborate, *data parallelism (DP)* [24, 37, 59] partitions input samples across devices, minimizing communication but duplicating model parameters, leading to high storage consumption, which is regarded as parallelizing at the batch dimension. As such, data parallelism does not require dedicated operators. *Tensor parallelism (TP)* [43] parallelizes the reduction dimension of the linear operator. This shards model parameters across devices, reducing storage but incurring significant communication overhead due to partial result reductions.

Due to the attention calculation's complex computation flow, its multi-GPU operator design raises a much larger design space, mainly determined by the parallel dimension as shown in Tbl. 1. *Context parallelism (CP)* [23] distributes workloads along the spatial query dimension but requires replication of large KV activations. This can be abstracted as parallelizing the query dimension of the attention operator. *Head parallelism* introduced by DeepSpeed-Ulysses [20] distributes the workloads at the attention head dimension. Similarly, TreeAtten [44] proposes to parallelize the reduction dimension of the attention operator with fine-grained collective communications. Mnemosyne [2] further extends this idea by combining it with other parallelism strategies.

Asynchronous Operators. Advanced distributed operators, in recent research, have introduced asynchronous patterns into the operators to reduce memory and improve communication efficiency. For attention operators, several research studies [18, 27, 50, 52] propose passing data among parallel workers to reduce the storage consumption with overlapped communication as shown in the upper part of Tbl. 1. RingAtten [27] proposes a universal shift pass of the sharded KV activation on top of the CP design. Yet this universal logical ring launches intra- and inter-node communication together, thus bottlenecked by the low-bandwidth inter-node communication. LoongTrain [18], TokenRing [52], and USP [15] propose multi-level shift patterns that separate the intra- and inter-node communication with a multi-level shift design for better overlapping, as shown in Fig. 2-(c).

Similarly, the shift pattern is also applied to the GEMM operators [6, 51] to overlap the communication at a finer granularity. These works focus on different operators and network settings, resulting in ad-hoc development efforts and difficulty generalizing to different configurations.

Automatic Approaches. While several template-based tuning frameworks or compilers support the generation and optimization of multi-GPU operators with synchronous communication patterns, no open-sourced distributed compiler provides general support for generating high-performance multi-GPU operators, especially in multi-node scenarios. Existing frameworks overlook the importance of using remote GPU memory as a source of input sharing, thus delivering sub-optimal performance. Alpa [61] proposes a communication-computation-communication paradigm to gather inputs and reduce outputs on parallel workers. The applicable parallel pattern is determined as a template for each operator. Centauri [7] further proposes partitioning

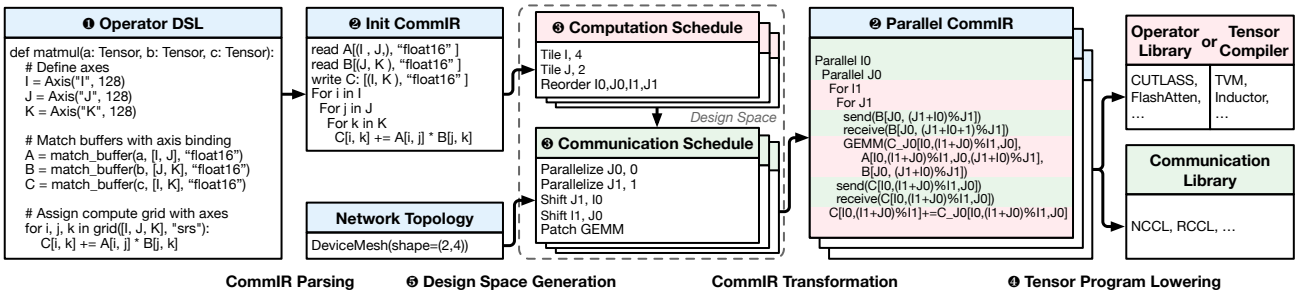


Figure 3. Overview of CommIR's workflow.

the communication and computation for fine-grained overlapping with pre-defined splitable axes on each operator. This enables more flexible fine-grained parallel strategies for compute-communication overlapping, yet still relies on template-based designs. Similarly, CoCoNet [21] defines a novel DSL with scheduling primitives to orchestrate collective communications with templated-free auto-tuning. However, its design space is restricted to synchronous communication patterns, leading to sub-optimal results compared to manual designs. None of these works manages to navigate the extensive design space that encompasses both synchronous and asynchronous communication patterns automatically, which is crucial for achieving optimal performance across diverse operator and hardware configurations.

3 Overview

To address the growing challenges of multi-GPU operator optimization, we introduce Mercury, a distributed operator compiler that unifies computation, communication, and memory management via a novel intermediate representation, CommIR. As illustrated in Fig. 3, Mercury systematically transforms a high-level operator specification into an efficient, distributed execution plan through four key stages: parsing, transformation, code generation, and tuning.

DSL. Mercury starts with a Python-like DSL that is simple, intuitive, and easy to adopt as shown in Fig. 3 ❶. It closely matches the syntax and structure of existing tensor DSLs [16, 56] to minimize the learning curve, while introducing a key distinction: the use of explicit loop symbols to expose iteration structure for lowering. This loop-based abstraction makes the DSL not only transformation-friendly but also expressive enough to capture tensor computations and their distribution semantics in a unified way. Users can directly specify parallelism levels, data shifts, replication, and other communication patterns via loop annotations, enabling seamless integration of computation and communication.

CommIR and Transformation Schedules. This DSL is parsed into CommIR (Fig. 3 ❷), a structured IR that preserves the hierarchy of loop nests and encodes distribution intent through a set of computation and communication primitives (Fig. 3 ❸). Computation primitives such as tile, reorder, and patch rewrite the loop structure, supporting rich scheduling transformations. Communication primitives such as parallelize, shard, and shift annotate loop variables and buffer layouts to indicate how data is partitioned, accessed, and exchanged across a device mesh. These primitives do not emit code directly but serve as symbolic annotations maintained through optimization and lowering.

Communication and Local Operator Lowering. Once a candidate schedule is selected, Mercury lowers it into backend-compatible code as shown in Fig. 3 ❹. The code generation process consists of two stages. First, communication kernels are synthesized by symbolically analyzing the loop index transformations and buffer annotations to determine P2P or collective communication patterns. This includes staggered sends/receives introduced by shift as well as collective communications. Second, the local computation kernels are lowered into device-specific IRs (e.g., TorchInductor), optionally patching regions with optimized libraries such as FlashAttention when applicable.

Auto-Tuner. To identify the most efficient distributed schedules, Mercury employs an auto-tuner that explores a structured design space generated from CommIR transformations as shown in Fig. 3 ❺. It first enumerates local computation schedules, then overlays communication strategies such as parallelize and shift, constrained by the target hardware mesh. Each candidate is profiled for latency, while memory usage is statically checked to prune infeasible options. This phased and constraint-aware search enables fast convergence to high-performance schedules.

4 CommIR

In this section, we draw the core abstraction of CommIR and how to use it to represent existing parallelism and beyond.

4.1 Definition and Primitives of CommIR

The insight of CommIR is that the loop-based IR in the tensor compilers for local operators already contains the semantics for parallel execution natively. For example, in the tensor compiler TVM[8], *bind* primitive assigns a loop to hardware constructs such as CUDA thread blocks or threads. Extending such a loop-based representation to a coarse-grained

Primitive	Demonstration	Definition	Example
CommIR	Defined by user	Weighted Sum	For i in I For j in J C[i]+=A[i,j]*B[j]
Computation Primitives			
Tile	Split loop and add buffer	Tile(J)	For i in I For j0 in J0 For j1 in J1 C_j0[i,j0]+=A[i,j0,j1]*B[j0,j1] C[i]+=C_j0[i,j0]
Join	Merge loops	Join(I,J)	For ij in IJ C[ij//j_len]+=C[ij//j_len,ij%j_len]
Reorder	Change the order of loops (for better locality)	Reorder(J,I)	For j in J For i in I C[i]+=A[i,j]*B[j]
Patch	Replace subgraph with micro-kernel by pattern matching	Patch(op)	For i in I op(C[i],A[i,:],B[:])
Communication Primitives			
Parallelize	Parallelize a loop at a network hierarchy level	Parallelize(I, 0)	Parallel i {length: I, mesh: 0} For j in J C[i]+=A[i,j]*B[j]
Shift	Shift a local loop according to a parallel loop	Shift(J,I)	Parallel i {length: I, mesh: 0} For j in J C[i]+=A[i,(j+i)%j]*B[(j+i)%j]
Shard	Shard a buffer to distributed buffer	Shard(B,I)	Parallel i {length: I, mesh: 0} AllGather(B) For j in J C[i]+=A[i,j]*B[j]
Replicate	Replicate a buffer among parallel ranks explicitly	Replicate(A,I)	Parallel i {length: I, mesh: 0} For j in J C[i]+=A[i,j]*B[j]

Table 2. Transformation primitives in CommIR.

distributed scenario is a natural fit. As such, we adopt a loop-based IR design inheriting previous tensor compilers and the computation-related transformations as introduced in Tbl. 2. We introduce four transformation primitives to introduce remote memory access semantics into the loop-based IR. The parallelize and shift are applied on the loop nodes to schedule the computation and the shard and replicate are applied on the buffer nodes to manage the memory. Tbl. 2 show minimal examples of the transformation effect on top of a weighted sum CommIR object in the first row.

• **Parallelize** distributes the iterations of a loop across parallel workers at a specified hierarchy level in the network (e.g., inter-node or intra-node). The second argument specifies the hierarchy order, and we restrict the loop length and hardware size to be equal.

• **Shift** offsets a local loop's index relative to a parallel loop, introducing asynchronous access patterns that stagger data access across ranks. This explicitly introduces remote memory access at different temporal steps. The first argument of shift identifies the target local temporal loop, and the second argument identifies the regarding parallel loop.

• **Shard** splits a buffer across workers, so each parallel rank owns a disjoint portion of the buffer. Note that a buffer can be sharded even on a loop that is not in its access index.

• **Replicate** duplicates a buffer across all workers participating in the parallel loop, so every rank has a full copy.

4.2 Remote Memory Access with CommIR

These communication primitives enable CommIR to represent a broad range of remote memory access patterns, as discussed in Sec. 2.1. Unlike computation transformations, communication primitives annotate the IR rather than modify it directly. These annotations are interpreted during the lowering phase to insert appropriate communication operations. This approach maintains a clean separation of concerns, facilitating joint computation and communication scheduling without premature commitment to a particular data layout or communication pattern.

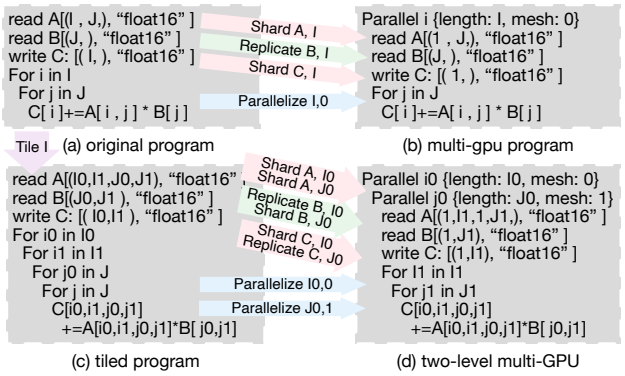


Figure 4. Workload partitioning with parallelize transformation at multiple topology levels.

Parallel Semantic with Parallelize. The parallelize primitive partitions a loop across devices and determines the workload distribution. Although memory placement and loop parallelization are conceptually independent, we apply a default initialization to avoid unnecessary remote memory accesses. The rationale is that buffers indexed by the parallelized loop are sharded across devices, and buffers not indexed by the parallel loop are replicated. This initialization ensures that each worker has local access to its required data. For example, in Fig. 4-(a),(b), buffers A and C are sharded, while B is replicated to all devices.

Furthermore, by explicitly specifying the network mesh hierarchy in parallelize, we expose hierarchical memory sharing. For instance, in Fig. 4-(c),(d), loops $I0$ and $J0$ are mapped to mesh levels 0 (inter-node) and 1 (intra-node), respectively. This results in buffer B being shared at the inter-node level and replicated at the intra-node level. Additionally, loop tiling and joining can combine axes from different dimensions, providing more flexibility in buffer sharing or replication within a hierarchy.

Asynchronous Access with Shift. The shift primitive introduces asynchrony by offsetting loop indices across workers. In Fig. 5, loop J is the local loop and I is the parallelized dimension. Shifting loop J by I causes each worker to access a different segment of the shared buffer B at staggered time steps, reducing contention. We automatically shard buffers associated with shifted loops to facilitate efficient communication. In the example, buffer B is replicated and A is locally sharded before the shift transformation. With the transformed access pattern: $(j+i)\%J$, B can also be sharded with staggered access from worker i.

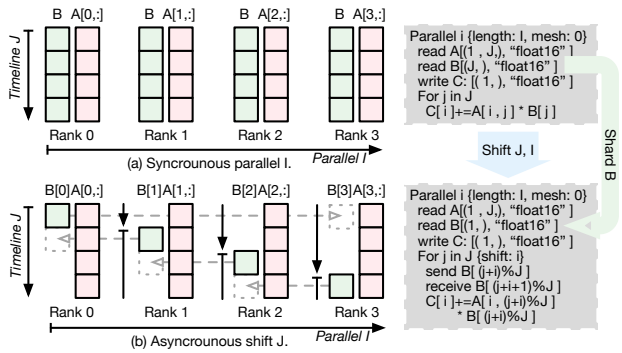


Figure 5. Asynchronous communication lowering with shift transformation.

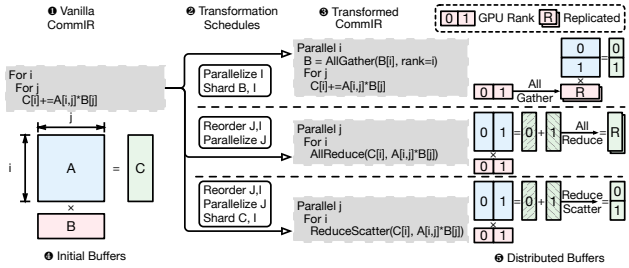


Figure 6. Representative collective communications synthesized from transformation schedules.

This shift-based design not only reduces storage by a factor of the number of devices but also overlaps computation and communication, as shown by the dashed arrows. Furthermore, multiple loop levels can be shifted independently, forming complex communication patterns that adapt to hierarchical hardware topologies (e.g., Fig. 2-(c)). The shift primitive can thus introduce asynchronous communications, resulting in schedules that are beyond the traditional synchronous worker model. We present hybrid shift patterns in Sec. 5.

Collective Access with Shard. Collective operations such as AllGather, Broadcast, and ReduceScatter are derived from buffer sharding patterns during the lowering phase. Fig. 6 illustrates three examples: (1) Read buffers: Sharding a read buffer triggers collective gathering, e.g., AllGather or Broadcast, depending on access and layout. (2) Write buffers: Sharded or replicated write buffers trigger reduce operations (e.g., AllReduce). (3) If the result buffer of a reduction is also sharded, the operation simplifies to ReduceScatter, reducing overhead.

The exact operation depends on the arithmetic semantics of the computation (e.g., sum, product) and the storage layout derived from the IR annotations, which is determined during the lowering phase via rule-based pattern matching over the CommIR. We incorporate predefined rules to insert collective communications whenever possible, otherwise fall back to point-to-point communication. While the lowering itself applies a deterministic rule set, the outer schedule-level search explores different computation and communication patterns to select the most performant configuration end-to-end.

5 CommIR's Expressiveness

With the well-defined CommIR, we can represent a wide range of parallelism strategies, capturing both established patterns and uncovering new ones. We illustrate this expressiveness using the attention operator [48], a core component of LLMs, as shown in Fig. 7. For simplicity, we reduce the attention operator to a scaled accumulation, mapping the query dimension to I and the context dimension to J, while omitting the batch and head dimensions, which follow a similar transformation process. The A represents the production of Q and K tensors before Softmax in attention (the P tensors). B represents the KV tensor. The vanilla local computation is shown in ❶, where buffer B (KV activation) is shared along the I axis and the output is reduced over J. By parallelizing the I axis, we naturally express context parallelism as in ❷. The shift primitive enables asynchronous communication patterns, aligning with prior designs like [27]. Further splitting of the I axis across device hierarchies allows hardware-aware communication planning, illustrated in Fig. 2-(c). Reordering and parallelizing the reduction loop yields patterns resembling TreeAttention [44], beneficial for decoding.

Beyond manually derived patterns, CommIR enables the discovery of novel strategies by automatically applying transformations along new axes and composing them. For example, shifting the reduction axis allows partial sums to be passed across workers in parallel, as shown in ❻. To further increase parallelism, the J axis can be split to enable partial reductions with multiple workers, parallelizing both I and outer J loops as in ❼. A more advanced composition appears in ❽, an actual searched result in our evaluation. Here, two workers collaborate on the same J0 reduction loop, with partial results shifted along I1, avoiding collective reduction. Additionally, J1 is shifted along I0, introducing intra-group shift communication for the shared B buffer. This complex design involves intricate scheduling of compute and communication, making it challenging to craft manually.

Besides the attention operator, CommIR can also support other operators. For instance, the TP method for linear layers [43] is expressible by splitting and parallelizing the GEMM reduction axis. The more advanced AsyncTP[51] strategy is captured by applying shift over an outer loop of size two. Altogether, CommIR offers broad expressiveness and generality across operators, enabling exploration of an expansive design space for distributed computation.

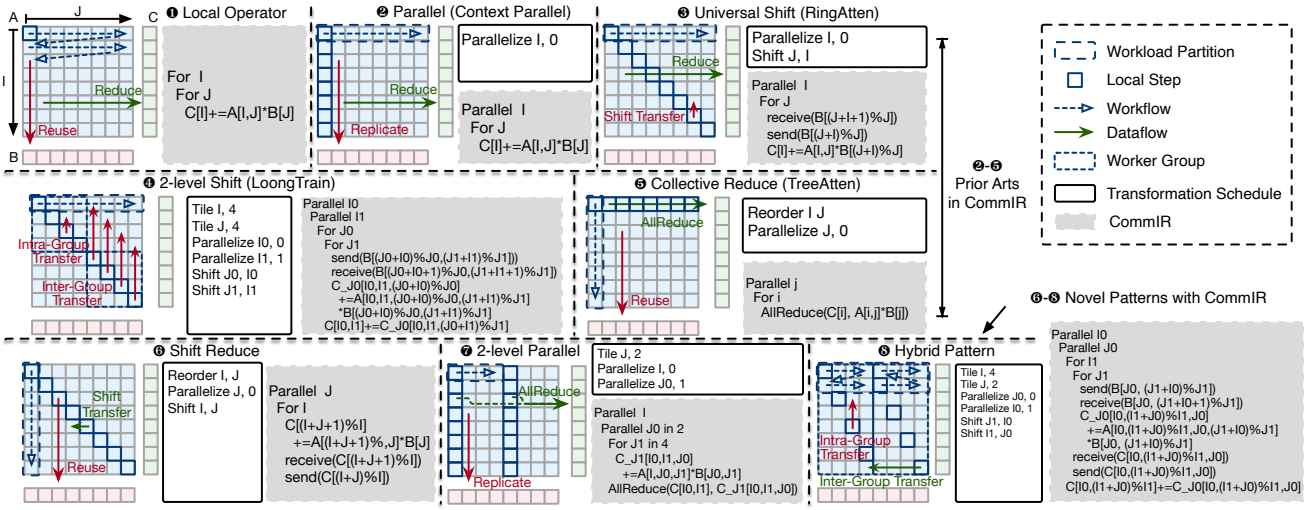


Figure 7. Attention operator examples and their CommIR expression. We approximate the attention operation as scaled accumulation in ❶ to simplify the illustration without any loss of generality.

6 Auto Tuner

With the expressive design of CommIR and a unified code generation pipeline, we build an auto-tuning system that searches for optimal distributed operator schedules by exploring a rich, transformation-driven design space.

Design Space Generation. We build the design space by enumerating the transformation primitives and sizes. In practice, we introduce several empirical rules to reduce the problem size without sacrificing expressiveness.

Firstly, We divide the generation process into two sequential schedules. (a) the computation schedule applies tiling, reordering, and join transformations to define the local loop structure and buffer layout. This schedule explores different ways to divide computation for next-step multi-GPU distribution. (b) the communication schedule applies communication primitives, such as parallelize, shift, shard, and replicate, to distribute computation and manage remote memory access. This phase determines how to parallelize the computation across devices and orchestrate inter-device communication. This phased approach retains the full expressiveness of the transformation space while reducing redundancy and avoiding invalid configurations. Candidates produced in the computation phase can be reused across multiple parallelization strategies.

Secondly, we explicitly incorporate hardware mesh configuration to regularize candidate generation. In particular, loops are only tiled according to mesh size (e.g., number of devices at each hierarchy level). Loops parallelized or shifted must have a length equal to the corresponding mesh dimension. Combined with reordering and loop merging, this constraint preserves coverage of the relevant schedules while significantly reducing the overall number of candidates.

Besides, we also bundle the parallelize and shift with corresponding shard or replicate operations. Although these can be decoupled in principle, shifting without sharding produces no semantic change. While this bundling could theoretically exclude some valid trade-offs (e.g., selectively sharding some shifted buffers), the evaluated workloads do not exhibit such patterns. Thus, this simplification reduces search complexity without sacrificing coverage in practice.

Search Objectives. The tuner aims to minimize end-to-end latency of the generated distributed operator, subject to memory constraints. Specifically:

- Latency evaluation: each candidate schedule is fully lowered and profiled in real hardware settings to obtain empirical runtime measurements.
- Memory constraint: we statically analyze each candidate's storage layout from its CommIR representation and compute the per-worker memory footprint. Candidates exceeding the provided capacity are pruned early in the search process.

With these empirical rules, the tuning completes within 10 minutes per operator in our evaluations. However, we acknowledge that search cost may grow with larger operator complexity or mesh topologies. Exploring more advanced tuning algorithms, e.g., cost-model-guided [22, 61] sampling or ML-based predictors, is a promising direction for future work.

7 Implementation

We introduce the implementation of the distributed compiler Mercury as shown in Fig. 1-(b).

7.1 DSL

To support the specification and transformation of distributed tensor operators, we design a new DSL that allows

concise and semantically rich operator definitions. We chose to build a new DSL instead of extending existing ones like TVM[8] or Triton[46], as they lack loop-level distribution semantics beyond local tensor programs. Our DSL extends PyTorch's frontend to leverage its ecosystem. PyTorch's DSL-to-IR path lacked loop-based distribution semantics, which we add via loop-centric DSL extensions. Integrating Mercury with other loop-based IRs (e.g., TVM/Triton) is feasible but requires engineering effort—most notably backend rewrites for communication code generation and multi-GPU scheduling—because these stacks primarily target single-GPU kernels. We will clarify these trade-offs in the revision. In our approach, loop transformations are first-class citizens across hierarchical device meshes, enabling unified local scheduling and global communication planning in the IR. The DSL is built around three key abstractions: axis declarations, buffer bindings, and computation grids, aligning with standard IR concepts including loop variables and storage objects.

• **Axis.** An Axis represents a named loop variable that defines a dimension of iteration. Each axis has a statically known extent and can be annotated with distribution metadata, such as its mapping to a hardware mesh level or a communication shift offset.

• **Buffer.** Buffers are declared using the *match_buffer* API, which binds a symbolic name to a tensor with a shape expressed over a set of axes. This declaration determines both storage layout and access semantics, allowing the compiler to infer the necessary communication based on axis transformations. The DSL tracks which axes are involved in parallelization or shifting, and associates the buffers with appropriate distribution primitives such as shard or replicate. We use the read and write annotations to mark the required data buffers directly for each local operator. As such, the annotated buffer is sharded or replicated according to the parallel axis. For example, if axis J is parallelized, then buffer B is interpreted as sharded across devices along that dimension.

• **Grid.** Computation is specified using the grid construct, which defines the iteration space of the operator over a set of axes and allows annotations for reduction semantics or fused loop scheduling. This construct corresponds to the loop nest in traditional IRs and forms the basis for tiling, reordering, and transformation during optimization. In the example shown in Fig. 3, "srs" refers to the loop types: "s" for spatial and "r" for reduction, which can influence communication patterns and data reuse strategies when compiled.

7.2 Schedule Primitives

Upon the CommIR, we implement the transformation primitives introduced in Sec. 4.1.

Computation primitives. Computation primitives in CommIR are implemented by rewriting the loop nest representation. Each transformation uses structured rewrite rules over the IR's loop tree. Internally, transformation passes traverse

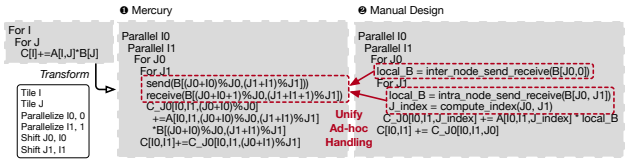


Figure 8. Comparison of code generated for the same two-level shift pattern from manual design and Mercury.

the loop nest using pattern matching, apply local rewrites, and update metadata tied to loop variables (e.g., iteration bounds, index expressions, and loop tags). These rewrites preserve referential transparency and are composable, enabling multi-pass scheduling without inconsistencies.

The Patch primitive is a special transformation designed to utilize the existing high-optimized local operator libraries, implemented as a subgraph substitution mechanism. Users can register pattern-to-kernel mapping rules, which describes the required loop shape, buffer access pattern, and optionally data types. During transformation, a graph matcher scans the IR for compatible subgraphs and replaces them with opaque external calls, wrapped with the necessary buffer bindings. This simplifies downstream lowering, especially when targeting well-optimized libraries like cuDNN[11] or FlashAttention[12, 13, 39].

Communication primitives. As introduced in Sec. 4.1, communication primitives are implemented as annotations on loop and buffer objects within the IR. They do not directly insert communication code; instead, they decorate the program with distribution intent, which is later materialized in the code generation stage. In summary, the communication primitives are annotated with the following metadata. Parallelize marks the loop variable with a target mesh level and records the mapping between loop iterations and device ranks. Shard and replicate update buffer metadata with a logical partitioning plan. Shift modifies the loop initialization and indexing logic to introduce staggered access across devices. This transformation is performed symbolically and maintained until lowering. In effect, communication is inferred based on the loop hierarchy and buffer sharing annotations.

7.3 Code Generation

The code generation pipeline in Mercury lowers the CommIR into an executable distributed program. This process is divided into two main stages: (1) generation of communication kernels and (2) lowering of local computation kernels.

Communication Kernel Generation. We begin by analyzing the annotated CommIR to infer the required inter-device communication patterns. P2P communications, introduced via the shift primitive, are derived firstly by statically analyzing the loop index transformations. The receiver and sender ranks are inferred using symbolic offset formulas (as

introduced in Sec. 4.1), enabling a concise and general formulation of staggered data flows. Collective communication operations are inferred for buffers annotated with shard or replicate. These are determined by analyzing buffer access patterns and the aggregation semantics of the surrounding loops.

This symbolic analysis decouples communication intent from implementation, allowing us to generate complex, topology-aware communication schedules automatically. As illustrated in Fig. 8, Mercury synthesizes a two-level shifted attention kernel using a unified representation of P2P communication. The result matches or exceeds the quality of hand-optimized kernels, while significantly reducing manual effort. Notably, the compiler is capable of generating sophisticated nested communication patterns that are challenging for human developers to design and verify manually.

Local Computation Lowering Once communication is resolved, we lower the computation portion of the IR to backend-specific code. The loop-based structure of CommIR allows for straightforward translation into existing tensor compilers or runtime libraries. In our implementation, TorchInductor is used as the primary backend for lowering and optimizing local computation. Because CommIR retains loop-level structure and buffer semantics, it integrates naturally with TorchInductor's operator representation and scheduling mechanisms. For compute-intensive regions, such as the innermost loops of attention kernels, we optionally invoke highly optimized operator libraries. Specifically, we use FlashAttention (FA) to patch local subgraphs where supported. These patches are inserted via the Patch transformation and treated as opaque external calls during lowering. FA patching is exposed as a tunable candidate in the search space to balance performance and generality.

This modular design enables joint optimization of computation and communication, accounting for both the device-level execution environment and the distributed topology. In addition, patching local computations with pre-optimized kernels significantly reduces the size of the design space, thereby accelerating the tuning process. Furthermore, this design makes Mercury extensible to support more complex scenarios, such ragged tensors or dynamic routing for mixture-of-expert (MoE) operators[57, 58], while Mercury currently focuses on static operators. This can be achieved by pre-compiling for common communication patterns or profiling-based variant selection at runtime. We see this as a valuable direction for future exploration.

7.4 Inter-Operator Resharding

To extend the benefits of operator-level tuning to the entire model, we incorporate graph-level reasoning by considering both the execution time of individual operators and the communication overhead required for resharding between them. This supports the synergy of the proposed operator-level tuning with model-level optimizations, such as Pipeline Parallelism (PP) [29]. In distributed settings, adjacent operators in a model often require different parallelization strategies, leading to incompatible sharding specifications on intermediate buffers. To resolve this mismatch, explicit resharding communication must be inserted, which contributes non-negligible latency beyond intra-operator communication.

Algorithm 1 Graph-level Search over Operator DAG

Input: Operator DAG $G = (O, E)$, where each O_i has candidate shardings $\{S_{i,1}, \cdots, S_{i,K_i}\}$
Output: Optimal sharding config $S^* = \{S_1^*, \cdots, S_N^*\}$ minimizing total cost

Initialize $S^* \leftarrow \emptyset$, $C_{\min} \leftarrow \infty$
for each full candidate $S = \{S_{1,j_1}, \cdots, S_{N,j_N}\}$ **do**
  $C \leftarrow 0$
  for each operator $O_i \in O$ **do**
    $C_{\text{exec}} \leftarrow \text{OpExecCost}(O_i, S_{i,j_i})$, $C_{\text{reshard}} \leftarrow 0$
    for each predecessor O_k of O_i in DAG **do**
      $C_{\text{reshard}} \leftarrow C_{\text{reshard}} + \text{ReshardCost}(S_{k,j_k}, S_{i,j_i})$
    end for
    $C \leftarrow C + C_{\text{exec}} + C_{\text{reshard}}$
  end for
  if $C < C_{\min}$ **then**
    $S^* \leftarrow S$, $C_{\min} \leftarrow C$
  end if
end for
return S^*

Our approach maintains a searchable database of distributed operator configurations, where each record includes the operator's input/output sharding settings and the corresponding execution time. Given a computation graph represented as a directed acyclic graph (DAG), we first compute the pairwise resharding cost for each edge connecting two dependent operators. This cost accounts for the communication required to transform the output sharding of a producer into the expected input sharding of its consumer.

To find the optimal combination of operator configurations, we perform a global search over the space of candidate assignments, aiming to minimize the total cost composed of per-operator execution latency and inter-operator resharding cost. To support seamless integration with our DSL, the input and output sharding constraints for each operator are defined using the Parallelize primitive in Mercury. Resharding communication routines are synthesized by interpreting mismatches in sharding specifications along DAG edges, following the procedure illustrated in Alg. 1. This design ensures that graph-level scheduling remains aware of both computation and communication costs, enabling more effective parallel strategy selection for the model.

8 Evaluation

This section introduces our evaluation settings and demonstrates the experimental findings. Our evaluation is structured into four parts: (1) operator benchmarks across hardware backends, (2) adaptability to network topologies, (3) scalability with increasing sequence lengths, and (4) design space analysis. This organization highlights the performance, portability, scalability, and expressiveness of Mercury.

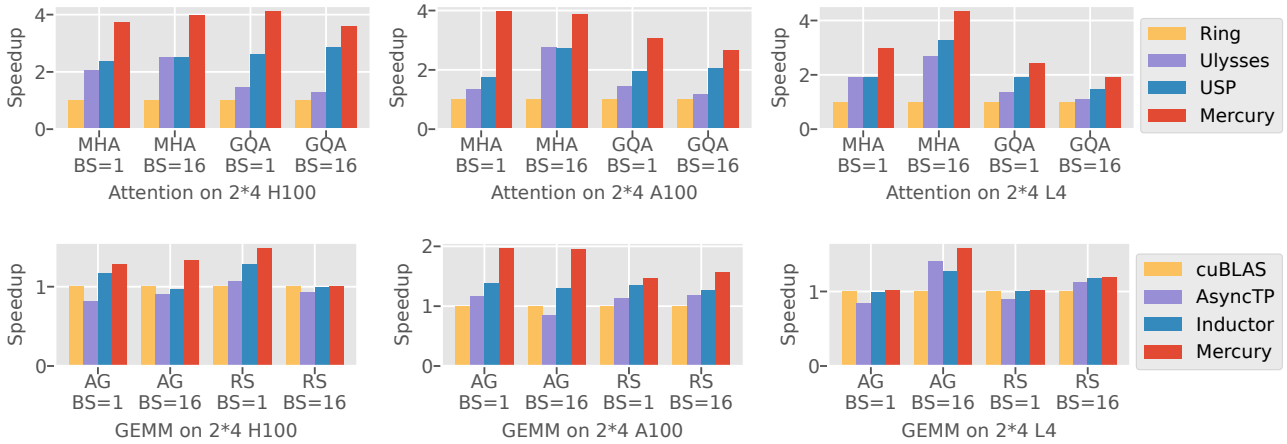


Figure 9. Multi-GPU operators performance benchmark from LLMs.

	L4	A100	H100
TFLOPs @FP16	242	312	1979
Memory (GB)	24	40	80
Memory Bandwidth (GB/s)	300	1555	3350
Intra-node Connection	PCIe	NVLink	NVLink
Intra-node Bandwidth (GB/s)	64	600	900
Inter-node Bandwidth (Gbps)	50	50	100

Table 3. Distributed hardware and network configurations.

8.1 Experimental Setup

Testbed. We deploy the proposed Mercury compiler with diverse GPU devices and interconnection settings to evaluate its generality. The configurations are summarized in Tbl. 3, showing the key varying factors. The nodes are connected with RoCE, and the intra-node connection is facilitated by NVLink[31] or PCIe. We adapt the number of nodes (ranging from 1 to 4) and GPU devices per node (ranging from 1 to 8) in each benchmark to evaluate the portability of Mercury with different settings, which will be elaborated accordingly. Mercury is implemented with CUDA-v12.6, NCCL-v2.26.2 and TorchInductor released with PyTorch-v2.8.

Workloads. We use Mercury to optimize the representative operators of modern LLMs, including different variants of attention (MHA, GQA). For the GEMM operator, we benchmark two settings, AllGather-GEMM (AG) and GEMM-ReduceScatter (RS), which typically show in the linear layers of LLMs with TP. The configuration of the evaluated operators is selected to match the model specification of the Llama-3 series[17], which stands for a typical setting for many LLM models. Besides, we also evaluate these operators under common batch size settings of 1 16 and scale the sequence length from 4K to 2M to show Mercury's generality.

Baselines. To evaluate the effectiveness of operators generated and tuned by Mercury, we compare with the SOTA manual written operators and auto-searching methods available. For the attention operator, we compare it with the asynchronous CP design RingAtten[27] (denoted as Ring) and synchronous HP design from DeepSpeed-Ulysses[20] (denoted as Ulysses), adopting collective communications. Furthermore, we also include an advanced template-based adaptive operator USP[15] using a hybrid communication pattern of CP and HP. Specifically, we report the best performance result in USP's design space, which will be elaborated in a case study. For the linear operator, we benchmark the synchronous operators with NCCL collective communication in cuBLAS[35]. We also benchmark a promising asynchronous design AsyncTP[51] as the SOTA manual efforts. Besides, we also compare with the operators generated by the TorchInductor[5] compiler, which tunes the best TP setting with a manual template of multi-GPU schedules.

8.2 Operator Benchmarks.

Operator Results. Across all benchmarks, Mercury consistently outperforms existing solutions, demonstrating both superior performance and broad generality. As shown in Fig. 9, Mercury achieves the highest speedup in every setting across both attention and GEMM workloads on multiple hardware backends (H100, A100, and L4), indicating its robustness and hardware portability.

In attention benchmarks, Mercury delivers significant speedups. For example, on H100, Mercury achieves up to 4× speedup on MHA with batch size 16, substantially outperforming USP and Ulysses. Notably, Ulysses excels only in the MHA setting due to its reliance on head-wise partitioning and fixed all-to-all communication, but suffers on GQA where the number of heads is reduced. USP, while introducing a combined CP+HP template, remains limited by a narrower design space. In contrast, Mercury automatically searches for optimized strategies tailored to operator characteristics and hardware, maintaining high performance even on GQA tasks. This highlights its adaptability to operator-specific constraints. In conclusion, Mercury demonstrates

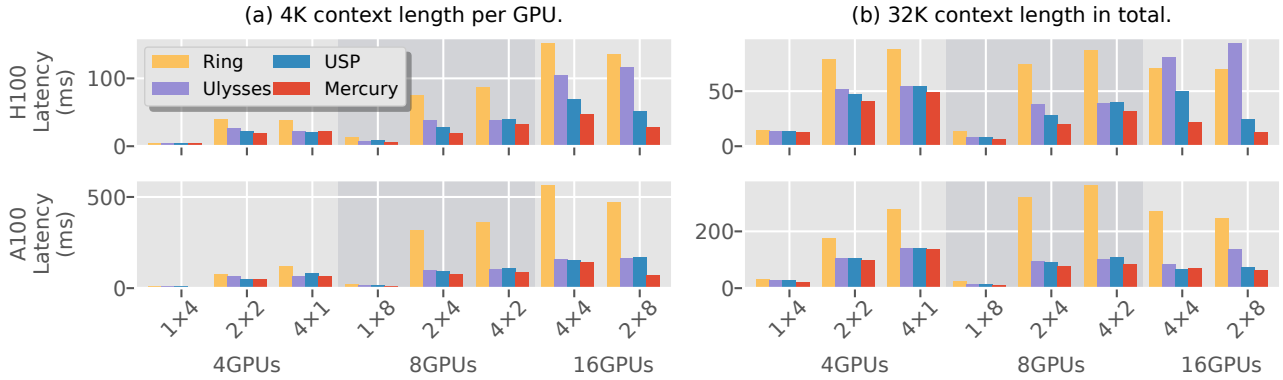


Figure 10. Attention operator benchmark with various hardware network configurations.

unmatched performance and flexibility across all attention configurations.

Mercury also shows strong results on GEMM benchmarks, which are generally considered more regular and communication-heavy. Although the performance gap is smaller compared to attention, Mercury still consistently surpasses both manually tuned baselines (e.g., AsyncTP) and compiler-generated ones (e.g., Inductor). For example, it achieves up to 1.9× speedup on AG with batch size 16 on A100, and maintains a stable lead across all devices. The improvement stems from Mercury's ability to break collective communication into finer-grained transactions and optimize overlapping computation and communication. This demonstrates its advantage even in conventional workloads with less inherent structural diversity. Therefore, Mercury's effectiveness on GEMM benchmarks further supports its general-purpose design.

Network Topology Adaptability. We evaluate Mercury across various multi-GPU configurations, shown in Fig. 10. The left plots fix the per-GPU workload (4K context), revealing scalability trends as the GPU count increases. The right plots fix total workload (32K context), highlighting communication efficiency under increasing parallelism. Across all settings on H100 and A100, Mercury consistently achieves the lowest latency, outperforming all baselines with an average 2.91× speedup. Unlike handcrafted strategies or fixed-template methods, Mercury adapts its communication and parallelism plan to each topology automatically.

In the 4K-per-GPU setting, intra-node topologies (e.g., 1×8) show better performance due to faster links, while inter-node layouts (e.g., 4×4) suffer from higher latency. Mercury maintains efficiency even in these bandwidth-heavy cases by avoiding excessive inter-node communication. In contrast, RingAtten becomes bandwidth-bound, and Ulysses and USP show inconsistent performance depending on how well their fixed strategies match the mesh.

In the 32K total context case, Mercury continues to outperform all baselines. However, in configurations with only a single-level hierarchy—such as purely intra-node (1×4) or inter-node (4×1) setups—the performance gap between Mercury and existing approaches narrows, as handcrafted strategies like USP and Ulysses are already well-optimized for such scenarios. This underscores Mercury's strength: its advantage becomes more pronounced in complex, hybrid topologies (e.g., 2×4, 4×2) where static methods struggle, and dynamic, topology-aware scheduling is crucial.

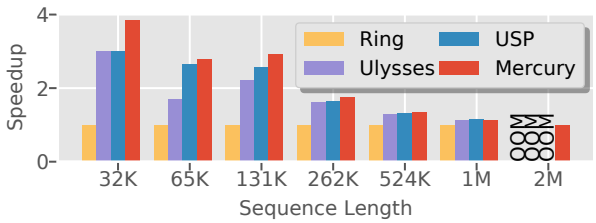


Figure 11. Attention operators scaling sequence lengths.

Context Scalability. We evaluate the scalability of Mercury by increasing the sequence length of the MHA operator under a 2×4 configuration on the H100 platform. As shown in Fig. 11, Mercury consistently outperforms all baselines as the sequence length scales from 32K up to 2M tokens.

When the sequence length exceeds 1 million, computation dominates the attention operator's runtime, diminishing the relative impact of communication. As a result, the speedup margin between Mercury and the baselines narrows. Nevertheless, Mercury still maintains competitive or superior performance in this regime. A key highlight is Mercury's ability to generate feasible execution plans even under extreme memory pressure. While other baselines fail with out-of-memory (OOM) errors at the 2M token mark, Mercury adapts by aggressively sharding KV caches and output tensors, trading increased communication for reduced memory usage. This flexibility enables Mercury to meet strict memory constraints without manual intervention. Another observation is the shift in relative performance between Ulysses and USP. Ulysses performs better for sequence lengths up to 65K,

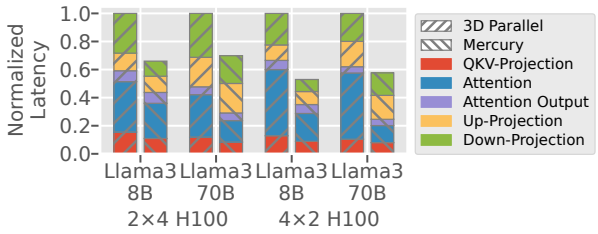


Figure 12. Model-level results of Llama3 series.

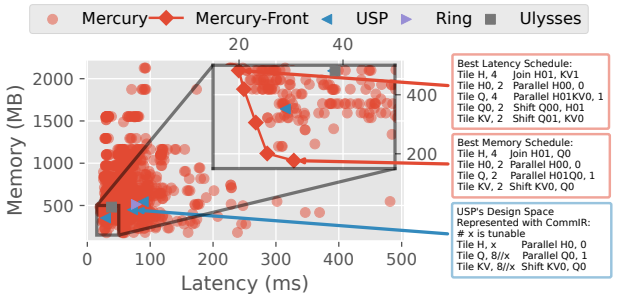


Figure 13. Design space analysis.

but USP overtakes it beyond that point. This behavior underscores the complexity of selecting optimal parallelization strategies and reinforces the value of Mercury's automatic search mechanism, which adapts to both workload and hardware constraints.

8.3 Model Benchmark

We evaluate Mercury's performance at the model level with operator configurations from Llama3-8B and Llama3-70 B. Specifically, we construct the computation graph using the graph-level search algorithm introduced in Sec. 7.4. We benchmark one Transformer layer with attention and linear operators since all layers share the same configuration. We compare to the 3D parallelism[26] strategy, combining DP, TP, and CP with the best configuration. In this benchmark, we set the sequence length to 4096 with batch size 1. The latency is broken down by key operators and resharding steps, including QKV Projection, Attention, and MLP layers.

Fig. 12 shows the normalized latency result of Llama3-8B and 70B models under two settings: 2×4 and 4×2. Across all configurations, Mercury achieves significantly lower latency than 3D Parallel. This improvement is not solely from optimizing individual operators but stems from a synergistic coordination at the model level. By jointly considering operator schedules and resharding decisions, Mercury eliminates redundant layout transformations and streamlines data flow across layers. The pronounced reduction in resharding overhead highlights Mercury's capability to treat the model as a unified computational graph, where layout choices are optimized globally, rather than in isolation. This global view enables more efficient execution pipelines, revealing the advantage of integrating operator-level compilation with model-wide communication planning.

8.4 Design Space Analysis

To better understand the expressiveness and tunability of Mercury, we visualize its design space on an MHA operator using a 2×4 H100 configuration, as shown in Fig. 13. Each red dot represents a candidate schedule evaluated during Mercury's auto-tuning process, plotted in terms of latency and memory consumption. We also include the results of baseline methods, USP, Ring, and Ulysses, for comparison. To highlight the most practical region, we zoom into the lower-latency, lower-memory corner and outline the Pareto front. Mercury's large and expressive search space, enabled by the proposed CommIR, encompasses both existing strategies and novel schedules that are not easily reachable through manual design. This allows Mercury to adapt to diverse operators and hardware topologies.

The best-latency candidate (marked in the zoom-in) features a hybrid parallelism strategy: it applies HP with a degree of 4 across both intra-node and inter-node levels (with Shift Q00 H01 and Shift Q01 KV0), combined with a shifted CP-2 on the intra-node level. Additionally, each local operator is shifted along the reduction dimension, enabling fine-grained overlap between computation and communication. In contrast, the best-memory candidate uses intra-node parallelism over the context dimension and reuses the local Q dimension with a shift transformation. This significantly reduces peak memory usage by efficiently shifting KV activation and output tensors, as demonstrated in Fig. 7.

For comparison, we also represent USP's manually crafted design space using CommIR's abstraction. As shown in the blue box, USP only explores a narrow subregion of the full design space, with limited tunability over key axes like tiling granularity or parallelism layout. This further illustrates the necessity of automated exploration in achieving both optimal performance and memory efficiency across diverse settings.

9 Related Work

9.1 Tensor Compilers

Scheduling DSLs, such as Halide [36], pioneered the separation of algorithm from schedule and enabled powerful loop transformations. Early systems emphasized explicit schedules and locality-aware tiling on CPUs/GPUs, but largely assumed that all data resided in local memory and that communication was outside the scheduling model.

Subsequent ML-oriented tensor compilers[47] introduced richer tensor semantics and annotations, easing operator authoring while relying on template- or cost-model-guided autotuning to search intra-GPU schedules. AutoTVM[9], Ansor[60] and MetaSchedule[40], which are distinct tuning

systems built on top of TVM[8], further automated schedule generation for single devices, extending portability across accelerators and operators.

As model sizes grew, tensor compilers incorporated basic multi-GPU support[4, 21, 38, 50], typically by composing single-GPU kernels with predefined collective primitives. These approaches improved usability but continued to treat inter-device data movement as an external mechanism, limiting opportunities for remote-memory reuse, asynchronous sharing, and topology-aware schedule synthesis.

Recent tensor compiler systems trends reinforce the centrality of LLM operators by extending the auto-tuning search space with semantic algebra transformations[53] or fine-grained intra-kernel pipelining[10]. Mercury retains the algorithm/schedule separation concept but advances it for multi-GPU by treating remote GPU memory as first-class and unifying compute, memory, and communication with explicit primitives.

9.2 Fused Distributed Operators

Beyond the manual design of distributed operators discussed in Sec. 2, recent systems have explored fused designs that tightly integrate communication and computation to maximize overlap and efficiency. The concept of expressing asynchronous remote memory access with Shift primitive in Mercury can be applied to these fused designs as well. However, the code generation and autotuning of such fused kernels remain open challenges and are left for future work.

Flux[6] fuses GEMM with collectives at tile granularity and over-decomposes work to maximize compute-communication overlap for both training and inference. Comet[57] targets MoE, using a shared-tensor abstraction and NVSHMEM-backed buffers to overlap token-wise communication with tile-wise compute at production scale. Triton-Distributed[62] adds OpenSHMEM-style one-sided primitives and signal-based coordination to Triton, enabling Python-authored overlapped kernels (e.g., AllGather/GEMM, GEMM/ReduceScatter) on NVIDIA and AMD GPUs. TileLink[63] introduces tile-centric primitives that link compute tiles to communication so the compiler can automatically generate overlapped kernels while decoupling compute and communication choices. FlashOverlap[19] uses lightweight readiness signaling and layout reordering to trigger overlap with standard NCCL collectives while leaving compute kernels unchanged.

10 Conclusion

We present Mercury, an automated compiler framework for multi-GPU tensor programs, built on a novel loop-based IR, CommIR. By combining a custom DSL with advanced scheduling and communication primitives, Mercury jointly optimizes computation and communication, discovering novel parallel strategies that outperform state-of-the-art on attention and GEMM operators. It simplifies complex multi-GPU operator design and adapts to diverse hardware, enabling scalable, efficient execution for large-scale models and opening paths for future tuning, graph-level integration, and heterogeneous device support.

Acknowledgments

We thank the anonymous reviewers for their thoughtful feedback, which helped improve this work. We are also grateful to our shepherd, Zhihao Jia, for guidance during the revision process. This work was supported in part by NSF awards 2124039 and 2411134.

References

[1] Advanced Micro Devices, Inc. *RCCL: ROCm Communication Collectives Library*, 2025. Version 2.23.4.

[2] Amey Agrawal, Junda Chen, Íñigo Goiri, Ramachandran Ramjee, Chaojie Zhang, Alexey Tumanov, and Esha Choukse. Mnemosyne: Parallelization strategies for efficiently serving multi-million context length llm inference requests without approximations. *arXiv preprint arXiv:2409.17264*, 2024.

[3] Joshua Ainslie, James Lee-Thorp, Xuezhi Wu, Adam Roberts, Sharan Narang, Hongkun Zhou, Zihang Wang, Jaehoon Lee, Maarten Bosma, and Yi Chen. Grouped-query attention. *arXiv preprint arXiv:2305.13245*, 2023.

[4] Sami Alabed, Daniel Belov, Bart Chrzaszcz, Juliana Franco, Dominik Grewe, Dougal Maclaurin, James Molloy, Tom Natan, Tamara Norman, Xiaoyue Pan, et al. Partir: Composing spmd partitioning strategies for machine learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 794–810, 2025.

[5] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 929–947, New York, NY, USA, 2024. Association for Computing Machinery.

[6] Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, Xuanrun Zhang, Zuquan Song, Chengji Yao, Ziheng Jiang, et al. Flux: fast software-based communication overlap on gpus through kernel fusion. *arXiv preprint arXiv:2406.06858*, 2024.

[7] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 178–191, 2024.

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler

for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[9] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 3393–3404, Red Hook, NY, USA, 2018. Curran Associates Inc.

[10] Yu Cheng, Lei Wang, Yining Shi, Yuqing Xia, Lingxiao Ma, Jilong Xue, Yang Wang, Zhiwen Mo, Feiyang Chen, Fan Yang, Mao Yang, and Zhi Yang. Pipethreader: Software-defined pipelining for efficient dnn execution. In *OSDI*. USENIX, July 2025.

[11] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[12] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[13] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[14] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, March 1990.

[15] Jiarui Fang and Shangchun Zhao. A unified sequence parallelism approach for long context generative ai. *arXiv preprint arXiv:2405.07719*, 2024.

[16] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 804–817, 2023.

[17] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[18] Diandian Gu, Peng Sun, Qinghao Hu, Ting Huang, Xun Chen, Yingtong Xiong, Guoteng Wang, Qiaoling Chen, Shangchun Zhao, Jiarui Fang, et al. Loongtrain: Efficient training of long-sequence llms with head-context parallelism. *arXiv preprint arXiv:2406.18485*, 2024.

[19] Ke Hong, Xiuhong Li, Minxu Liu, Qiuli Mao, Tianqi Wu, Zixiao Huang, Lufang Chen, Zhong Wang, Yichong Zhang, Zhenhua Zhu, et al. Flashoverlap: A lightweight design for efficiently overlapping communication and computation. *arXiv preprint arXiv:2504.19519*, 2025.

[20] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. *arXiv preprint arXiv:2309.14509*, 2023.

[21] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 402–416, 2022.

[22] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[23] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5:341–353, 2023.

[24] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12):3005–3018, August 2020.

[25] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2391–2404, 2023.

[26] Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, et al. Torchtitan: One-stop pytorch native solution for production ready llm pre-training. *arXiv preprint arXiv:2410.06511*, 2024.

[27] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ringattention with blockwise transformers for near-infinite context. In *The Twelfth International Conference on Learning Representations*.

[28] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. Benchmarking and dissecting the nvidia hopper gpu architecture. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 656–667. IEEE, 2024.

[29] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, 2019.

[30] NVIDIA. NVIDIA H100 Tensor Core GPU Architecture. Technical report, NVIDIA, mar 2022. White paper.

[31] NVIDIA Corporation. Nvidia nvlink high-speed interconnect: Application performance. Technical report, NVIDIA Corporation, 2015. Accessed: 2025-04-16.

[32] NVIDIA Corporation. *cuBLAS Library*, 2023. Retrieved from https://docs.nvidia.com/cuda/cublas/.

[33] NVIDIA Corporation. Nvidia b100 blackwell gpu. https://www.cudocompute.com/blog/nvidias-blackwell-architecture-breaking-down-the-b100-b200-and-gb200, 2024. Accessed: 2025-04-17.

[34] NVIDIA Corporation. *NVIDIA Collective Communications Library (NCCL)*, 2025. Version 2.26.2.

[35] NVIDIA Corporation. *NVIDIA cuBLAS Library*, 2025. Version 12.8.

[36] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, June 2013.

[37] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[38] Keshav Santhanam, Siddharth Krishna, Ryota Tomioka, Tim Harris, and Matei Zaharia. Distir: An intermediate representation and simulator for efficient neural network distribution. *arXiv preprint arXiv:2111.05426*, 2021.

[39] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems*, 37:68658–68685, 2024.

[40] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. Tensor program optimization with probabilistic programs. *Advances in Neural Information Processing Systems*, 35:35783–35796, 2022.

[41] Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

[42] Galen M. Shipman, Tim S. Woodall, Rich L. Graham, Arthur B. Maccabe, and Patrick G. Bridges. Infiniband scalability in open mpi. In

Proceedings of the 20th International Conference on Parallel and Distributed Processing, IPDPS'06, page 100, USA, 2006. IEEE Computer Society.

[43] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[44] Vasudev Shyam, Jonathan Pilault, Emily Shepperd, Quentin Anthony, and Beren Millidge. Tree attention: Topology-aware decoding for long-context attention on gpu clusters. *arXiv preprint arXiv:2408.04093*, 2024.

[45] Muhammet Abdullah Soytürk, Palwisha Akhtar, Erhan Tezcan, and Didem Unat. Monitoring collective communication among gpus. In *European Conference on Parallel Processing*, pages 41–52. Springer, 2021.

[46] Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 10–19, New York, NY, USA, 2019. Association for Computing Machinery.

[47] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[49] Jerome Vienne, Jitong Chen, Md. Wasi-Ur-Rahman, Nusrat S. Islam, Hari Subramoni, and Dhabaleswar K. Panda. Performance analysis and evaluation of infiniband fdr and 40gige roce on hpc and cloud computing systems. In *2012 IEEE 20th Annual Symposium on High-Performance Interconnects*, pages 48–55, 2012.

[50] Haoran Wang, Lei Wang, Haobo Xu, Ying Wang, Yuming Li, and Yinhe Han. Primepar: Efficient spatial-temporal tensor partitioning for large transformer model training. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 801–817, New York, NY, USA, 2024. Association for Computing Machinery.

[51] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, et al. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 93–106, 2022.

[52] Zongwu Wang, Fangxin Liu, Mingshuai Li, and Li Jiang. Tokenring: An efficient parallelism framework for infinite-context llms via bidirectional communication. *arXiv preprint arXiv:2412.20501*, 2024.

[53] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Man Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. Mirage: A {Multi-Level} superoptimizer for tensor programs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 21–38, 2025.

[54] Tongtong Wu, Linhao Luo, Yuan-Fang Li, Shirui Pan, Thuy-Trang Vu, and Gholamreza Haffari. Continual learning for large language models: A survey. *CoRR*, 2024.

[55] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. DISTAL: the distributed tensor algebra compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 286–300, San Diego CA USA, June 2022. ACM.

[56] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. Sparsetir: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 660–678, New York, NY, USA, 2023. Association for Computing Machinery.

[57] Shulai Zhang, Ningxin Zheng, Haibin Lin, Ziheng Jiang, Wenlei Bao, Chengquan Jiang, Qi Hou, Weihao Cui, Size Zheng, Li-Wen Chang, et al. Comet: Fine-grained computation-communication overlapping for mixture-of-experts. *arXiv preprint arXiv:2502.19811*, 2025.

[58] Chenggang Zhao, Shangyan Zhou, Liyue Zhang, Chengqi Deng, Zhean Xu, Yuxuan Liu, Kuai Yu, Jiashi Li, and Liang Zhao. Deepep: an efficient expert-parallel communication library. https://github.com/deepseek-ai/DeepEP, 2025.

[59] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16(12):3848–3860, August 2023.

[60] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[61] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[62] Size Zheng, Wenlei Bao, Qi Hou, Xuegui Zheng, Jin Fang, Chenhui Huang, Tianqi Li, Haojie Duanmu, Renze Chen, Ruifan Xu, et al. Triton-distributed: Programming overlapping kernels on distributed ai systems with the triton compiler. *arXiv preprint arXiv:2504.19442*, 2025.

[63] Size Zheng, Jin Fang, Xuegui Zheng, Qi Hou, Wenlei Bao, Ningxin Zheng, Ziheng Jiang, Dongyang Wang, Jianxi Ye, Haibin Lin, et al. Tilelink: Generating efficient compute-communication overlapping kernels using tile-centric primitives. *arXiv preprint arXiv:2503.20313*, 2025.

Managing Scalable Direct Storage Accesses for GPUs with GoFS

Shaobo Li*
shaobol2@illinois.edu
University of Illinois
Urbana-Champaign, USA

Yirui Eric Zhou*
yiruiz2@illinois.edu
University of Illinois
Urbana-Champaign, USA

Yuqi Xue
yuqixue2@illinois.edu
University of Illinois
Urbana-Champaign, USA

Yuan Xu
yuanxu4@illinois.edu
University of Illinois
Urbana-Champaign, USA

Jian Huang
jianh@illinois.edu
University of Illinois
Urbana-Champaign, USA

Abstract

As we shift from CPU-centric computing to GPU-accelerated computing for supporting intelligent data processing at scale, the storage bottleneck has been exacerbated. To bypass the host CPU and alleviate unnecessary data movements, modern GPUs enable direct storage access to SSDs (i.e., GPUDirect Storage). However, current GPUDirect Storage solutions still rely on the host file system to manage the storage device, direct storage accesses are still bottlenecked by the host.

In this paper, we develop a **G**PU-**o**rchestrated **f**ile **s**ystem (GoFS) for scaling the direct storage accesses for GPU programs, by fully offloading the storage management to the GPU. As GoFS provides POSIX API and manages core filesystem structures in GPU memory, it can execute both control path and data path without host CPU involvement. To enable highly concurrent direct storage accesses, we rethink the design and implementation of core filesystem structures with various optimization techniques, such as scalable data indexing, fine-grained per-SM (streaming multiprocessor) block management, and zero-copy I/O accesses, by carefully exploring the GPU-accelerated computing paradigm. GoFS preserves the essential filesystem properties such as crash consistency, and it is compatible with existing host-based file systems like F2FS. GoFS does not require changes to the on-disk filesystem organization, therefore, the host and GPU can manage the SSD in a coordinated fashion, and maintain the data consistency in a primary/secondary mode. We implement GoFS based on F2FS using 7.9K lines of codes with CUDA programming. We examine its efficiency on an A100 GPU. Our experiments with various GPU-based applications show that GoFS outperforms state-of-the-art storage access solutions for GPUs by 1.61× on average.

*Co-primary authors.

SOSP '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1870-0/2025/10
https://doi.org/10.1145/3731569.3764857

CCS Concepts: • **Software and its engineering → File systems management**; **Massively parallel systems**.

Keywords: GPU, GPUDirect Storage, File System

ACM Reference Format:
Shaobo Li, Yirui Eric Zhou, Yuqi Xue, Yuan Xu, and Jian Huang. 2025. Managing Scalable Direct Storage Accesses for GPUs with GoFS. In *ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3731569.3764857

1 Introduction

As we utilize GPU-accelerated computing for intelligent data processing of large-scale datasets [11, 22, 26, 28, 29, 45, 57, 63], the storage bottleneck is exacerbated. For instance, prior study of using GPUs to execute intelligent queries [43] and graph analytics [58] against datasets on solid-state drives (SSDs) showed that the storage I/O took a significant amount of the total execution time. This is because they still rely on the host CPU to access data on the SSDs, although the data loading via storage I/O has been pipelined with the data transfers between the host memory and GPU memory.

To bypass the host CPU and alleviate extra data movements, hardware accelerators like GPUs support peer-to-peer (P2P) communication with storage devices [4, 21, 58, 76]. A typical example is NVIDIA's GPUDirect Storage (GDS) [21], which programs the GPU memory to the Base Address Register (BAR) of the PCIe. With PCIe memory-mapped I/O (MMIO), the memory requests can be directly issued to the PCIe endpoint devices like SSDs. GDS offers a promising approach for enabling the GPU to directly communicate with SSDs without host CPU involvements. However, since the SSD is still managed by the host file systems, existing solutions like NVIDIA's cuFile [56] and GPUfs [62] have to rely on the host CPU to manage the control path and even the data path when interacting with the SSD, resulting in suboptimal performance.

To bypass the host CPU, BaM [58] allows GPU to initiate direct access to the raw disk at the sacrifice of dropping existing storage software stack, and enforces applications to manage the storage devices on their own, which inevitably

introduces burden to application developers. As we enable GPUs to directly access storage for high performance, we still need generic and scalable software to manage the storage device. Similar to any other types of persistent storage device, the file system is still the best fit for managing GPUDirect storage, because of its well-developed properties of managing persistent data blocks. However, simply relying on the host for GPUDirect storage management causes severe performance overhead and scalability issues (see our evaluation in §4).

Most recently, GeminiFS [57] developed an application-specific solution by preloading file metadata from the host file system into GPU memory to accelerate file metadata accesses, with the assumption that GPU applications have predictable storage I/O access patterns and most of their on-disk data is read-only. However, this approach is not always effective, as many GPU-accelerated applications such as intelligent queries [22, 24, 42, 72], generic graph computing [6], graph neural network (GNN) training [23, 33], and LLM-based retrieval-augmented generation (RAG) [19, 38], do not always generate predictable storage accesses, and their file metadata could be significant large (tens of GBs), especially in comparison with precious GPU memory. And they may incur new writes (e.g., intermediate data) during their workload execution. Therefore, to support diverse GPU-accelerated applications, it is highly desirable to develop a generic solution for managing scalable direct storage accesses for GPUs.

To this end, we develop GoFS, a GPU-orchestrated file system with direct storage access. GoFS provides POSIX APIs and manages core filesystem structures (e.g., inode/dentry, block management, and data pointers) in GPU memory, such that we can execute both control path and data path on the GPU. It enables on-demand direct access to on-disk metadata and data, avoiding unnecessary metadata/data caching in GPU memory. GoFS is compatible with existing host file systems like F2FS. It does not require changes to the on-disk file organization. The host and GPU can manage the SSD in a coordinated fashion and maintain data consistency in a primary/secondary mode.

Developing GoFS is not easy. This is for three major reasons. First, unlike conventional multi-core CPU architecture, modern GPUs offer massive parallelism with more than a hundred SMs (Figure 1). We have to explicitly consider the hierarchy of GPU threads in the design for alleviating the scalability bottlenecks in core filesystem data structures. Second, following the single-instruction-multiple-data (SIMD) computing paradigm, the programming model on the GPU is unique, which requires explicit parallelization in the program. Third, there are limited utility and helper libraries on GPUs for developing system software. Given these reasons, the drop-in replacement approach for migrating existing scalability optimizations in host-based file systems [8, 13, 74] cannot be directly employed in the development of GoFS.

In GoFS, we rethink the design and implementation of core filesystem structures for GPUs, and develop various optimization techniques for scaling their concurrent accesses. They involve both metadata (§3.3) and data I/O management (§3.4).

Scalable metadata management. We scale the inode/dentry accesses by replacing the conventional mutex lock with a range lock customized for GPUs to alleviate the high contention between storage access requests. We parallelize range checks with GPU's warp-level reduction primitives to achieve better performance (from $O(log(n))$ to near-$O(1)$) than conventional range lock management in conjunction with interval trees [34]. GoFS also introduces the batched inode (*bnode*) to track a collection of opened files under the same directory for enabling batched file operations. This greatly reduces metadata management overhead by coalescing metadata operations for thousands of GPU cores. To scale the block allocation on GPUs, GoFS develops per-SM bitmaps to track occupied and free blocks, and coalesces the block allocations of all threads of a thread block into a single allocation. This eliminates the need to track per-core structures [40] proposed for scaling CPU-centric file systems, alleviates the allocation contention, and minimizes the metadata management overheads.

Similar to existing file systems, GoFS uses a tree-like structure to index data blocks in files (i.e., data pointers), but it develops a *level-synchronous* parallelism scheme that consists of a two-stage (metadata and data) procedure to scale the traversal of data pointers. At the metadata traversal stage, all nodes (data pointers) at the same level in the tree will be processed in parallel to find the nodes at the next level. After that, at the data traversal stage, GoFS fetches all data blocks in parallel to best utilize the SSD bandwidth.

Scalable data I/O management. GoFS maintains multiple NVMe queues in the GPU memory to serve parallel I/O requests from multiple SMs. It enables zero-copy data access by directly transferring data from the SSD to the user's data buffer via DMA without maintaining a complicated page cache, by considering the characteristics (e.g., stream and batch processing) of GPU applications. It will automatically scale the number of I/O threads based on I/O request size with CUDA dynamic parallelism, and provide both synchronous and asynchronous modes to GPU programs. This simplifies the user's effort to manage parallel I/O, while allowing GoFS to maximize the I/O throughput with massive parallelism on GPUs.

We preserve the essential filesystem properties such as crash consistency (§3.5) in GoFS. As GoFS allows both the host and GPU to access the on-disk file system, it maintains the data consistency in a primary/secondary mode by running GoFS daemon on the GPU side as the primary, and GoFS client on the host as the secondary. This is reasonable, as GoFS is designed for GPU applications that prefer minimum host involvement for performance. When GoFS daemon is not running, the host client can use the host-based file system like F2FS to manage and access the SSD. When GoFS daemon

is active, the host client will sync the filesystem operations with the GPU-side daemon to maintain data consistency.

We leverage the GPU virtual memory to enforce the memory protection between the file system (GoFS daemon) and regular GPU applications. GoFS fulfills the file access control by using the host GoFS client to generate a unique and unforgeable identification (signature) for each user process with a cryptographic algorithm (see detailed procedure in §3.6). Therefore, when a user process initiates file accesses, GoFS daemon will validate its signature for access control.

We implement GoFS daemon using 5.5K lines of code with CUDA programming. We use the popular log-structured file system F2FS to format the SSD and manage the on-disk blocks. We implement GoFS client with 0.8K lines of C/C++ programming code based on FUSE [70] to intercept all the filesystem calls at the host side. We examine the efficiency of GoFS on a server consisting of an A100 GPU with GPUDirect storage enabled, a 16-core Intel Xeon processor, and multiple 2TB Samsung 990 Pro SSDs. We conduct the experiments with a variety of GPU-accelerated applications that include graph analytics, deep learning-based queries for vision, text, and audio data, GNN, and LLM-based RAG system. Our experiments show that GoFS outperforms state-of-the-art storage access solutions for GPUs by 1.61× on average. It also scales well as we increase the number of SSDs. In summary, we make the following contributions in the paper.

- We develop a GPU-orchestrated file system for scaling the direct storage accesses for GPU programs.
- We rethink the design and implementation of metadata and data I/O management in file systems for GPU-accelerated computing, and develop a set of optimization techniques.
- We enable the host and GPU to manage the shared SSD in a coordinated fashion, while ensuring data consistency by running GoFS in a primary/secondary mode.
- We implement an end-to-end system prototype of GoFS based on F2FS, and demonstrate its efficiency using diverse workloads on a real GPU with GPUDirect storage enabled.

2 Background and Motivation

In this section, we describe the technical background of GPUDirect storage. And then, we discuss its system support.

2.1 GPUDirect Storage

To feed data to the GPU for computation, the default method is to use the host CPU to first read data from the disk to the host memory, and then use `cudaMemcpy` to move data to the GPU. However, the redundant data copy not only causes performance overhead but also wastes precious CPU cycles [21, 58]. To relieve the host CPU of the costly data transfer, modern GPUs and NVMe SSDs support GPUDirect Storage (GDS) [66], which allows direct PCIe peer-to-peer (P2P) communication between a GPU and an SSD by exposing the GPU device memory to the PCIe Base Address Register (BAR) region. With PCIe memory-mapped I/O (MMIO), the memory requests can be directly issued to the PCIe end-point devices like SSDs.

To provide system software support for GDS, existing systems mainly rely on two methods. In the first method, the host CPU still handles all file system operations, such as indexing the data blocks. When programming the DMA engine on the SSD, the source/destination address is set to be the GPU device memory rather than the host memory. Thus, the data is directly copied between GPU and SSD without involving the host memory. A typical example is the NVIDIA cuFile library [66]. Since the host CPU still needs to manage the file system, it inevitably becomes a scalability bottleneck. In the second method, the NVMe queues are mapped in the GPU memory, which allows the GPU threads to directly issue NVMe commands to the SSD. An example is BaM [58], which completely removes the host CPU involvement in accessing the SSD. However, the GPU program must explicitly manage the SSD by itself as a raw block device. The most recent study [57] proposed to take advantage of both methods by preloading file metadata from the host to GPU memory beforehand and relying on BaM for direct storage accesses. It has fundamental limitations due to its strong assumption on GPU applications (i.e., predictable I/O pattern and read-only), and this assumption does not always hold in reality (see Table 1). As we accelerate more diverse applications with GPUs, it is essential to develop a generic approach for managing direct storage accesses.

2.2 GPU Architecture and Execution Model

Figure 1 shows the GPU architecture. Without loss of generality, we use NVIDIA CUDA terminology. A GPU consists of many streaming multiprocessors (SMs). Each SM has multiple SIMD units (CUDA cores), a private L1 cache, and a software-managed scratchpad memory called shared memory. All SMs share a global L2 cache and the off-chip DRAM memory.

The GPU uses a hierarchical thread execution model. A thread is the smallest unit of sequential execution. Threads are organized into warps (typically 32 threads/warp) to align with the width of the hardware SIMD units. All threads in a warp execute in lock-step. Multiple warps form a thread block, which is a group of threads executing on the same SM. Multiple thread blocks form a GPU program (also called a kernel). To synchronize among threads, the programmer can leverage atomic operations and memory fences at different scopes, such as local synchronization within a warp or a thread block, or global synchronization across thread blocks.

2.3 Challenges with System Support for GDS

Due to the unique execution model of GPU, building software for GPUs is much more challenging than for multi-core CPUs. First, we must explicitly consider the hierarchy of GPU threads (i.e., warp, thread block, and kernel) to achieve the best performance. We cannot simply port a file system designed for a multi-core CPU to a GPU, because the filesystem data structures and concurrency mechanisms designed for the CPU will

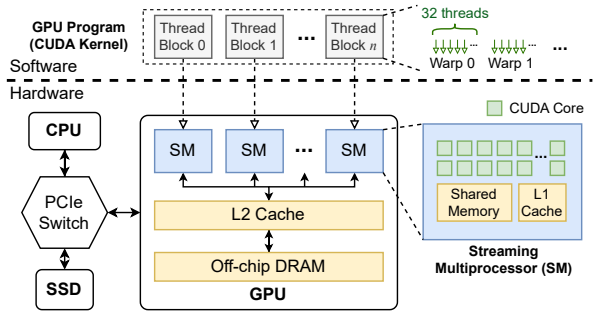


Figure 1. Modern GPU architecture and its execution model.

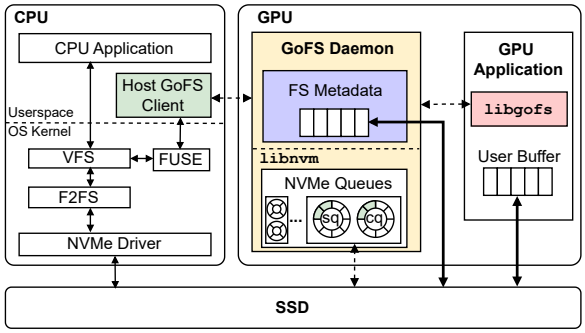


Figure 2. System architecture of GoFS.

not fit on the GPU. Second, the programming model of a GPU thread resembles a bare-metal CPU core. There is no underlying OS support, such as memory management (e.g., `kmalloc`). As the GPU programming model follows SIMD computing paradigm, it requires explicit parallelization in the program. Third, there are limited utility and helper libraries on GPUs for developing system software. Most existing GPU libraries focus on high-performance computing and machine learning, such as graph and linear algebra algorithms. However, developing system software requires a completely different set of tools, such as timers, customized data structures, and synchronization primitives. As we develop GoFS, we expect its practice will facilitate the development of these utilities.

3 Design and Implementation

In this paper, we develop GoFS, a GPU-orchestrated file system for managing scalable direct storage accesses.

3.1 Design Goals of GoFS

- *Scalability.* GoFS should exploit massive parallelism on the GPU to scale both metadata and data I/O management, while enabling scalable direct storage accesses.
- *Programmability.* GoFS should provide an easy-to-use interface for GPU programs. GoFS supports POSIX API, which are compatible with the API provided by most other file systems and libraries such as cuFile [66].
- *Data consistency.* GoFS needs to maintain data consistency between CPU and GPU, such that the host can also access the file system to manage the data on the storage device.
- *Protection.* GoFS needs to provide protection mechanisms for enforcing the isolation between a GPU program and the file system. It should also enforce file access controls when multiple GPU applications share the storage.

3.2 System Overview

We design GoFS as a log-structured file system. It uses the same on-disk format as F2FS [37] – a popular file system optimized for flash-based SSDs. We choose to build upon F2FS to provide the compatibility with existing systems and henceforth minimize the development efforts. We show the system architecture of GoFS in Figure 2. GoFS consists of a host-side client, a GPU-side daemon, and the `libgofs` library.

The GPU-side daemon in GoFS manages the in-memory filesystem metadata. We carefully examine all critical filesystem data structures and make them "GPU-friendly" by considering the bulk-synchronous parallel computing paradigm of GPU (§3.3). The daemon maintains the NVMe queues and performs data I/O accesses. GoFS exploits the massive parallelism of GPU to maximize the storage I/O throughput. It also enables zero-copy I/O access by directly transferring data into the user-provided buffer via DMA (§3.4).

The host applications can access GoFS via the host client, which uses FUSE [70] to intercept all filesystem calls. When the GPU-side daemon is not running, the host client uses the unmodified F2FS stack to manage and access the SSD. When the GPU-side daemon is active, the host-side client and GPU-side daemon will work in a primary/secondary mode to maintain data consistency. The host client will sync the filesystem operations with the GPU-side daemon. GoFS ensures data consistency and crash consistency (§3.5). GoFS provides data protection and file access control. The GoFS daemon runs in a separate process, which prevents the applications from corrupting the in-memory data structures. To enforce file access controls, the daemon uses cryptographic signatures to verify user identities (§3.6). We will show the workflow of GoFS in §3.7.

3.3 Filesystem Metadata Management in GPU

We rethink the design and implementation of in-memory metadata structures of log-structured file systems with various optimization techniques, such as scalable accesses to inode/dentry, scalable data block management with block bitmaps, and parallel traversal of data pointers by carefully exploring the GPU-accelerated computing paradigm. We elaborate each of them as follows.

Scalable accesses to inode. An inode stores the basic information of a file, such as file size, data location, access rights, and modified time. Traditional Linux file systems use a mutex lock or R/W semaphore on the entire inode to serialize concurrent access requests to a single file. However, to best exploit the data parallelism of GPU workloads, GoFS splits data access to a single large file across GPU threads (see §3.4

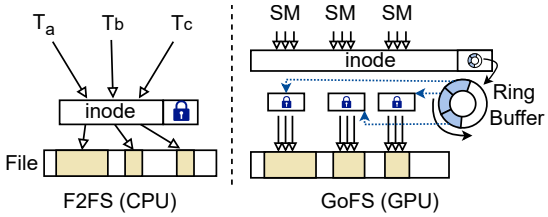


Figure 3. Inode design with a range lock in GoFS.

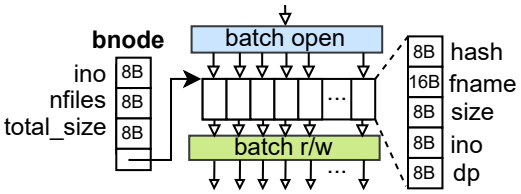


Figure 4. Batched requests in GoFS.

for details). Thus, a coarse-grained lock on the entire inode will incur high contention between threads even if they do not access overlapped data regions (see Figure 3 left).

GoFS replaces the inode mutex with a range lock for fine-grained concurrency control based on accessed I/O ranges (see Figure 3 right). The range lock is structured as a ring buffer. Each slot in the ring buffer can hold an I/O request range, including the I/O state (1B), file offset (8B), and access size (8B). We choose the ring buffer, since it allows us to parallelize range checks with GPU's warp-level reduction primitives [41].

GoFS acquires a single range lock for all threads in the same thread block, since GPU applications typically organize threads with spatial data locality into the same thread block. GoFS acquires the lock by appending the new range into the ring buffer and performing a conflict check against previous slots. The conflict check is parallelized within a warp using warp reduction primitives, and it incurs negligible overhead compared to the end-to-end latency of an I/O request. Upon access range conflict, the newer request must wait by polling the I/O state field of the conflicting range. For threads in the same thread block, GoFS guarantees non-overlapping data access within a file system operation by partitioning the large request across all threads and performing barrier synchronizations to avoid any lock contention. To prevent starvation, where a lock request with a large range is blocked by subsequent lock requests in smaller overlapping ranges, we guarantee a newer lock acquisition will not succeed earlier than a previous request within an overlapping range.

Unlike range lock management in conjunction with interval trees [34], which has an optimal $O(log(n))$ complexity for the range conflict check for each single CPU thread, GoFS parallelizes checking the ranges and achieves effectively near-$O(1)$ complexity with fast warp-level reduction primitives.

Scalable accesses to dentry. GoFS implements a dentry cache on the GPU to speed up the path resolution process. Similar to the dentry cache in the host virtual file system (VFS), each in-memory GoFS dentry holds a hash table storing all cached child dentries indexed by the directory/file name. A lookup operation uses the pathname as the key to traverse down the file system hierarchy without reading on-disk inodes. A direct lookup hash table (DLHT) [69] indexed by full pathname serves as a fast path for directories with access locality. Different from a host file system, which uses the RCU lock [46] for the concurrency control of the hash table, GoFS uses a simple per-bucket lock. This is because RCU involves dynamic memory (de)allocation and complicated inter-thread callbacks, which incurs extra performance overhead on GPU. The lock contention will happen only when multiple threads access the same file/directory and one of them tries to rename it, which is a rare case in GPU applications.

To further exploit the parallelism across massive files, GoFS introduces the *batched node* (bnode) structure to support batched file operations. The batch operations are useful for accessing datasets organized in the form of multiple small files in the same directory, which is common for many popular datasets (see Table 1). A bnode tracks a collection of opened files under the same directory, and it uses the same inode number (`ino`) as the directory inode (see Figure 4). Within the bnode, GoFS records the total file count, total data size, and an array of batch-opened files. With the batched operations, GoFS coalesces metadata operations (e.g., the parent inode for all files under a directory only needs to be updated once) and eliminates unnecessary locks (e.g., all threads in a batched API will process different metadata/data without overlaps).

To use the batched operations, a GPU application needs to call `batch_open()` in GoFS with a directory path to open all files under this directory (see §3.7 for API details). GoFS leverages parallel threads to read all child entries under the directory node, check the access permission of the child file, and fill the bnode file array. For each file, we store its name hash, short filename, file size, inode number, and a pointer to its cached data pointer. After calling `batch_open()`, the application can issue batched read/write requests on the bnode instance.

Scalable block allocation with per-SM bitmaps. Traditional host file systems typically maintain a centralized data structure (i.e., block bitmaps in F2FS) to track the free blocks. Each bitmap encodes the occupied and free blocks in a 2MB segment (the segment size is usually aligned with the erase block size of an SSD). Upon block allocation, multiple CPU cores will compete for the block bitmaps (Figure 5 left). To avoid this, GoFS applies two optimizations (Figure 5 right).

First, GoFS maintains per-SM bitmaps to eliminate contention between SMs. Upon initialization, GoFS assigns bitmaps to an SM based on the number of free blocks in the segments, such that all SMs have similar numbers of free blocks. When the number of free blocks to any SM is below a certain threshold (5% by default), GoFS will redistribute the free segments among all SMs, which can be fulfilled in-memory quickly.

Second, GoFS coalesces the block allocation operations of all threads in a thread block into a single allocation to reduce

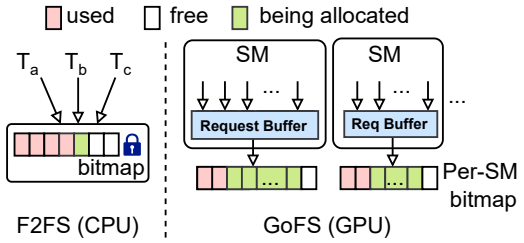


Figure 5. Block bitmap used in GoFS.

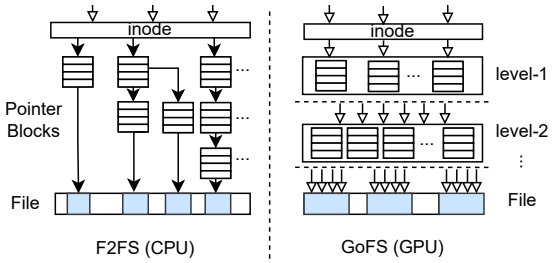


Figure 6. Level-synchronous data pointer traversal in GoFS.

contention on the per-SM bitmaps. Instead of letting each thread allocate new blocks on its own and contend on the per-SM bitmaps, we first use all threads in the thread block to collaboratively compute the total number of blocks needed by all threads. Then, a single thread (e.g., thread 0) will be responsible for manipulating the per-SM bitmaps. After that, we assign the allocated blocks to each thread.

Parallel traversal of data pointers. Similar to other file systems, GoFS uses a tree-like structure to index the file data blocks. The root of the tree is the inode. Each intermediate node is a block of data pointers, and each data pointer either points to a data block (i.e., direct pointer) or another block of pointers (i.e., indirect pointer). All data blocks are leaf nodes.

In a traditional file system, when multiple CPU threads read the same file, each thread independently traverses the tree from the root to find the leaf data blocks (see Figure 6 left). This scheme leads to two problems on the GPU. First, there is a load imbalance among different threads since the leaf nodes can have different depths. As a result, the straggler thread will stall the entire GPU kernel. Second, data blocks and pointer blocks need to be handled differently. This leads to branch divergence if two threads in the same warp follow different code paths to handle different block types, which significantly wastes the parallelism of the GPU.

Inspired by previous GPU graph traversal algorithms [47], we develop a *level-synchronous* parallelism scheme to traverse the data pointers (see Figure 6 right). We use a two-stage procedure to index and fetch the data blocks. In stage one (metadata stage), we traverse the data pointer tree to find the logical flash page addresses (LPAs) of all leaf data blocks. In the tree traversal, all nodes at the current level can be processed in parallel to find the nodes at the next level, and the threads perform barrier synchronization between consecutive levels. This balances the number of nodes being processed by each thread at each level and eliminates branch divergence since all threads execute the same code path when processing data pointers. In stage two (data stage), we fetch all data blocks in parallel to maximize the storage throughput of SSDs by utilizing the data I/O techniques in §3.4.

Synchronized GPU clock. To ensure the correct ordering of filesystem operations, GoFS requires a timestamp within each log entry. And the timestamps generated on the GPU should be *synchronous* with the host clock. This is because the timestamps will be used as the creation or modification times of directories and files, and many popular applications depend on this information for correctness. For example, `Makefile` leverages the file modification time to detect file changes to implement incremental compilation. As GoFS allows both host and GPU to create and modify files, it is necessary to ensure that both sides use consistent timestamps.

However, the current GPU does not provide a clock synchronized with the host CPU. Naïvely retrieving every timestamp from a host clock will require frequent communication between CPU and GPU. To avoid such overhead, we implement a GPU-side clock by leveraging the GPU's special register `globaltimer` [52] as a fast local timer, and we periodically (every 10 minutes by default) calibrate the GPU clock with the CPU clock. With the default period, the worst clock skew for the GPU-side clock is 1368 μs (326 μs on average). GoFS allows the user to configure the synchronization period to achieve higher clock accuracy (e.g., with a 1-minute period, the worst clock skew is less than 150 μs). This is sufficient for most applications and systems. For example, the common time skew tolerance of Windows Time service (W32time) ranges from 1 millisecond to 1 second [48].

To synchronize the clock, GoFS requests a CPU timestamp and stores the clock offset between the CPU clock and the GPU local timer. GPU local timestamps are calculated by the clock offset and the local timer value.

3.4 Data I/O in GoFS

The GoFS daemon maintains multiple NVMe queues in the GPU memory to serve parallel I/O requests from multiple SMs based on `libnvm` [44]. It enables zero-copy I/O access by directly DMAing data from the SSD to the user's data buffer without maintaining a complicated page cache, by considering the characteristics (e.g., streaming data access and batch processing) of GPU applications. GoFS automatically scales the number of I/O threads based on request size. And it provides both synchronous and asynchronous APIs to user programs.

I/O queue setup. In GoFS, the number of I/O queues (i.e., NVMe queue pairs) and the queue depth can be configured based on the number of SMs and the SSD specification. By default, GoFS creates as many I/O queues as supported by the SSD (e.g., 16 queues for a commercial SSD like Samsung 990 Pro[1] and 128 queues for datacenter SSDs like Intel Optane [1]).

[1]We obtained the number of supported queues using `nvme-cli` on the Samsung 990 Pro device.

The default queue depth is 2048 (i.e., the maximum number of concurrent threads supported by an SM [55], which often cannot be reached due to register and shared memory limits). This is more than enough to saturate the SSD throughput.

I/O request dispatch. GoFS employs a static I/O request dispatching policy. GoFS partitions the I/O queues evenly among all SMs and routes the requests from an SM to its corresponding queues. This simple policy incurs no runtime scheduling overhead. It will not cause queue imbalance since most GPU applications will balance the load across SMs by default.

Zero-copy I/O access. In GoFS, all data I/O accesses are zero-copy. The data will be directly DMAed from/to the user program's buffer, and we let the user decide whether to employ a caching layer. This is in sharp contrast to a traditional host file system, which uses a page cache to exploit data locality. A page cache will not be helpful to most GPU applications, since they often (1) work on a large dataset that cannot be fully cached, (2) have a streaming data access pattern without any temporal locality, (3) already manually keep frequently used data in GPU memory, and (4) are throughput-oriented workloads that can tolerate the long SSD access latency without a cache. For these applications, adding another layer of cache not only increases I/O overhead but also wastes precious GPU memory. For applications that can benefit from a page cache, we can still rely on existing libraries on top of GoFS to provide caching functions and an `mmap`-like interface. Note that GoFS still maintains a filesystem metadata cache.

Automatic data-parallel I/O access. To serve a read/write request, GoFS will launch a new GPU kernel using CUDA dynamic parallelism [3]. This gives GoFS the flexibility of automatically scaling the number of threads, such that each thread serves a chunk of the requested data (i.e., data parallelism). In contrast, a host file system will not automatically spawn multiple threads to serve a single read/write request.

GoFS determines the kernel launch configuration (i.e., the number of thread blocks and the number of threads per block) based on the request size. GoFS will try to utilize all SMs so that all I/O queues will be utilized to maximize I/O throughput. Therefore, GoFS launches as many thread blocks as necessary where each thread block serves a minimum of 4KB data chunk (the FS data block size). The maximum number of thread blocks is constrained by the number of SMs. The number of threads per block is 64 by default, as we find out empirically that having more threads per block will not improve I/O throughput when there are already enough thread blocks. GoFS also allows the user to explicitly specify the kernel configuration for a read/write request.

Synchronous and asynchronous data access. In synchronous mode, a read/write API call will block until the data is ready in the user buffer. GoFS will keep polling the I/O queue to determine if the data has been DMAed to the destination. In asynchronous mode, an API call will just forward the request to the GoFS daemon, and the user kernel can continue execution. `libgofs` provides another API to wait for the request completion by polling the request status from the daemon. The user application can leverage the asynchronous APIs to implement software pipelining to hide the I/O access latency.

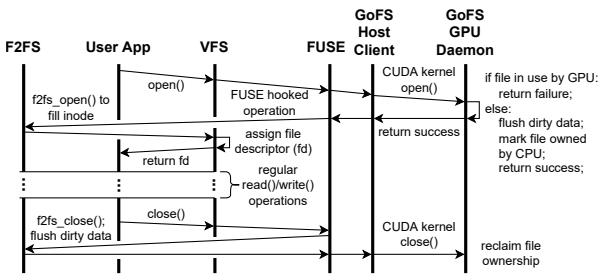


Figure 7. Data consistency between CPU and GPU in GoFS.

3.5 Data Consistency

Data consistency between CPU and GPU. GoFS allows both CPU and GPU programs to access the filesystem. GoFS uses FUSE [70] to implement a host-side client to maintain data consistency between CPU and GPU. FUSE enables the OS kernel to redirect CPU-side GoFS file system syscalls to the userspace host client *without modifying user applications or the OS kernel stack*, offering the best compatibility with the existing system stack. A userspace client is necessary since a CUDA kernel can only be launched from the userspace. GoFS does not change the F2FS on-disk layout, so we do not need to modify the kernel F2FS implementation.

In GoFS, the GPU daemon manages the ownership of opened files. The GPU-centric design is reasonable as GPU applications typically prefer minimum host involvement.

Figure 7 shows the workflow of file operations issued from CPU-side applications. GoFS redirects the `open()` syscall to the userspace GoFS client. When the GPU daemon is not running, the host client directly returns the redirected `open` and enters the original F2FS stack. The host client invokes custom GPU kernels to acquire access ownership of the target file. If no GPU application is accessing the file, the GPU daemon grants the file ownership to the host. FUSE then calls the normal `f2fs_open()` function to open the file. After the file is opened, the user application can transparently issue file I/O requests to the underlying F2FS without involving the GPU.

When the application `close()`s a file, FUSE directs the function call to the normal `f2fs_close()`, letting it flush all related dirty data. Then, it forwards the `close()` to the GPU daemon (similar path as `open()`) to return the file ownership.

In addition, GoFS allows the GPU and CPU to concurrently read the same file (i.e., sharing ownership) by opening the file in read-only mode. If the CPU/GPU wants to reopen the file in read/write mode, it will wait for all previously opened file instances to be closed. GoFS keeps a per-file reference counter to track the number of opened file instances to achieve this.

Crash consistency. Since GoFS does not change the on-disk format of F2FS, it provides the same crash consistency guarantees. As a log-structured file system, GoFS persists new

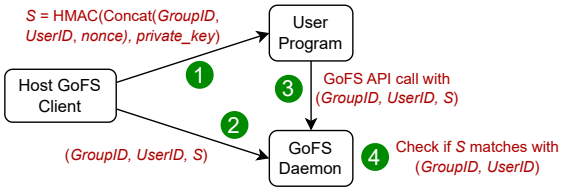


Figure 8. User identity authentication in GoFS. ❶ and ❷ only happen once when the user process registers itself with GoFS daemon. ❸ and ❹ happen for every GoFS API call.

```
// POSIX vector read/write
readv(int fd, void* buf, struct request reqs[]);
writev(int fd, void* buf, struct request reqs[]);

// GoFS batched open operation
batch_open(char* path, char* fnames[], int boffset[]);
batch_close(int fd);

// GoFS batched read/write operation
batch_read(int fd, void* buf, int boffset[]);
batch_write(int fd, void* buf, int boffset[]);
```

Figure 9. New file operation APIs supported by GoFS.

data blocks and node blocks out-of-place and propagates the changes by updating the block address in the NAT table. To provide a consistent recovery point, GoFS triggers a checkpoint procedure to flush all dirty metadata and persist the current metadata within a checkpoint pack. GoFS can directly use the host F2FS stack for crash recovery without GPU involvement. GoFS scans the checkpoint region to extract the latest checkpoint pack and roll back the file system metadata to the latest consistent stage. The recovery overhead is less of a concern as it is not on the critical path of application execution.

3.6 Data Protection

Unlike the CPU, a GPU does not provide multiple privilege rings. Thus, GoFS needs to enable data protection differently.

In-memory filesystem integrity protection. GoFS leverages GPU's virtual memory isolation to enforce "privilege rings". It maintains a daemon process (i.e., the "kernel space"), which has the privilege of managing filesystem metadata and the NVMe queues in the GPU memory. The user process (i.e., the "user space") cannot access the memory of the daemon process. The daemon launches a persistent daemon kernel on the GPU to listen to the filesystem API calls from user processes. It only occupies a single SM, and the user process can utilize other SMs with CUDA MPS [51]. The user process forwards GoFS API calls to the daemon via a shared request queue in the GPU memory (implemented using CUDA IPC [55]).

File access control. When GoFS's daemon process opens a file, it checks whether the user process has permission to access this file. For a host file system, this can be done by directly checking the user/group ID of the user process, since the OS kernel by default maintains the data structures to manage all processes. However, on the GPU, the GoFS daemon cannot directly track the user/group ID of a user process.

To address this challenge, we leverage the trusted host GoFS client to generate a unique and unforgeable "ID" for each user process. We illustrate the user identity authentication mechanism in Figure 8. When a user process initializes GoFS, it needs to register itself with the host GoFS client. The client runs with sudo privilege and can retrieve the correct user/group ID of the user process. Then, the client uses a cryptographic algorithm (e.g., HMAC-SHA256 [50]) to sign the user identity (i.e., group/user ID) and a randomly generated nonce with a private key that can only be accessed by the client. Then, the client passes this signature to the user process and guarantees that the signature will not be exposed to other processes (❶). It also passes the user identity and the signature to the GoFS GPU daemon, such that the daemon can maintain a mapping from user identities to signatures (❷). When the user process invokes a GoFS API, it sends the signature along with the user identity to the daemon process (❸). The daemon then verifies that the user identity and the signature match, which guarantees that the provided user identity is not forged by a malicious process (❹).

3.7 Put It All Together

GoFS APIs. GoFS provides the POSIX API for GPU applications for programmability and compatibility, as it is a commonly adopted standard for file systems. Applications only need to call filesystem APIs to access files, GoFS transparently maximizes the I/O throughput with automatic data-parallel I/O accesses. Besides, GoFS also provides a few extended APIs to further optimize file operations, as shown in Figure 9.

`readv/writev`: Perform file read/write with an array of requests (`reqs[]`). Each request contains the requested file offset, length, and the offset to the data buffer (`buf`).

`batch_open/batch_close`: Open up multiple files under the directory with `pathname` as a bnode. Users may pass an array of `fnames[]` to specify files to open, otherwise all files are opened under the directory. The file position within the bnode is returned in an array of `boffset`.

`batch_read/batch_write`: Issue batch read/write requests to the files within the bnode. Batch read fetches the entire file in `boffset` and loads data into the provided buffer. Batch write overwrites the buffer data to the file according to the `boffset`.

Daemon setup. By design, GoFS can be mounted on any SSD formatted in F2FS. To mount GoFS, a privileged user calls the `mount` function with the disk controller identification, and the host GoFS client then launches a GPU daemon kernel. The daemon initializes NVMe queue pairs and reads the file system superblock to check the on-disk FS structure. It then builds the runtime structure (e.g., block bitmap) in GPU memory. Before calling GoFS APIs, the user process registers itself with the host GoFS client via IPC. The host client will then set up the GoFS API request queue in the GPU memory. The request queue is shared between the GPU daemon and the user's GPU

kernels via CUDA IPC. The overhead of GoFS initialization is trivial (less than 50 milliseconds in total on our testbed).

File operations. We use an example to show the workflow of GoFS's file operations. An application first opens the target file with the file path, and GoFS performs a file lookup. After finding the inode and opening the file, the application can issue a file `read`. GoFS acquires the data range lock (3.3), and performs the two-stage (metadata and data) file read with multiple GPU threads. The read requests are dispatched among SSD I/O queues, and the SSD directly DMAs the requested data to the application buffer (3.4). The range lock is lifted after request completion. After performing some computation, the application can overwrite the result back to the file. During parallel writes, GoFS allocates new free blocks with the per-SM allocation pool for data. After the data is persisted, GoFS updates the data pointer in parallel.

Garbage collection. GoFS implements garbage collection process to reclaim scattered and invalidated blocks in the file system. GoFS daemon periodically checks the number of available free segments, and it triggers foreground (on-demand) GC process when the unused capacity drops below a configurable threshold (5% by default). To speedup the GC procedure, GoFS daemon launches a new GC kernel with multiple thread blocks. Each thread block greedily selects a victim segment (2MB) with the least valid block, and writes the valid 4KB data blocks to a newly allocated segment. It then sends a trim NVMe command to the SSD with the I/O queue to clean up the blocks in the segment. Finally, GoFS updates the in-memory block bitmap and the on-disk SIT table (segment info table).

GoFS also supports background GC. The host client of GoFS monitors GPU utilization using the `PROF_SM_OCCUPANCY` metric of DCGM [54], and derives the number of active warps on the GPU. When there is only one active warp (i.e., only the GoFS daemon is running on the GPU), the host client will launch special GC kernels. GoFS does not trigger the default GC of F2FS on the CPU, since this may require frequent synchronization of file ownership and global file system metadata between the CPU and GPU.

3.8 Implementation Details

GPU daemon. We implement the GPU-side GoFS daemon using 5.5K lines of CUDA C++ code with CUDA 12.4. The in-memory data structures, such as hash table, ring buffer, lists, and trees account for 2.4K lines of code. We also implement a buddy allocator within GoFS daemon for managing the file metadata caching. We use the `libnvm` [44] to enable direct NVMe access inside GPU kernels.

`libgofs`. We implement a `libgofs` library with 1.4K LoC which provides library functions for GPU applications to interact with GoFS host client and GPU daemon.

Host client. The GoFS host client is implemented in 0.8K lines of C++ code. We modified around 0.2K lines of C code to redirect the GoFS file operation calls to the userspace GoFS client using the FUSE kernel module.

The kernel launch overhead in GoFS is 9.2 μs on average. In sync mode, most thread blocks launched by the GoFS kernel will finish after they have submitted the NVMe commands, and only a few thread blocks (empirically, we use 16 thread blocks by default) will remain active to poll for I/O completion. Only up to 16 out of 108 SMs on our A100 GPU are occupied by GoFS, and the user application's kernels can execute on other SMs. For I/O-intensive applications, occupying a few SMs for polling is acceptable, since they are not bottlenecked by the computation. For compute-intensive workloads like ML training/inference, their I/O demand is relatively low compared to computation, therefore, it usually requires fewer SMs for serving I/O requests. Note that GoFS needs only one SM for its daemon when the I/O is idle, it will dynamically adjust the SM allocation depending on the I/O intensity. In async mode, the GoFS kernels will finish after submitting the NVMe commands and do not occupy any SM to wait for I/O completion, so the interference to the user threads is insignificant.

4 Evaluation

Our evaluation shows that: (1) GoFS achieves 1.10×/1.03×, 1.38×/1.65×, and 2.04×/2.40× higher sequential, random, and multi-file read/write throughput compared to current storage solutions for GPUs. GoFS significantly saves host CPU cycles while achieving high I/O throughput (§4.2). (2) GoFS improves real-world GPU application performance by 1.61× compared to state-of-the-art designs (§4.3). (3) GoFS achieves near-linear I/O throughput scalability as we increase the number of SSDs (up to 20.4GB/s sequential read with four SSDs) (§4.4).

4.1 Experimental Setup

Testbed. Our testbed has a 16-core Intel Xeon W5-3435X CPU, 64GB DDR5 host memory, and an NVIDIA A100 GPU with 40GB device memory. We use Ubuntu 20.04 with Linux kernel 5.4.0, CUDA 12.4, and NVIDIA GPU driver 550.127. GoFS is mounted on a 2TB Samsung 990 Pro SSD.

Workloads. We evaluate GoFS with microbenchmarks and real data-intensive GPU applications (see Table 1). For the microbenchmarks, we vary the number of host CPU threads and the number of GPU thread blocks for sensitivity analysis. For real applications, we include intelligent queries, graph analytics, graph neural network (GNN) trainings and retrieval-augmented generation (RAG) systems, as they are representative GPU workloads in production today [15, 19, 38, 39]. Finally, we also test GoFS's scalability with multiple SSDs.

Baselines. We compare GoFS to state-of-the-art storage access solutions for GPUs, which enable different degrees of GPU control over the filesystem:

- **Basic:** The host CPU executes filesystem operations and accesses the disk using F2FS. Data is first staged in host memory and then transferred to GPU using `cudaMemcpy`.
- **GPUfs [62]:** A wrapper library that enables GPU threads to invoke file system APIs. It delegates the API calls to

Table 1. Workloads used in our evaluation. The "Predictable" column indicates whether the workload has a predictable storage I/O access patterns or not.

Workload	Description	Predictable
Microbenchmarks		
`Seq-R/W`	4KB sequential file read/write	Yes
`Rand-R/W`	4KB random file read/write	No
`Multi-file-R/W`	100K 16KB small files read/write	Yes
Applications		
`IIR` [22]	Given an image query, searches for the most similar image in the 636GB LSUN [75] dataset (53M samples with 12kB file size, preprocessed).	Yes
`TIR` [72]	Takes a textual query to retrieve the matched images in 24GB ImageNet [24] dataset (2M samples with 12KB file size, preprocessed).	Yes
`MIR` [42]	Searches for music samples matching a given audio clip in 110GB AudioSet [42] dataset (10M samples with 12KB file size, preprocessed).	Yes
`RAG` [19, 38]	LLM-based retrieval-augmented generation with Llama2-13B [68] and 3.6TB vector DB.	No
`BFS` [6]	BFS search with 15GB GAP-web [6] and 920MB GAP-road [6] dataset.	No
`CC` [6]	Connected Component Search with 32GB GAP-kron [6] and 15GB GAP-web graph dataset.	No
`GNN` [23, 33]	Graph neural network training with GraphSAGE [23, 25] on 3.2TB WebGraph [9] dataset.	No
`Image Preprocess` [35]	Preprocessing original 167GB ImageNet [24] dataset for downstream tasks like DNN training.	Yes

the host filesystem (i.e., F2FS) via Remote Procedure Calls (RPCs). The host CPU runs server threads to process the RPCs and execute filesystem operations. The data still needs to be transferred between CPU and GPU via `cudaMemcpy`. It maintains a file cache in the GPU memory.

- **cuFile [66]:** The application uses NVIDIA cuFile APIs to load data directly from SSD to GPU device memory via GPUDirect Storage. The data will not be staged in the host memory. However, the host CPU still processes filesystem operations and issues NVMe commands to the SSD. cuFile uses ext4 since it does not support F2FS.
- **GeminiFS [57]:** GeminiFS uses the host to manage metadata while offloading data I/O to the GPU. Upon opening a file, GeminiFS uses the host filesystem to retrieve its metadata (e.g., data pointer) and copies it to the GPU. With this metadata, the GPU can issue NVMe commands to read/write the data blocks in the SSD. The GPU can read/write the per-file metadata, but it cannot access directory or global filesystem metadata. The GPU-side metadata is freed upon file closing. We also allocate a 1GB page cache in the GPU for its prefetching mechanism.

4.2 Microbenchmarks

We use microbenchmarks to evaluate GoFS's I/O throughput against the baselines, following popular file system benchmarks such as fio [18] and filebench [65]. For sequential read/write, we directly call GoFS's read/write APIs. For random read/write, we pre-generate random 4KB-aligned file offsets and use GoFS's vector APIs. For multi-file read/write, we use GoFS's batch APIs (see Figure 9). For all microbenchmarks, GoFS enforces the locking on all file operations to ensure atomicity and guarantee data correctness. GoFS uses range locks to support concurrent accesses to different file regions. We list our findings as follows.

(1) GoFS scales its I/O throughput much better than all baselines as we increase the number of GPU thread blocks. Figure 10 shows the I/O throughput of all designs for varying numbers of GPU thread blocks. For all baselines, we use the number of CPU threads that maximizes the throughput.

For sequential read/write, `cuFile` has the lowest throughput (0.44GBps/0.48GBps) due to high host stack overhead. The write throughput of Basic is only 1.1GB/s due to lock contentions among CPU threads. GPUfs performs 1.6×/1.2× worse than Basic for read/write, as it has high RPC overhead between CPU and GPU when the request size is small. Both GeminiFS and GoFS achieve better throughput than other baselines, as they leverage the GPU to accelerate data I/O operations. GoFS achieves slightly better sequential read/write throughput than GeminiFS. This is because GoFS carefully optimizes the metadata management in GPU memory to reduce unnecessary locking overheads (e.g., range locks and per-SM bitmaps). In contrast, GeminiFS suffers from higher lock acquire/release overhead even when there is little contention with the sequential access pattern. GoFS achieves 5.5GBps/6.5GBps read/write throughput, which is close to the peak raw throughput of our SSD (6.9–7.5GB/s [2]).

For random read, GoFS achieves 5.1GB/s peak throughput, which is close to the peak raw SSD throughput (5.4GB/s, as measured on the raw block device without a file system). While both GoFS and GeminiFS leverage GPU's high parallelism to accelerate data I/O, GoFS is 1.4× faster. This is because GoFS allows all GPU threads to issue requests in parallel (at maximum parallelism, each of 6,912 CUDA cores can execute a thread on A100), while GeminiFS's warp-level lock serializes all requests in a single warp (at most 432 active warps), wasting the thread-level parallelism inside each warp.

For random write, GoFS achieves near-sequential throughput (6.1GB/s). This is because GoFS is log-structured and optimizes for parallel log updates, so random writes are sequentially appended into the log using multiple GPU threads. Basic also uses the log-structured F2FS but cannot scale due to multi-thread contention. GeminiFS is 1.6× slower than GoFS for random writes since (1) it does not exploit parallelism inside a warp (for the similar reason to random read), and (2) it is based on ext4 and the overwrite operations cannot be converted to sequential log appends. Note that GeminiFS cannot be adapted to a log-structured file system by design since the GPU does not manage global filesystem metadata and cannot support data block allocation and invalidation.

For multi-file read/write, GoFS employs the batched APIs to process the files in parallel. This significantly reduces the file

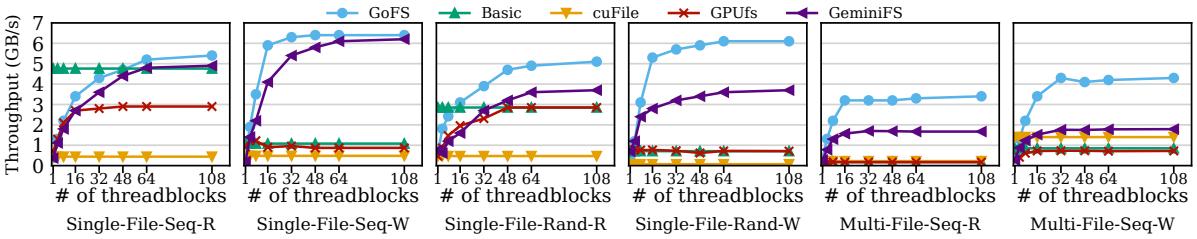


Figure 10. Performance of microbenchmarks with varying numbers of thread blocks on the GPU.

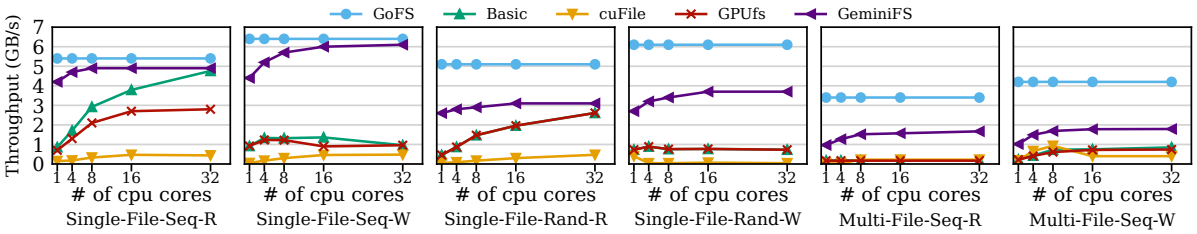


Figure 11. Performance of microbenchmarks with varying numbers of CPU cores.

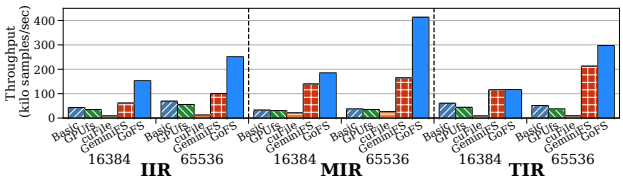


Figure 12. End-to-end query processing throughput of intelligent query applications.

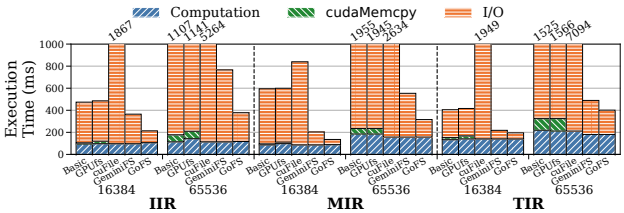


Figure 13. Execution time breakdown of a single batch in intelligent queries. We show the GPU Computation time, `cudaMemcpy` time, and I/O time (including filesystem software overhead and disk access time).

system metadata overhead and fully exploits the massive data parallelism of GPU threads. In contrast, all other baselines rely on the host to process metadata operations upon opening/closing a file. As a result, GoFS can achieve 15.5× and 4.6× multi-file read and write throughput improvements compared to the state-of-the-art baseline. To further understand the benefit of batch API enabled in GoFS, we conduct an ablation study in comparison with the case in which each thread calls open/read/write/close on a separate file. Our experiments show that using the batch API improves the throughput by 1.67×/1.38× for multi-file read/write respectively[2].

(2) GoFS significantly saves host CPU cycles while achieving the peak SSD throughput. Figure 11 shows the I/O throughput for varying numbers of CPU threads. GoFS is not affected by this parameter since it does not involve the host CPU at all during I/O accesses. For GPUfs, GoFS, and GeminiFS, we use 108 thread blocks on the GPU to maximize the parallelism. For all microbenchmarks, GoFS achieves better peak I/O throughput than all other baselines as discussed above. Even when other baselines perform close to GoFS (e.g., for sequential read/write), they need to occupy at least 8–16 CPU cores to achieve the best performance. GoFS frees up the precious CPU cores. As we scale to multiple GPUs and multiple SSDs, the CPU bottleneck will be more severe, so saving CPU cycles will become more beneficial.

[2]We use ablation studies to quantify the benefit of the batch API in GoFS, the results are not shown in the figures of the paper.

4.3 Real-World Applications

We now evaluate GoFS with real-world GPU-accelerated applications as listed in Table 1.

4.3.1 Intelligent Queries. The intelligent queries perform batch processing of small files. The dataset is processed iteratively. In each iteration, the application first fetches a batch of samples into the GPU memory, which involves opening and reading a batch of small files. And then, GPU executes a DNN model to conduct similarity search against all samples in the dataset. The computation of the current batch and the data fetch of the next batch are pipelined.

Figure 12 shows the end-to-end query processing throughput. GoFS outperforms Basic, GPUfs, cuFile, and GeminiFS by 6.2/7.5/21.3/2.1× on average. Figure 13 further breaks down the execution time of a single batch. For small batch sizes, the GPU compute capability is not saturated, and computation is the bottleneck. Hence, GoFS can have similar performance to GeminiFS. For larger batch sizes, the end-to-end execution is

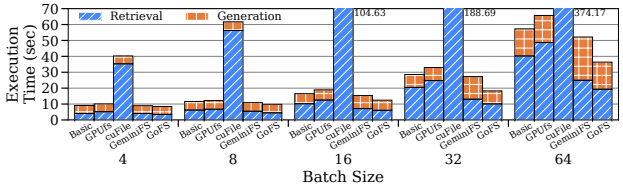


Figure 14. Request latency breakdown of RAG applications.

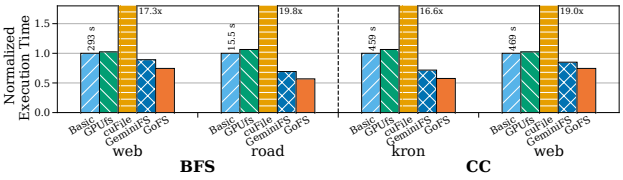


Figure 15. End-to-end execution time of graph analytics.

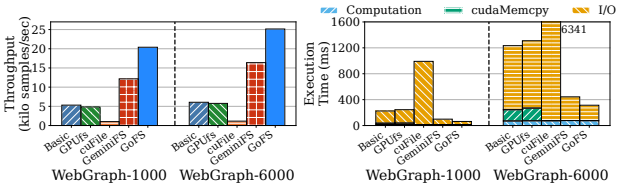


Figure 16. Throughput (left) and single-batch execution time breakdown (right) of GNN training. 1000/6000 are batch sizes.

bounded by I/O time. GoFS significantly reduces the I/O time compared to all other baselines. This is because GoFS employs the batched file API to leverage GPU's parallelism to handle massive metadata operations due to multi-file accesses, while all other baselines rely on the host to access on-disk metadata. For intelligent queries, our ablation study shows that the batch API of GoFS improves the query throughput by 1.41× on average, compared to the case without using the batch API.

Notably, GoFS's improvement over GeminiFS for `IIR` (2.5×) and `MIR` (2.7×) is higher than that for `TIR` (1.4×). This is because GeminiFS needs to keep the metadata of all opened files in GPU memory, as the GPU cannot retrieve any missing metadata from disk without host intervention. The metadata in GeminiFS occupies at least 8KB of GPU memory per opened file. For `TIR`, the metadata for the entire dataset (15.3GB) can be kept in GPU memory, so there is no host metadata access overhead. As the dataset gets larger, the metadata (404.4GB for `IIR` and 76.3GB for `MIR`) greatly exceeds GPU memory capacity (40GB), so GeminiFS needs to open and close the files for each batch, leading to significant host metadata access overhead.

4.3.2 Retrieval-Augmented Generation. RAG [19, 38] performs similarity search (e.g., ANN) to retrieve the top-relevant items from an on-disk vector database. The database indexing involves graph traversals, resulting in frequent, small, and data-dependent I/O accesses [27]. The retrieved items are fed into the LLM to generate the final answer. We use vLLM [36] as the LLM engine. For Basic, we use the host to perform the retrieval and the GPU to perform LLM generation. For cuFile, GPUfs, GeminiFS, and GoFS, the GPU performs both retrieval and generation. For GeminiFS, we keep the metadata for the entire dataset in GPU memory (7.2GB for our 3.6TB dataset) as opening the files on demand will result in lower performance. We use two 2TB SSDs with RAID0 as the dataset is 3.6TB (see the multi-SSD setup in §4.4).

Figure 14 shows the request latency breakdown for varying batch sizes. cuFile performs significantly worse than all other designs due to its poor random 4K I/O performance (see §4.2). GoFS outperforms Basic, GPUfs, and GeminiFS by up to 1.6/1.8/1.4×. Basic, GPUfs, and GoFS have similar generation time as this does not involve any file I/O. However, GeminiFS suffers from up to 1.6× slower generation time as the batch size increases to 64. This is because GeminiFS keeps the metadata of the entire 3.6TB database in GPU memory (7.2GB footprint), leaving less space for the KV cache. For the retrieval time, both GeminiFS and GoFS perform better than Basic and GPUfs, and GoFS outperforms GeminiFS by 1.2×. This aligns with our observations for the random read access pattern in §4.2.

4.3.3 Graph Analytics. We evaluate two representative graph algorithms: breadth-first search (BFS) and connected components (CC). Both follow a level-synchronous scheme to iteratively traverse the graph. In each iteration, the algorithm processes all vertices in the current level in parallel to find all vertices in the next level. Then, it fetches the new vertices from the disk into GPU memory, which involves data-dependent, fine-grained, and random accesses to a single large file. After that, it proceeds to the next iteration.

Figure 15 shows the end-to-end execution time of graph analysis applications. The performance trend is highly correlated with the random read throughput in §4.2. GPUfs and Basic have similar performance, as they both utilize the full host file system stack to access the disk and require `cudaMemcpy` to move data to the GPU. cuFile has the worst performance (up to 19.8× slower than Basic and GPUfs) due to the random access pattern with small I/O sizes. GoFS achieves 1.34–1.76× speedup (1.53× on average) over Basic, since it alleviates host CPU bottleneck and scales the I/O throughput with massive GPU threads. Compared to GeminiFS, GoFS is 1.2× faster, as GoFS utilizes more GPU parallelism to improve I/O throughput, which is especially useful for small random I/O accesses.

4.3.4 Graph Neural Network Training. GNN training involves three steps [23, 33]. In each training iteration, a batch is first formed by sampling multiple subgraphs from the entire graph. This step performs BFS on multiple source nodes up to a certain distance. Second, the node features for each subgraph are gathered from the disk to the GPU memory. Finally, the GPU performs DNN computation on each subgraph. The first two steps incur random I/O accesses on large files and are a major bottleneck in GNN training.

Table 2. Total execution time and speedup vs. Host-Only for dataset preprocessing.

	Host-Only	GeminiFS	GoFS
1 SSD	200.08s (1.0×)	399.18s (0.50×)	172.26s (1.16×)
2 SSDs	190.56s (1.0×)	339.40s (0.56×)	97.11s (1.96×)
4 SSDs	185.68s (1.0×)	327.93s (0.57×)	58.80s (3.16×)

Our graph dataset consists of a graph represented by a sparse matrix with per-node features and ground truth labels for training. Due to SSD capacity limitation, we striped the 3.2TB dataset across two 2TB SSDs using RAID0. We extend PyTorch Geometric (PyG) [17] for GNN training.

Figure 16 shows the training throughput and the single-batch execution time breakdown with different batch sizes. While GNN training also involves BFS, the performance benefits of GoFS over other baselines are more obvious than in the BFS benchmark (§4.3.3). This is because GoFS scales better than other baselines on two SSDs, since GoFS can exploit the GPU parallelism to maximize the bandwidth utilization of all SSDs (see §4.4). For Basic, GPUfs, and cuFile, the host filesystem stack encounters scalability issues and cannot fully utilize the bandwidth of two SSDs [49]. GoFS is 1.54× faster than GeminiFS. The gap between GeminiFS and GoFS is more obvious than in Figure 15 because it requires more GPU parallelism to utilize the bandwidth of two SSDs.

4.3.5 Dataset Preprocessing.

Dataset preprocessing is a representative GPU workload that involves intensive read/write I/O to the storage. The task involves reading the files in the dataset into the GPU, transforming them into a specific format required by the downstream tasks, and writing them back to the storage. The task is common in many real applications like ML training and data analytics. GoFS can benefit data preprocessing, as it accelerates multi-file creation, opening, and writing by managing metadata with GPU parallelism.

Table 2 compares the time of preprocessing the ImageNet dataset (see Table 1) with Host-Only (CPU preprocesses the dataset), GeminiFS (CPU converts input files to the GVDK format required by GeminiFS, opens the GVDK files, and pre-allocates output GVDK files; GPU performs computation and writes to the output files), and GoFS. Both Host-Only and GeminiFS pipeline I/O and computation. GoFS is 1.16–3.16× faster than Host-Only as it leverages GPU parallelism for metadata operations. In particular, the use of batch API improves the performance of GoFS by 1.10× on average, according to our ablation study. GeminiFS performs worse than Host-Only since it not only suffers from host file open and create overhead, but also incurs extra GVDK file conversion overhead.

4.4 Performance with Multiple SSDs

We evaluate GoFS's scalability with multiple SSDs. We use `mdadm` to create RAID0 arrays of up to 4 SSDs. On the GPU, GeminiFS and GoFS support a RAID0 array by striping the data across multiple SSDs. As shown in Figure 17, GoFS achieves linear throughput improvement for all microbenchmarks as we increase the number of SSDs. With 4 SSDs, GoFS achieves 20.4GBps/22.1GBps sequential read/write throughput. This is because GoFS exploits the massive GPU parallelism to saturate the total throughput of all four SSDs. In contrast, Basic can only achieve up to 7.8GB/s throughput with 4 SSDs due to host software overhead. GeminiFS also cannot scale beyond 2 SSDs except for sequential read/write since it only exploits warp-level parallelism.

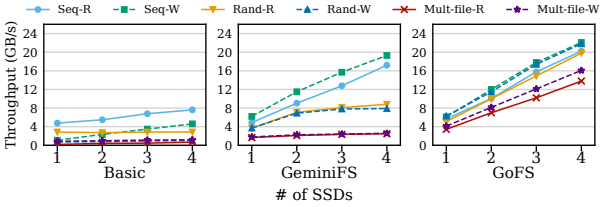


Figure 17. Performance improvement with multiple SSDs.

4.5 GPU Memory Footprint of GoFS

The major sources of GoFS's GPU memory consumption are the cached inodes and data pointers. With more opened files, GoFS needs more memory space for caching the dentry and inode structures. GoFS minimizes this using a more condensed bnode structure for batched operations. For the applications examined in our study, intelligent query with the batch size of 65,536 incurs the largest GPU memory footprint for GoFS (278 MB, of which 263 MB is the inode cache for small files). This is insignificant compared to tens of GBs of GPU memory.

5 Discussion and Future Work

Support for different file system formats. GoFS can be extended to support other on-disk file system formats. Most in-memory data structures in GoFS and the CPU-GPU coordination mechanism can be reused. However, we need to consider two factors for supporting a different file system format. First, log-structured file systems intrinsically help reduce the complexity and overhead of managing crash consistency. For file systems like ext4, extra care may need to be taken, such as the journaling implementation. Second, the concrete implementations of parallelizing on-disk data structure accesses may be different. For example, F2FS uses direct/indirect pointers (Figure 6), while ext4 uses an extent-tree structure and requires a different traversal mechanism.

As for the storage pooling support in file systems like Btrfs [61] and ZFS [10], we implemented RAID0 data sharding to manage multiple SSDs in GoFS. To support more sophisticated multi-device management functions, a similar effort is needed in GoFS, we wish to explore it in future work.

Support for GPUs from different vendors. The design of GoFS is not tied to a specific GPU vendor. GPUs from different vendors (e.g., AMD) usually share a similar architecture and programming model, because they all follow a single-instruction-multiple-data (SIMD) hardware architecture and

a hierarchical thread programming model. As long as the GPU device memory can be mapped to the BAR space on PCIe and supports PCIe P2P DMA with NVMe devices [14, 53], GoFS can be ported to the GPUs from a different vendor.

Support for multiple GPUs. As modern GPU servers are equipped with multiple GPUs to scale the computation, GoFS can be deployed on each GPU to manage its group of local SSDs. To achieve this, we can mount each group of SSDs as an individual file system instance and partition application data across these instances. As multi-GPU applications explicitly manage data and computation partitioning across GPUs (e.g., data/model parallelism in ML workloads), it requires minimal effort for them to manage data sharding across file system instances (such as duplicating or splitting files across instances). This setup is sufficient for most multi-GPU applications to exploit the bandwidth of multiple SSDs. For the rare case that requires data sharing across multiple GPUs, multi-GPU applications need to explicitly manage it upon GoFS.

As future work, we wish to extend GoFS into a distributed file system and manage data sharing across multiple GPUs in a transparent manner. For instance, each GPU runs a GoFS daemon instance, and all daemon instances synchronize to maintain data consistency.

Support for remote storage. GoFS is designed for managing scalable direct storage accesses to local SSDs. It works coordinately with remote storage systems. For input data from a remote storage, it is common for applications to cache data in local storage. For example, the TensorFlow data API [20] caches the entire or part of the dataset locally for future training epochs to avoid expensive data transfer over the network. Also, many applications demand high storage performance that cannot be satisfied by remote storage. For example, an LLM-based RAG online inference service needs to keep the RAG database in local SSDs for fast data retrieval.

Support for application-level optimizations. GoFS opens up new opportunities for GPU applications to optimize the storage access performance. We have already demonstrated example applications in §4.3 that benefit from GoFS's batched and vector APIs. GoFS can also be employed by GPU memory expander solutions for DNN training, which offloads temporally unused tensors to the SSDs [5, 59, 76]. GoFS can help better utilize the SSD offloading and prefetching bandwidth.

As discussed, GoFS supports atomic file operations, fine-grained locking mechanisms, and crash-consistency logging. These features can be leveraged to build full ACID-compliant transaction support on top of GoFS (e.g., building databases on GoFS). As future work, we wish to explore more applications and provide programming guidelines for users to utilize GoFS.

GoFS can also use GPUs to accelerate file system tasks such as data deduplication and compression. For example, existing GPU-based data deduplication algorithms show massive speedup over CPU-based ones [64]. We would like to explore them as future work as well.

6 Related Work

Scalable file systems for many-core processors. Prior studies have explored various techniques to scale file systems on CPUs [8, 12, 30, 31, 40, 49, 60]. SpanFS [30] partitioned the file system into independent domains to enable scalable data accesses. ScaleFS [8] used a per-core operation log to delay propagating updates to the disk until an `fsync`. MAX [40] reduced lock contentions for concurrent file system operations using a reader-writer semaphore. Many of them relied on concurrent data structures developed for CPUs, which cannot be easily adapted to GPUs. GoFS addresses these challenges and builds the GPU-orchestrated file system.

Direct storage access for GPUs. Prior studies have explored techniques for direct data movement between the GPU and NVMe SSDs. NVMMU [77], SPIN [7], and cuFile [66] use PCIe P2P DMA to move data between GPU and SSD. However, they still rely on the host file system to manage the SSD, such as file indexing and access control. BaM [58] places NVMe queues in the GPU device memory and allows the GPU to manage the SSD without host CPU involvement. However, it treats the SSD as a raw block device and does not have the system support. GeminiFS [57] leverages BaM to offload data I/O management to the GPU, but it still relies on the host filesystem to manage metadata. GoFS alleviates the host CPU overhead and builds a full-fledged scalable file system on the GPU.

Accelerator-centric OS. There is an ongoing trend to build accelerator-centric systems [16, 32, 58, 62, 67, 71, 73]. For instance, GPUfs [62] built a wrapper library for GPU threads to invoke host file system APIs. GPUnet [32] developed networking APIs for GPU programs. Lynx [67] and FpgaNIC [73] enabled GPU programs to offload network packet processing to SmartNICs. Genesys [71] enabled system calls for GPU programs. Our work GoFS is the first to develop a file system stack that completely runs on the GPU.

7 Conclusion

We develop a GPU-orchestrated file system named GoFS for managing scalable direct storage accesses to SSDs. GoFS enables the GPU to share the same on-disk file system with the host, but redesigns core filesystem structures that include both metadata and data I/O management with a set of optimizations for GPU-accelerated computing. GoFS outperforms current storage access solutions for GPUs by 1.61× on average for various GPU-accelerated applications.

Acknowledgments

We thank the anonymous reviewers and our shepherd Gala Yadgar for their insightful comments and feedback. We thank the members in the Systems Platform Research Group (Illinois PlatformX) at UIUC for constructive discussions. We also thank Anthony Xianyue Zhang for his help with the experiments. This work was partially supported by NSF under the grants CAREER CNS-2144796 and CCF-2107470.

References

[1] 2021. Optane SSD. https://www.intel.com/content/www/us/en/products/docs/memory-and-storage/optane-ssd/optane-ssd-overview.html.

[2] 2022. Samsung V-NAND SSD 990 PRO Datasheet Rev. 1.0. https://download.semiconductor.samsung.com/resources/data-sheet/Samsung_NVMe_SSD_990_PRO_Datasheet_Rev.1.0.pdf.

[3] Andy Adinets. [n. d.]. CUDA Dynamic Parallelism API and Principles. https://developer.nvidia.com/blog/cuda-dynamic-parallelism-api-principles/.

[4] AMD DirectGMA. [n. d.]. https://www.bitflow.com/technology/directgma/.

[5] Jonghyun Bae, Jongsung Lee, Yunho Jin, Sam Son, Shine Kim, Hakbeom Jang, Tae Jun Ham, and Jae W. Lee. 2021. FlashNeuron: SSD-Enabled Large-Batch Training of Very Deep Neural Networks. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 387–401. https://www.usenix.org/conference/fast21/presentation/bae

[6] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[7] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. 2017. SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 167–179. https://www.usenix.org/conference/atc17/technical-sessions/presentation/bergman

[8] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 69–86. doi:10.1145/3132747.3132779

[9] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[10] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. 2003. The zettabyte file system. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, Vol. 215. 1.

[11] Fedor Borisyuk, Albert Gordo, and Viswanath Sivakumar. 2018. Rosetta: Large Scale System for Text Detection and Recognition in Images. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18)*. London, United Kingdom.

[12] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. 2021. Scalable Persistent Memory File System with Kernel-Userspace Collaboration. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 81–95. https://www.usenix.org/conference/fast21/presentation/chen-youmin

[13] Vijay Chidambaram, Thanu S. Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic Crash Consistency. In *The 24th ACM Symposium on Operating System Principles (SOSP'13)*. Farmington, PA.

[14] Advanced Micro Devices. 2025. Troubleshoot BAR access limitation. https://rocm.docs.amd.com/en/latest/how-to/Bar-Memory.html.

[15] Ming Du, Arnau Ramisa, Amit Kumar K C, Sampath Chanda, Mengjiao Wang, Neelakandan Rajesh, Shasha Li, Yingchuan Hu, Tao Zhou, Nagashri Lakshminarayana, Son Tran, and Doug Gray. 2022. Amazon Shop the Look: A Visual Search System for Fashion and Home. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Washington DC, USA) *(KDD '22)*. Association for Computing Machinery, New York, NY, USA, 2822–2830. doi:10.1145/3534678.3539071

[16] Haggai Eran, Maxim Fudim, Gabi Malka, Gal Shalom, Noam Cohen, Amit Hermony, Dotan Levi, Liran Liss, and Mark Silberstein. 2022. FlexDriver: a network driver for your accelerator. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 1115–1129. doi:10.1145/3503222.3507776

[17] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. https://arxiv.org/abs/1903.02428

[18] FIO Benchmarks. [n. d.]. https://linux.die.net/man/1/fio.

[19] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. 2024. Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv:2312.10997 [cs.CL] https://arxiv.org/abs/2312.10997

[20] Google. 2024. Better performance with the tf.data API | TensorFlow Core. https://www.tensorflow.org/guide/data_performance.

[21] GPUDirect Storage: A Direct Path Between Storage and GPU Memory. [n. d.]. https://developer.nvidia.com/blog/gpudirect-storage/.

[22] M Hadi Kiapour, Xufeng Han, Svetlana Lazebnik, Alexander C Berg, and Tamara L Berg. 2015. Where to Buy It: Matching Street Clothing Photos in Online Shops. In *Proceedings of the IEEE international conference on computer vision (ICCV'15)*. Santiago, Chile.

[23] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[24] Di Hu, Zheng Wang, Haoyi Xiong, Dong Wang, Feiping Nie, and Dejing Dou. 2020. Curriculum audiovisual learning. *arXiv preprint arXiv:2001.09414* (2020).

[25] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. 2021. Ogb-lsc: A large-scale challenge for machine learning on graphs. *arXiv preprint arXiv:2103.09430* (2021).

[26] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of Machine Learning and Systems (MLSys'20)*.

[27] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/09853c7fb1d3f8ee67a61b6bf4a7f8e6-Paper.pdf

[28] Yushi Jing, David Liu, Dmitry Kislyuk, Andrew Zhai, Jiajing Xu, Jeff Donahue, and Sarah Tavel. 2015. Visual Search at Pinterest. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'15)*. Sydney, Australia.

[29] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734* (2017).

[30] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. 2015. SpanFS: A Scalable File System on Fast Storage Devices. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 249–261. https://www.usenix.org/conference/atc15/technical-session/presentation/kang

[31] Dohyun Kim, Kwangwon Min, Joontaek Oh, and Youjip Won. 2022. ScaleXFS: Getting scalability of XFS back on the ring. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 329–344. https://www.usenix.org/conference/fast22/presentation/kim-dohyun

[32] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. 2014. GPUnet: Networking Abstractions for GPU Programs. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 201–216. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kim

[33] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[34] Alex Kogan, Dave Dice, and Shady Issa. 2020. Scalable range locks for scalable address spaces and beyond. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*. Heraklion, Greece.

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*.

[36] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

[37] Changman Lee, Dongbo Sim, Joo-Young Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. Santa Clara, CA.

[38] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2021. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv:2005.11401 [cs.CL] https://arxiv.org/abs/2005.11401

[39] Shuying Liang. 2024. Simple is beautiful: Revolutionizing GNN Training Infrastructure at LinkedIn. https://flyte.org/case-study/simple-is-beautiful-revolutionizing-gnn-training-infrastructure-at-linkedin.

[40] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. 2021. Max: A Multicore-Accelerated File System for Flash Storage. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 877–891. https://www.usenix.org/conference/atc21/presentation/liao

[41] Yuan Lin and Vinod Grover. 2018. Using CUDA Warp-Level Primitives. https://developer.nvidia.com/blog/using-cuda-warp-level-primitives.

[42] Rui Lu, Kailun Wu, Zhiyao Duan, and Changshui Zhang. 2017. Deep ranking: Triplet MatchNet for music metric learning. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 121–125.

[43] Vikram Sharma Mailthoday, Zaid Qureshi, Weixin Liang, Ziyan Feng, Simon Garcia de Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wen mei Hwu. 2019. DeepStore: In-Storage Acceleration for Intelligent Queries. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*. Columbus, OH.

[44] Jonas Markussen, Lars Bjørlykke Kristiansen, Pål Halvorsen, Halvor Kielland-Gyrud, Håkon Kvale Stensland, and Carsten Griwodz. 2021. SmartIO: Zero-Overhead Device Sharing through PCIe Networking. *ACM Transactions on Computer Systems* 38, 1–2, Article 2 (jul 2021), 78 pages. doi:10.1145/3462545

[45] Daniel Mawhirter and Bo Wu. 2019. AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*. Huntsville, Ontario, Canada.

[46] P. E. McKenney, D. Sarma, and M. Soni. 2004. Scaling dcache with RCU. *Linux Journal* (2004).

[47] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2015. High-Performance and Scalable GPU Graph Traversal. *ACM Trans. Parallel Comput.* 1, 2, Article 14 (Feb. 2015), 30 pages. doi:10.1145/2717511

[48] Microsoft. 2025. Support boundary for high accuracy time - Windows Server | Microsoft Learn. https://learn.microsoft.com/en-us/troubleshoot/windows-server/active-directory/support-boundary-high-accuracy-time.

[49] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. 2016. Understanding Manycore Scalability of File Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 71–85. https://www.usenix.org/conference/atc16/technical-sessions/presentation/min

[50] National Institute of Standards and Technology (US). 2008. *The keyed-hash message authentication code (HMAC)*. Technical Report. Washington, D.C.

[51] NVIDIA. 2024. Multi-Process Service r555 documentation. https://docs.nvidia.com/deploy/mps/index.html.

[52] NVIDIA. 2024. Time Library – libcudacxx 2.5 documentation. https://nvidia.github.io/cccl/libcudacxx/standard_api/time_library.html.

[53] NVIDIA. 2025. 1. Overview — GPUDirect RDMA 13.0 documentation. https://docs.nvidia.com/cuda/gpudirect-rdma/.

[54] NVIDIA. 2025. Field Identifiers — NVIDIA DCGM Documentation latest documentation. https://docs.nvidia.com/datacenter/dcgm/latest/dcgm-api/dcgm-api-field-ids.html.

[55] NVIDIA Corporation. 2018. NVIDIA CUDA C Programming Guide.

[56] NVIDIA Magnum IO GPUDirect Stoage CuFile API. [n. d.]. https://docs.nvidia.com/gpudirect-storage/pdf/api-reference-guide.pdf.

[57] Shi Qiu, Weinan Liu, Yifan Hu, Jianqin Yan, Zhirong Shen, Xin Yao, Renhai Chen, Gong Zhang, and Yiming Zhang. 2025. GeminiFS: A Companion File System for GPUs. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. USENIX Association, Santa Clara, CA, 221–236. https://www.usenix.org/conference/fast25/presentation/qiu

[58] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, C. J. Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. 2023. GPU-Initiated On-Demand High-Throughput Storage Access in the BaM System Architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 325–339. doi:10.1145/3575693.3575748

[59] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. arXiv:2104.07857 [cs.DC] https://arxiv.org/abs/2104.07857

[60] Yujie Ren, Changwoo Min, and Sudarsun Kannan. 2020. CrossFS: A Cross-layered Direct-Access File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 137–154. https://www.usenix.org/conference/osdi20/presentation/ren

[61] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-Tree Filesystem. *ACM Trans. Storage* 9, 3, Article 9 (Aug. 2013), 32 pages. doi:10.1145/2501620.2501623

[62] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: integrating file systems with GPUs. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. Houston, Texas, USA.

[63] Cooper Smith. 2013. Facebook users are uploading 350 million new photos each day. *Business insider* 18 (2013).

[64] Youngjun Son, Chaewon Kim, and Jaejin Lee. 2025. FED: Fast and Efficient Dataset Deduplication Framework with GPU Acceleration. arXiv:2501.01046 [cs.CL] https://arxiv.org/abs/2501.01046

[65] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A flexible framework for file system benchmarking. *The USENIX Magazine* 41, 1 (2016).

[66] Adam Thompson and CJ Newburn. 2019. GPUDirect Storage: A Direct Path Between Storage and GPU Memory. https://developer.nvidia.com/blog/gpudirect-storage/.

[67] Maroun Tork, Lina Maudlej, and Mark Silberstein. 2020. Lynx: A SmartNIC-driven Accelerator-centric Architecture for Network Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 117–131. doi:10.1145/3373376.3378528

[68] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[69] Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E. Porter. 2015. How to get more value from your file system directory cache. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 441–456. doi:10.1145/2815400.2815405

[70] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. Santa Clara, CA.

[71] Ján Veselý, Arkaprava Basu, Abhishek Bhattacharjee, Gabriel H. Loh, Mark Oskin, and Steven K. Reinhardt. 2018. Generic System Calls for GPUs. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. 843–856. doi:10.1109/ISCA.2018.00075

[72] Liwei Wang, Yin Li, Jing Huang, and Svetlana Lazebnik. 2018. Learning Two-branch Neural Networks for Image-text Matching Tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2018).

[73] Zeke Wang, Hongjing Huang, Jie Zhang, Fei Wu, and Gustavo Alonso. 2022. FpgaNIC: An FPGA-based Versatile 100Gb SmartNIC for GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 967–986. https://www.usenix.org/conference/atc22/presentation/wang-zeke

[74] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-Volatile Main Memories. In *The 14th USENIX Conference on File and Storage Technologies (FAST'16)*. Santa Clara, CA.

[75] Fisher Yu, Yinda Zhang, Shuran Song, Ari Seff, and Jianxiong Xiao. 2015. LSUN: Construction of a Large-scale Image Dataset using Deep Learning with Humans in the Loop. *arXiv preprint arXiv:1506.03365* (2015).

[76] Haoyang Zhang, Yirui Zhou, Yuqi Xue, Yiqi Liu, and Jian Huang. 2023. G10: Enabling An Efficient Unified GPU Memory and Storage Architecture with Smart Tensor Migrations. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*. Toronto, ON, Canada.

[77] Jie Zhang, David Donofrio, John Shalf, Mahmut T. Kandemir, and Myoungsoo Jung. 2015. NVMMU: A Non-volatile Memory Management Unit for Heterogeneous GPU-SSD Architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 13–24. doi:10.1109/PACT.2015.43

AutoOverlap: Enabling Fine-Grained Overlap of Computation and Communication with Chunk-Based Scheduling

Xinwei Qiang[1], Yue Guan[1], Zhengding Hu[1], Yufei Ding[1,2], Adnan Aziz[2]
[1]*University of California, San Diego,* [2]*Meta*
[1]*{x1qiang, yueguan, zhh068 yufeiding}@ucsd.edu*
[2]*{adnanaziz}@meta.com*

Abstract

Communication has become a first-order bottleneck in large-scale GPU workloads, and existing distributed compilers address it mainly by overlapping whole compute and communication kernels at the stream level. This coarse granularity incurs extra kernel launches, forces device-wide synchronizations at kernel boundaries, and leaves substantial slack when the slowest tile or kernel stretches the communication tail. We present AutoOverlap, a compiler and runtime that enable automatic fine-grained overlap inside a single fused kernel. AutoOverlap introduces a communication chunk abstraction that decouples communication granularity from kernel structure and backend mechanisms, allowing chunk-level plans to be ported from existing distributed compilers, written directly by users, or instantiated from reusable templates. Given a local Triton kernel and a chunk schedule, AutoOverlap performs transformations to align computation with chunk availability. Implemented as a source-to-source compiler on Triton, AutoOverlap delivers an average end-to-end speedup of 1.3× and up to 4.7× on multi-GPU workloads.

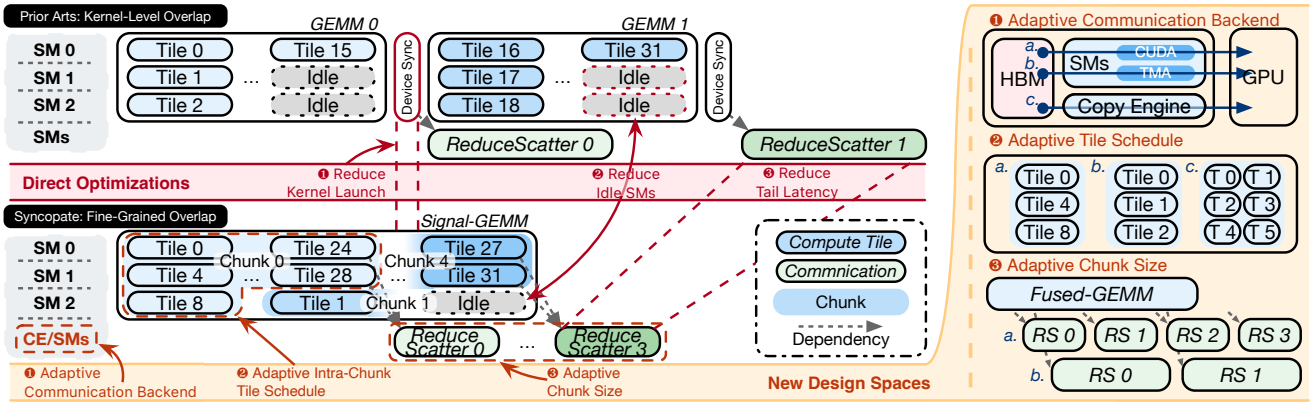
1 Introduction

Communication has become a first-order bottleneck for training and serving large neural networks on multi-GPU systems. Even with high-bandwidth interconnects such as NVLink [21] and NVSwitch, collective operations like AllGather, ReduceScatter, and All-to-All frequently dominate end-to-end latency for tensor-parallel feed-forward layers and attention layers. To hide this cost, recent systems and distributed compilers aggressively search for schedules that overlap computation and communication at the kernel level. Given a computation graph and a device mesh, these compilers select parallelization strategies, insert communication kernels, and assign compute and communication kernels to streams so that multiple kernels can run concurrently. This kernel-level overlap [4, 10, 29, 34, 37, 47] has become the default mechanism for improving utilization in distributed settings.

However, kernel-level overlap is fundamentally insufficient for fully hiding communication latency. As illustrated in Fig. 1, this kernel-level scheduling forces a device-wide synchronization at every kernel boundary and incurs extra launch and sync overhead for each communication phase (❶). Moreover, by splitting computation into multiple shorter kernels, the work within each launch is further partitioned into waves of compute tiles on each SM, increasing the fraction of time SMs sit idle; even when part of the data needed by later kernels is already available, tiles in the current wave must wait for the slowest tile to finish (❷). Finally, the coarse granularity of kernel-level overlap leaves a long segment of communication at the end of the timeline that receives little or no overlap (❸).

This motivates us to overlap computation and communication at a finer intra-kernel granularity. Such fine-grained overlap opens up new design space for improving end-to-end efficiency. As shown by the yellow region in Fig. 1, AutoOverlap launches communication directly from within the fused kernel, rather than delegating to external communication libraries, giving the compiler explicit control over which hardware backend to use for each transfer (copy engine, tensor memory accelerator, or load/store on CUDA cores) as shown with ❶. This also allows us to explore much smaller communication granularities without incurring additional kernel-launch overhead, and to tune the chunk size that best balances link throughput against synchronization cost (❷). Finally, because tiles and communication are orchestrated inside a single kernel, we can reshape the intra-kernel tile schedule to track communication progress while still preserving locality in the register, shared-memory, and cache hierarchy (❸).

We present AutoOverlap, a compiler that turns the vision of fine-grained overlap for distributed kernel generation into a practical and general system. At the heart of AutoOverlap is the notion of a communication chunk, which represents a logical block of data associated with a particular communication operation and the tiles that produce or consume it. This abstraction is motivated by a key observation: fine-grained overlap requires a communication granularity that flexibly matches how tiles generate data and how communication



(a) Comparison of compute–communication overlap at different levels (b) New design space for fine–grained overlap

Figure 1: Motivating example of AutoOverlap. Red numbers shows the direct improvements gained by fine-grained overlap over kernel-level overlap, while orange numbers show the additional improvements from the new design space enabled by AutoOverlap.

backends consume it, rather than assuming that tile-level or full-kernel granularity is always appropriate. By making chunks explicit, AutoOverlap can represent a wide range of overlap patterns and enable the compiler to reason about when each chunk becomes available, which tiles produce or use it, and how these dependencies interact with fused multi-stage tensor programs and potentially irregular collectives. This abstraction exposes a small set of principled knobs, including chunk size, backend choice, and tile order, that define the design space later explored by our autotuner.

Building on this abstraction, AutoOverlap implements a source-to-source compiler and runtime that transform standard Triton [32] kernels and a high-level chunk-based communication plan into fused distributed kernels capable of fine-grained compute–communication overlap. The compiler restructures the kernel's tile execution to follow communication progress and selects appropriate backends to realize each chunk transfer, while a lightweight runtime executes these transfers and integrates seamlessly with PyTorch distributed [2] so that AutoOverlap kernels can replace standard operators with only minimal changes to user code. AutoOverlap further performs inter- and intra-chunk autotuning, adjusting chunk sizes, backend choices, SM allocations, and tile schedules, to consistently achieve high performance across diverse operators. This implementation strategy makes the abstract chunk-based model concrete, enabling rapid prototyping of new overlap policies while remaining compatible with real distributed workloads and production communication stacks.

In summary, this work makes the following contributions:

- We introduce a chunk-based abstraction that decouples high-level overlap intent from low-level implementation, enabling fine-grained compute-communication overlap inside distributed kernels.
- We design and implement a compiler pipeline and runtime that takes annotated local kernels and chunk-level communication plans, then generates efficient fused distributed kernels with inter- and intra-chunk autotuning.
- We evaluate AutoOverlap on a diverse set of distributed workloads and observe an average speedup of 1.3× on common operators, with improvements reaching up to 4.7× in the best cases.

2 Background and Related Works

2.1 Distributed Compilers

As model sizes grew, tensor compilers [5, 6, 9, 27, 39, 42, 43] incorporated basic multi-GPU support [1, 16, 25, 28, 35, 40], typically by composing single-GPU kernels with predefined collective primitives. Systems like Alpa [44], Mercury [13], and other recent distributed compilers extend this idea by searching over communication patterns and schedules at the level of whole kernels: given a parallelization strategy, they first construct a communication plan containing AllGather, ReduceScatter, or All-to-All operators, and then explore different mappings of compute and communication kernels to streams in order to maximize kernel-level overlap (Table 1). This design has made distributed training far more accessible, but it also bakes in a rigid abstraction boundary: communication is planned as a sequence of full-kernel collectives whose launch and completion times are the basic units of scheduling. As a result, all these distributed compilers focus their search on the communication plan at the kernel level and are fundamentally blind to finer-grained opportunities inside kernels, such as overlapping per-shard or per-tile communication with computation, reusing remote data across tiles, or exploiting topology-aware pipelining within a fused kernel.

Table 1: Comparison of various projects on distributed operations.

Projects	Granularity	Compute	Communication	Schedule	Performance
Automatic Approaches					
Alpa [44]	Kernel	Auto	Auto	Template	✔
Mercury [13]	Kernel	Auto	Auto	Auto	✔✔
Manual Implementations					
Flux [3]	Tile	Manual	Manual	Manual	✔✔
AsyncTP [36]	Tile	Manual	Manual	Manual	✔✔
FlashOverlap [14]	Chunk	Manual	Manual	Manual	✔✔✔
Domain Specific Languages					
ThunderKittens [30, 31]	Tile	Manual	Manual	Manual	✔✔✔
TritonDistributed [45, 46]	Chunk	Manual	Manual	Manual	✔✔✔
AutoOverlap	Chunk	Auto	Auto	Template	✔✔✔

Once a kernel is chosen and its associated collective is placed, the compiler can only treat it as an atomic black box, leaving significant intra-kernel overlap potential untapped even under an "optimal" kernel-level schedule.

2.2 Mannual Kernel Designs

A complementary line of work pushes beyond kernel-level overlap and instead hand-crafts fine-grained pipelines that interleave communication and computation at the level of tiles, tokens, or shards. Flux [3] fuses GEMM with collectives at tile granularity and over-decomposes work to maximize overlap for both training and inference, while Comet [41] targets MoE with a shared-tensor abstraction and NVSHMEM-backed buffers to overlap token-wise communication with tile-wise compute at production scale. FlashOverlap [14] uses lightweight readiness signaling and layout reordering to trigger overlap with standard NCCL collectives without modifying existing compute kernels, and systems like Triton-Distributed [45, 46] introduce tile-centric or OpenSHMEM-style primitives to Triton [32] that make it easier to author overlapped kernels (e.g., fused AllGather/GEMM or GEMM/ReduceScatter) in a domain-specific language.

Despite these advances, all of these systems fundamentally rely on manual, operator-specific engineering: experts must design fused kernels, choose tiling and buffering schemes, and reason about synchronization and communication protocols for each new model architecture or hardware platform. The resulting implementations are highly optimized but difficult to generalize or retarget, and they do not provide a general compiler abstraction for expressing and reusing fine-grained overlap patterns across operators. Emerging fine-grained DSLs and primitives greatly lower the barrier to writing overlapped kernels, but they still place the burden of discovering effective overlap strategies, encoding dependency structure, and validating correctness squarely on the programmer.

2.3 Communication Backends

Modern GPU systems [19] expose several mechanisms for moving tensors across devices, each with distinct performance and programmability trade-offs (Tbl. 2). Copy Engines saturate NVLink [21] at 400 GB/s per direction on H100 and run independently of the SMs. Therefore, they do not consume compute resources and are ideal when communication can be decoupled from computation. However, they are typically driven by host APIs and can only transfer contiguous data, so high-dimensional strided tensors must be decomposed into many smaller transfers, each requiring a separate launch costing around 2-3μs, which can significantly reduce the effective bandwidth as each transfer time is also very short.

Table 2: Comparison of various GPU communication mechanisms.

	Hardware	Programming	Collective	Bandwidth
Copy Engine	Copy Engine	Host Launch	✗	✔✔✔
TMA	SM	Async. Instruction	✗	✔✔
Load/Store	SM	Sync. Instruction	✔	✔

Tensor Memory Accelerator (TMA) [19] paths achieve high bandwidth using dedicated asynchronous tensor-copy hardware, and can achieve a throughput of 300+ GB/s with only about 16 SMs issuing TMA instructions. This makes TMA attractive for overlapping structured tensor movement with computation within a node. At the same time, TMA must be launched by SM threads and does not currently support inter-node communication or in-network (switch-based) collective reduction, which limits its applicability to intra-node, point-to-point patterns.

Finally, plain load/store-based communication, often combined with registers and shared memory, attains slightly lower peak bandwidth than copy engines or TMA [31] but is significantly more flexible. These mechanisms integrate naturally with switch-based collective reduction (NVSHARP [20]), enabling fine-grained per-shard communication and in-network reductions. The downside is that they consume SM resources and are synchronous from the issuing warp's perspective, so they are harder to pipeline and require careful scheduling to hide communication latency.

3 Motivation

To fully exploit modern GPUs, distributed training systems must overlap computation and communication. Prior work mainly relies on coarse, kernel-level partitioning of computation kernels, which leaves substantial performance on the table. In this section, we revisit the motivating example in Figure 1 using detailed microbenchmarks (Figure 2), and show three key insights:

> ★***Insight 1: Limitations of Kernel-Level Overlap.*** *Kernel-level overlap is fundamentally limited by SM under-utilization and kernel-launch overheads.*

Figure 2(a) reports SM utilization for different GEMM

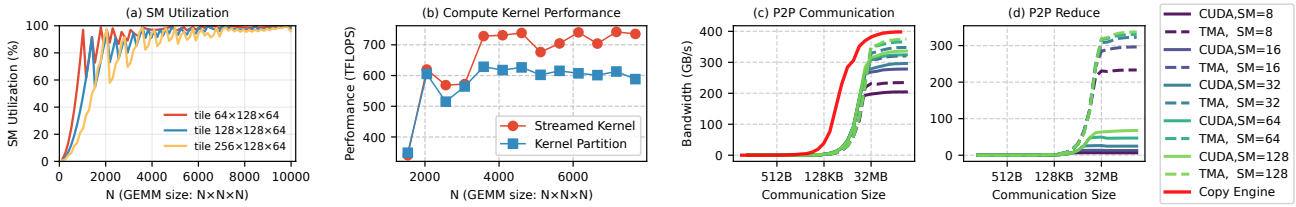


Figure 2: Motivation experiment results. (a) SM utilization under different GEMM sizes and tile sizes. (b) Performance comparison between a streamed GEMM kernel and a kernel-partitioned baseline. (c,d) Bandwidth of different communication backends under varying message sizes.

sizes under several commonly used tile-size configurations. Large GEMMs provide enough tile waves to saturate the SMs across all configurations. As the GEMM size decreases, fewer tile waves are generated, and the partially filled last wave dominates a larger portion of execution, causing SM utilization to drop due to wave quantization. Partitioning a GEMM into many sub-kernels forces each launch to operate on a smaller shape, pushing execution into exactly this low-utilization regime. This effect directly limits the benefit of kernel-level partition-based overlap (corresponding to ❷ in Figure 1), since overlapping many small kernels simply wastes SM capacity.

Figure 2(b) compares the end-to-end performance of (i) a baseline that partitions GEMM into multiple small kernels for overlap and (ii) a streamed GEMM kernel that internally pipelines tiles. Although both variants execute the same arithmetic operations, the kernel-partitioned baseline incurs substantial performance loss due to extra kernel launches (❶) and the SM under-utilization observed in Figure 2(a). In contrast, the streamed kernel maintains high utilization by exposing fine-grained compute tiles within a single launch, enabling overlap without fragmenting the workload. These results show that simply launching more kernels is not an effective path toward overlap; we need mechanisms that expose intra-kernel concurrency while preserving efficient GPU execution.

> ★***Insight 2: Granularity and Backend Effects.*** *Communication efficiency varies sharply with transfer granularity and backend selection.*

Communication efficiency varies sharply with transfer granularity and backend behavior. Figures 2(c) and 2(d) show the achieved bandwidth of different communication backends as we vary the transfer size and the number of SMs. Each backend shows distinct scaling behavior: some reach peak bandwidth at moderate transfer sizes, while others require larger transfers or more SMs to reach their full potential. Moreover, different backends support different communication patterns, such as point-to-point transfers versus reductions.

Taken together, these granularity and backend effects imply that the optimal configuration depends jointly on (i) the compute tile size, which determines the cadence at which results become available, (ii) the communication transfer size, which trades off latency and bandwidth, and (iii) the choice of communication backend. Coarse-grained kernel-level overlap cannot flexibly coordinate these parameters, since compute and communication are implemented as separate kernels with rigid interfaces. Intra-kernel overlap, by coordinating computation and communication at the same time, can align tile production with backend-specific sweet spots and select transfer granularities that sustain high utilization across SMs and copy engines.

> ★***Insight 3: A Unified Unit for Intra-Kernel Overlap.*** *Effective intra-kernel overlap therefore requires a communication unit with tunable granularity and a stable interface across communication backends.*

The combined granularity and backend effects indicate that effective overlap requires a communication unit whose size can match both the rate at which tiles produce data and the efficiency points of different communication backends. At the same time, this unit must provide a stable boundary between the high-level communication schedule and its backend-specific realization, so that schedules need not be rewritten for each backend. We therefore introduce a chunk abstraction that offers both tunable intra-kernel granularity and a unified interface for mapping communication onto diverse backends.

4 Overview

AutoOverlap is a compiler and runtime framework that turns locally written Triton kernels into distributed, fine-grained overlapped kernels. Rather than asking programmers to manually fuse communication and computation, AutoOverlap takes an existing local kernel and a high-level distribution specification as input, and automatically synthesizes an intra-kernel schedule that interleaves tile-wise communication with computation according to the available communication backends.

Input and User Interface. On the compute side, AutoOverlap consumes unmodified local Triton kernels annotated with

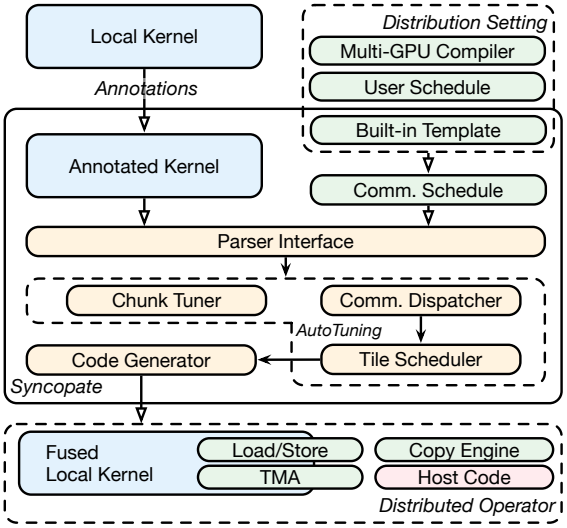


Figure 3: System overview of AutoOverlap.

Listing 1: Annotated Local Triton Kernel API.

```
@triton.jit
def kernel_gemm(a_ptr, b_ptr, ...):
    start_pid = tl.program_id(axis=0)
    # @sy.axis_count M block=BLOCK_SIZE_M
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    # @sy.tile_id persistent
    tile_id = start_pid - NUM_SMS
    ...
    a_desc = tl.make_tensor_descriptor(a_ptr, ...)
    ...
    for _ in range(0, k_tiles * tiles_per_SM):
        tile_id += NUM_SMS
        # @sy.dispatch begin
        # @sy.pid_map M=pid_m N=pid_n
        pid_m, pid_n = get_pid_mn(tile_id,
                                  num_pid_m, ...)
        # @sy.dispatch end
        offs_am = pid_m * BLOCK_SIZE_M
        offs_bn = pid_n * BLOCK_SIZE_N
        offs_k = ki * BLOCK_SIZE_K
        a = a_desc.load([offs_am, offs_k])
        b = b_desc.load([offs_bn, offs_k])
        accumulator = tl.dot(a, b.T, accumulator)
```

Listing 2: Communication Schedule Example.

```
def all_gather_1d_swizzle(shape,dtype,axis,rank,...):
    plan = DevicePlan(dev=rank)
    plan.tensors_involved[buf] = (torch.Size(shape))
    local = shard(rank)
    plan.local_regions.setdefault(buf,[]).append(local)
    for i in range(mesh):
        peer = (i + rank) % mesh  # 1D swizzle
        if peer == rank: continue
        r = shard(peer)
        plan.add_op(Transfer(
                        op=TransferOp.PULL,
                        dst_buf=buf, dst_region=r,
                        src_buf=buf, src_region=r,
                        peer=peer, shard_idx=peer,
                    ...))
    return plan
```

lightweight scheduling metadata (Listing 1). Programmers write kernels as if they were running on a single device, using standard Triton primitives for indexing, tiling, and tensor descriptors. Optional AutoOverlap annotations (e.g., axis counts, tile identifiers, and dispatch regions) identify the logical tiles and iteration structure but do not change the kernel's semantics. On the distribution side, AutoOverlap uses a communication plan that encodes the desired global data movement pattern and device topology (Listing 2). This plan is expressed using a small API defined by the users or imported directly from higher-level compilers searching parallel schedule.

To end users, AutoOverlap exposes a small set of APIs for (1) registering local kernels with their annotations, (2) constructing or selecting predefined communication plans (such as 1D/2D AllGather or ReduceScatter swizzles), and (3) compiling these into executable distributed kernels. High-level frameworks can wrap these APIs so that model authors only specify tensor partitioning and desired collectives, while AutoOverlap automatically generates the corresponding overlapped kernels. This separation of concerns allows experts to encode distribution and communication strategies once, while ordinary users invoke the resulting distributed operators through familiar, library-style calls.

AutoOverlap Architecture and Output. Given a local kernel and its communication plan, AutoOverlap lowers them into a unified dependence representation over tiles, shards, and communication operations. The compiler then searches for a fine-grained schedule that maps tiles to devices, assigns communication operations to appropriate backends (e.g., copy engine, TMA, or load/store), and inserts the necessary synchronization to respect both compute and communication dependencies. The output is a distributed Triton kernel that preserves the original numerical semantics but now issues asynchronous communication and computation in a tightly pipelined manner inside a single fused kernel. From the user's perspective, this kernel can be invoked with the same signature as the original local kernel, plus standard distributed-runtime arguments (e.g., rank, world size, mesh), and integrated into existing training or inference code without further changes.

5 AutoOverlap: Fine-Grained Overlap Compiler

AutoOverlap automatically transforms locally written kernels into fine-grained overlapped distributed kernels by aligning the execution of local computation tiles with a global communication schedule. At a high level, we introduce a *chunk-centric* compilation pipeline that treats communication and computation symmetrically: communication is described as transfers of logical chunks, while computation is expressed as tiles that consume and produce these chunks. The com-

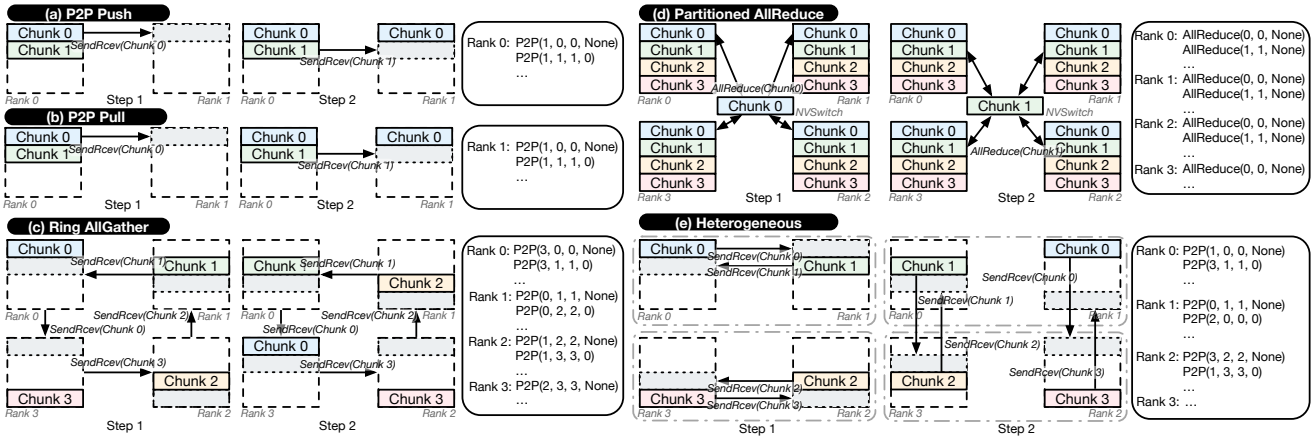


Figure 4: Communication schedule abstraction. (a) and (b) illustrate the same point-to-point exchange expressed as push and pull variants, respectively. (c) shows a ring-based AllGather pattern. (d) represents a partition-based AllReduce schedule. (e) depicts a heterogeneous swizzled AllGather pattern that pipelines communication across multiple hierarchy levels.

piler then derives dependencies between chunks and tiles, rewrites the tile scheduler to follow the communication order, inserts the necessary synchronization, and finally explores several implementation choices to generate high-performance overlapped code.

5.1 Communication Schedule Abstraction

We first define a communication-side abstraction that captures how data moves across devices independently of any particular local kernel implementation. This abstraction is built around the notion of a *chunk*, an intermediate layout between the global logical tensor and the local computation tiles. By operating at this intermediate granularity, the abstraction is expressive enough to describe a wide range of distributed schedules while remaining compatible with both partition-based and loop-based IRs in existing distributed compilers.

Definition. A *chunk* is a logical block of data that is communicated as a unit. Each chunk contains one or more tiles, where a tile is the basic unit of computation in the local kernel. Importantly, the chunk size in the communication schedule specifies *logical* transfers; the same logical chunk may later be implemented using different physical communication patterns or backends during lowering.

Conceptually, a chunk can be represented as: `chunk = Chunk(sizes=[...] , layout=..., tensor=...)`. Based on this abstraction, we define communication operators over chunks. We consider two primary classes of operators: point-to-point (P2P) transfers and collective communications.

- **P2P transfer** is represented as `P2P(src_rank, dst_rank, src_chunk, dst_chunk, dependency)`. This operator moves a chunk from a source rank to a destination rank, optionally guarded by a dependency on other chunks or operations. Note that for a pair of P2P operations on the source and destination ranks, we only include the operation on one side. If the P2P operation is defined on the source side, it represents a push operation; otherwise, it represents a pull operation. This will lead to different implementation choices during lowering.
- **Collective communication** is represented as `Collective(collective_type, src_chunk, dst_chunk, ranks, dependency)`. This operator applies a collective operation (e.g., AllGather, ReduceScatter) over a set of ranks on a given chunk with explicit dependency control. When explicitly defined as collective operations, the compiler can leverage the optimized collective implementations provided by the communication backends.

For both operator types, the `dependency` field encodes any ordering constraints that must be respected between communication operations. In specific, it is represented as a `(rank, index)` tuple, indicating that the current operation cannot start until the specified operation on the given rank has completed. This allows us to express complex communication patterns, such as ring exchanges or multi-stage collectives, by chaining dependencies between chunks.

Upon this abstraction, a *communication schedule* is defined as a sequence of chunk-level communication operations with their associated dependencies on each rank as `schedule := [rank:Int, operations:List[CommOp]]:List`. Since there is no restriction on the operation for each rank, the communication schedule can express heterogeneous communication patterns where different ranks perform different operations on different chunks at different times.

Expressiveness. Despite its simplicity, the chunk abstraction is sufficiently expressive to capture a wide range of communication schedules covering all the overlap patterns used in practice (Fig. 4). To elaborate, (a) and (b) show the same P2P communication between two ranks expressed as push and

pull operations, respectively, demonstrating the flexibility of pull/push semantics. (c) illustrates a ring-based AllGather pattern, which is a common pattern for asynchronous distributed operators [18], where each rank sends and receives chunks in a pipelined manner, with dependencies ensuring the correct order of operations. (d) depicts a partition-based collective AllReduce pattern, where each rank contributes a chunk to the collective operation and performs the accumulation on the fibre. This pattern is often used in partition-based distributed compilers for kernel-level overlap. Finally, (e) shows a complex heterogeneous swizzled AllGather pattern advancing (c). By utilizing the port abstraction, each rank processes communication at different mesh hierarchy levels, enabling fine-grained pipelining and overlap across multiple dimensions. With this abstraction, different collective patterns, pipelined P2P exchanges, and hybrid schemes that combine intra-node and inter-node communication can all be written as sequences of chunk-level P2P and collective operators with explicit dependencies. Because chunks are defined in terms of logical tensor regions rather than concrete buffers, the same schedule can be reused across different kernels and tensor shapes, and later specialized by the compiler.

Lowering from Higher-Level Compiler IRs. Communication schedules in this abstraction can either be defined manually or automatically derived from existing distributed compiler IRs. In the manual case, users construct chunk objects and communication operators directly using our API as shown in Listing 2. AutoOverlap also provides pre-defined templates such as 1D/2D AllGather or ReduceScatter swizzles for the common communication patterns. User can instantiate these templates with different chunk sizes, mesh topologies, communication axes, and pipeline stages to generate reusable communication schedules.

When integrating with a higher-level distributed compiler, AutoOverlap provides frontends for both partition-based and loop-based IRs. For partition-based IRs, we analyze the global data partitioning and the implied communication pattern between partitions to infer chunk sizes, participating ranks, and the corresponding P2P or collective operators. For loop-based IRs, we traverse loop nests, identify communication points, and group the communicated regions into chunks according to the chosen granularity. In both cases, the result is a uniform chunk-level schedule that decouples the high-level communication intent from any particular implementation (Listing 3). Specifically, the collective operators can be directly inserted ("direct") into our communication plan, or they can be further lowered to P2P communication operators using our templates ("template") or using some collective synthesis algorithms ("synth") such as TACOS [38].

Listing 3: Lowering from Higher-Level IR.

```
def emit_steps(steps, mesh, path="template"):
    comm = CommPlan()
    for step in steps:
        if step.is_p2p(): # push/pull/local_copy
            for rank in mesh:
                emit_p2p(comm.plans[rank], step, rank)
        else:  # collective
            for rank in mesh:
            # Path in ["direct", "synth", "template"]
                emit_collective(path, comm.plans[rank],
      step, rank)
    return comm

def lower_partition_ir(part_ir, axis_info, mesh, path="
     template"):
    steps = []
    for tensor in part_ir.tensors:
        layout = part_ir.placement[tensor]
        meta = axis_info[tensor]
        steps.extend(parse_partition_to_steps(tensor,
     layout, meta))
    return emit_steps(steps, mesh, path)

def lower_loop_ir(loop_ir, mesh, path="template"):
    steps = []
    for node in walk(loop_ir):
        steps.extend(parse_comm_intents(node))
    return emit_steps(steps, mesh, path)
```

5.2 Chunk-based Code Generation

Given a chunk-level communication schedule, the next step is to reorganize local computation so that tiles are executed in an order that aligns with the arrival and consumption of chunks. Intuitively, we want the kernel to compute exactly those tiles whose data has already been communicated, and to defer tiles whose input chunks are still in flight. This *compute chunk scheduling* bridges the gap between the communication abstraction and the tile-level structure of the original local kernel.

Compute Kernel Annotations. To make tile-level scheduling explicit, users provide lightweight annotations on the local computation kernel using AutoOverlap's API. These annotations do not change the numerical semantics of the kernel; instead, they expose its tiling structure and iteration order to the compiler. Although expressed as Python comments, they follow a structured directive format analogous to OpenMP [23] pragmas, allowing the compiler to reliably parse and verify them. Concretely, we require three pieces of information:

- **Tile size:** the logical shape of each tile along the relevant dimensions (e.g., GEMM blocks), which allows us to map tiles to chunks.
- **Tile index identifier:** a program variable (or tuple of variables) that uniquely identifies the tile being processed in a given iteration.
- **Tile scheduler:** the loop or control structure that advances the tile index and determines the order in which tiles are visited.

These annotations can often be derived from existing indexing expressions and loop bounds with minor code changes, as illustrated in the overview section.

Dependency Parsing. With both the communication sched-

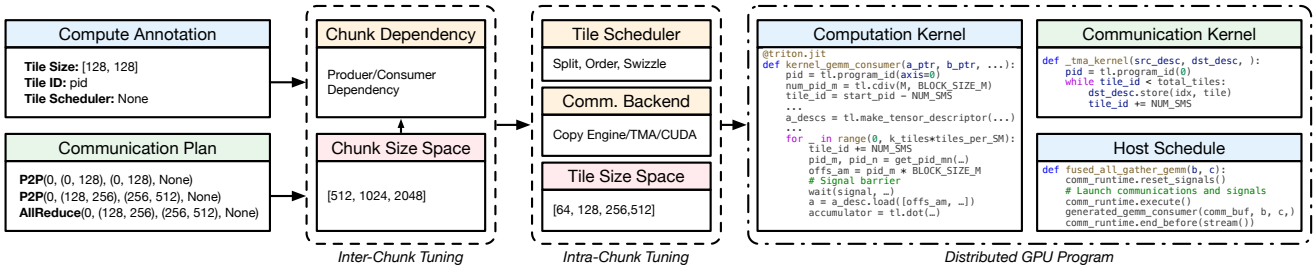


Figure 5: Compilation pipeline. In this example, we show communication using specialized SM as an independent kernel synchronized with signals. It can also be a fused kernel depending on the communication backend.

ule and the annotated compute kernel in hand, AutoOverlap constructs a dependence graph over chunks and tiles. For each chunk, we track its producer(s) and consumer(s), as well as any explicit ordering constraints encoded in the communication schedule (e.g., pipeline stages). For each tile, we determine which chunks it reads and writes based on its tile index and the tensor layout.

From this graph, the compiler identifies the minimal set of synchronization points needed to respect all data dependencies. Concretely, we insert wait operations in the kernel so that a tile that consumes a given chunk cannot start until the corresponding communication operator has completed. This synchronization can be implemented using different mechanisms depending on the chosen backend, but is always derived from the same chunk-level dependency structure.

Communication Code Generation. Once the communication schedule and tile scheduler have been aligned, AutoOverlap lowers the abstract chunk-level plan into concrete communication code. As illustrated in Fig. 7, the same logical schedule can be realized by several backends that differ in how they move chunks and how they allocate SM resources. Concretely, we support five realizations: (1) using the dedicated copy engine, (2) using TMA on a specialized SM, (3) using TMA on a co-located SM, (4) using operator-instruction load-/store on a specialized SM, and (5) using operator-instruction load/store on a co-located SM. In all cases, tiles are produced on one side and consumed by operations on the other side, but the mechanism for signaling readiness and the division of compute versus communication work across SMs differ.

For each operator, AutoOverlap first builds a dependency graph over tiles and communication steps from the chunk schedule, then lowers this graph into backend-specific code that enforces all dependencies by construction. When targeting the copy engine or a specialized SM, the compiler emits global-memory signals and kernel launches so that communication progresses asynchronously relative to the main compute tiles. When targeting co-located SM backends, it instead generates shared-memory barriers and index bookkeeping to coordinate communication and computation within the same SM. Because all five realizations share the same logical schedule but expose different latency/bandwidth and resource tradeoffs, they form a search space for the autotuner: AutoOverlap automatically generates all valid implementations, measures their end-to-end performance, and selects the best-performing backend for each operator and hardware configuration.

Tile-Scheduler Swizzling. As visualized in Fig. 6, the communication plan and the original computation kernel typically induce different layouts over the global tensor: communication groups tiles into chunks based on where data needs to move, while the kernel groups tiles into waves based on its own traversal order. Prior work reconciles this mismatch by explicitly reordering data between communication and computation, paying extra global-memory traffic and synchronization. In contrast, AutoOverlap keeps the communicated chunks in-place and instead *swizzles* the tile scheduler at the intra-kernel level. We reorder the sequence of waves so that each chunk is consumed as soon as it arrives, and apply an intra-chunk swizzle that visits tiles in an order that preserves locality within the chunk. This chunk-based tile schedule aligns compute with communication progress without additional reordering kernels, enabling fine-grained overlap purely through scheduling.

5.3 Communication-Centric Auto-Tuning

The chunk abstraction also provides a natural space for communication-centric auto-tuning. Because chunks sit exactly at the boundary between the global communication schedule and the local tile scheduler, changing chunk-level parameters simultaneously reshapes how data moves across ranks and how computation is ordered within each kernel. Rather than only tuning conventional kernel parameters (e.g., block sizes), AutoOverlap exposes a higher-level search space whose knobs directly control overlap and resource sharing.

At the *inter-chunk* level, we tune the chunk size, shape, and split factor for each logical transfer. Larger chunks tend to achieve higher effective bandwidth on copy engines and TMA but reduce the granularity of overlap, while smaller chunks enable more fine-grained pipelining at the cost of higher per-chunk overhead and more synchronization. Different operators and model sizes favor different trade-off points: communication-heavy A2A-GEMM and GEMM-AR, for example, benefit from intermediate split factors that balance bandwidth and overlap, as confirmed by our sensitivity study

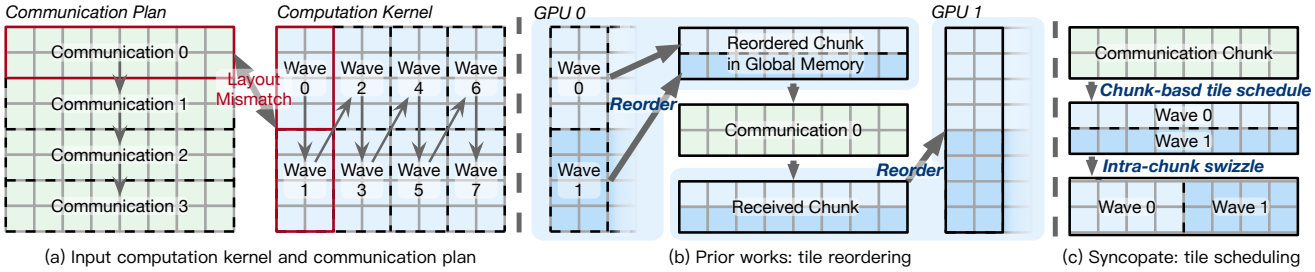


Figure 6: Tile scheduler transformation. (a) Computation and communication naturally follow different tile/chunk layouts, creating misalignment. (b) Prior approaches resolve this by inserting explicit data reordering between the two paths. (c) Syncopate instead rewrites the tile schedule to follow chunk order and applies intra-chunk swizzles for locality, aligning compute with communication progress without extra data movement.

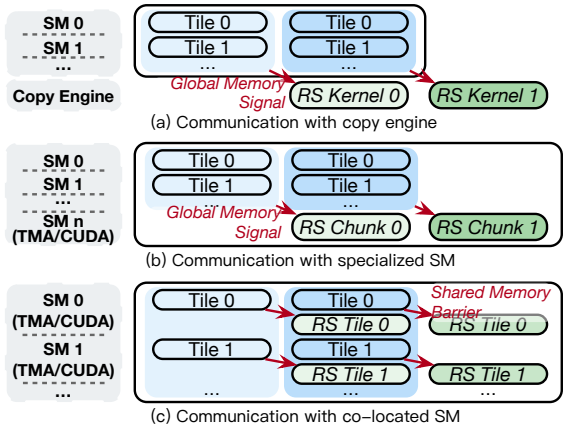


Figure 7: Communication backend selection. (a) Communication issued by the copy engine with global-memory signaling. (b) Dedicated SMs drive transfer. (c) Communication is co-located with compute on the same SM.

in §6. The tuner searches this space under hardware-specific constraints (e.g., minimum efficient transfer size for copy engines and TMA alignment rules) and prunes configurations that would violate these hardware limits.

At the *intra-chunk* level, we tune both the computation tile configuration and how each chunk is realized by a communication backend. Given a fixed logical schedule, the compiler can instantiate each transfer using any of the backends in Fig. 7 (copy engine, intra- or inter-SM TMA, or CUDA load-/store on specialized or co-located SMs) and can vary the number of SMs assigned to communication when applicable. Some schedules benefit from using TMA for intra-node tensor movement and load/store-based communication for small, reduction-heavy shards, while others favor copy engines for large bulk transfers with minimal SM involvement. In parallel, the autotuner explores different tile sizes and intra-tile orders that better align compute waves with the chosen chunk layout, improving locality and avoiding long communication tails.

Crucially, all of these decisions operate on top of the same chunk-level dependence graph. Changing the backend, SM allocation, or tile order never requires re-deriving the global communication plan; instead, AutoOverlap reuses the existing schedule and regenerates backend-specific code that enforces the same dependencies. This separation of logical schedule from physical realization is what makes the search space both rich and manageable: as our ablation results show, reasonable but suboptimal settings can easily leave more than a factor of two in performance, while the tuned configuration found by our communication-centric autotuner consistently coincides with the most balanced point between computation, communication, and hardware utilization.

6 Evaluation

6.1 Experimental Setup

Testbed. We evaluate AutoOverlap on a server with 8 NVIDIA H100 GPUs connected via NVLink with an aggregate bandwidth of 900 GB/s, which is a quite common setting used in previous works [3, 13, 31, 46]. Unless otherwise stated, all measurements are taken on a single node using all 8 GPUs; in later experiments, we vary the number of active devices to study scalability and portability. AutoOverlap is implemented with CUDA v12.9, NVSHMEM v3.3.9, and PyTorch v2.7, and we run all baselines on the same software stack to ensure a fair comparison.

Workloads. We use AutoOverlap to optimize representative multi-GPU operators that dominate the cost of modern LLM workloads: general matrix multiplication (GEMM) and attention [7, 8, 26, 33]. For GEMM, we benchmark three distributed variants: AllGather–GEMM (AG-GEMM) and GEMM–ReduceScatter (GEMM-RS), and GEMM–AllReduce (GEMM-AR), which appear in tensor-parallel [29] or sequence parallel [17] feed-forward network (FFN) layers. For attention, we evaluate both head-parallel (HP) [15] and sequence-parallel (SP) schedules, including the

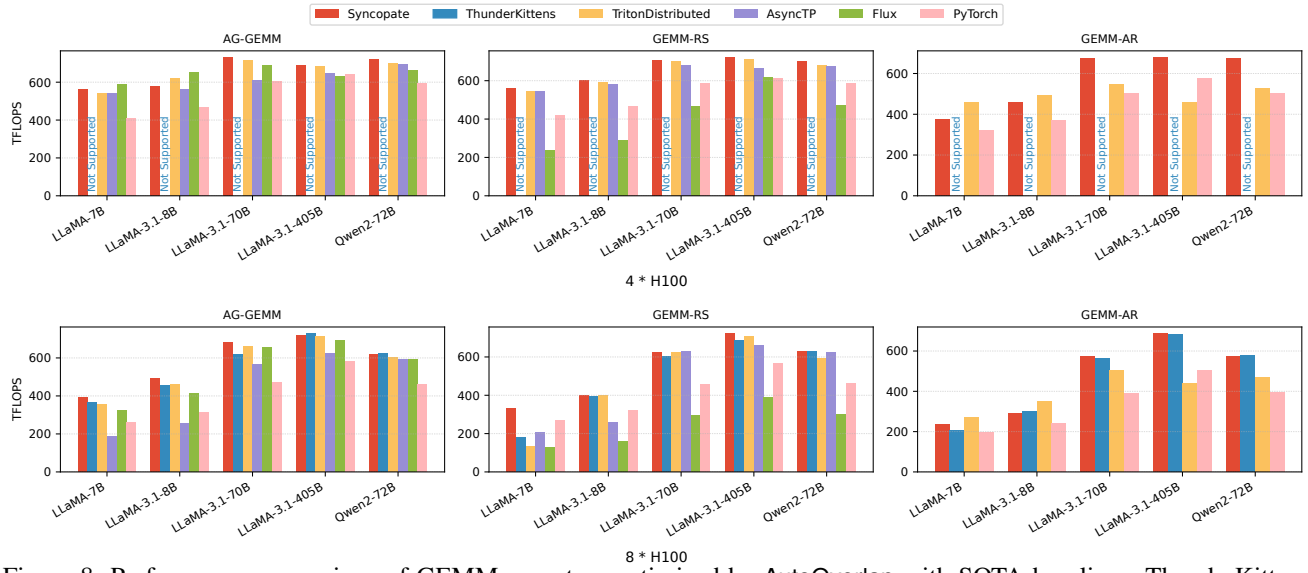


Figure 8: Performance comparison of GEMM operators optimized by AutoOverlap with SOTA baselines. ThunderKittens supports only 8 GPUs; 4-GPU is unsupported. When both settings are unsupported, the bar is omitted.

overlapped RingAttention (Ring-Attn) [18] variant.

Operator shapes are derived from the FFN layers and attention layers of open-source Llama-3 [11] and Qwen [24] models, covering a range of hidden dimensions, head counts, and parallelism configurations that are typical of large-scale LLM deployments. For attention, we sweep over multiple sequence lengths to reflect common short- and long-context use cases under different distribution strategies. Overall, this workload suite exercises AutoOverlap across both regular GEMM-heavy and more irregular attention patterns.

Baselines. To assess the effectiveness of operators generated and tuned by AutoOverlap, we compare against both state-of-the-art manually engineered kernels and fully automatic compiler-based approaches. As manual baselines, we include built-in operators from fine-grained overlap DSLs such as ThunderKittens [30, 31] and Triton-Distributed [45, 46], as well as highly optimized implementations including AsyncTP [36], Flux [3], and Triton kernels paired with NCCL [22] collectives.

To isolate the benefit of AutoOverlap's automatic fine-grained overlap, we further compare against automatic operators produced by existing distributed compiler frameworks, including Domino [34], Alpa [44], and Mercury [13]. For these comparisons, we transform the communication schedules found by each compiler into our chunk-level representation and reuse the same high-level plans in AutoOverlap, so that any performance difference reflects our intra-kernel overlap and backend-selection mechanisms rather than differences in global parallelization strategy.

6.2 Performance Benchmark

Operator Results. Across all evaluated settings, AutoOverlap generates automatically optimized operators whose performance exceeds, carefully hand-engineered baselines, while showing even larger gains over fully automatic distributed compilers. As summarized in Fig. 8 and Fig. 9, AutoOverlap sustains high TFLOPS on both GEMM and attention operators under multiple communication patterns (AG-GEMM, GEMM-RS, GEMM-AR, HP/SP attention, and Ring-Attn) and model configurations derived from Llama-3 and Qwen.

In Fig. 8, AutoOverlap is at or near the best-performing curve in almost every GEMM configuration. On common, heavily optimized AG-GEMM and GEMM-RS cases where manual kernels such as ThunderKittens, TritonDistributed, AsyncTP, and Flux already sit close to the hardware limits, AutoOverlap essentially matches their peak throughput on both 4- and 8-GPU settings, achieving on average 99.8% of the best baseline on 4 GPUs and 104% on 8 GPUs. For GEMM-AR, AutoOverlap is marginally below TritonDistributed on 7B/8B shapes, yet it scales more effectively and becomes the top-performing kernel on larger model configurations. These results indicate that our generic compilation pipeline can recover the same highly tuned overlap patterns that experts design by hand.

In Fig. 9, we observe a similar trend for attention operators. Under standard HP attention with moderate sequence lengths, AutoOverlap closely tracks the best manual implementations, confirming that the chunk-based abstraction does not sacrifice performance even for workloads that already benefit from hand-optimized kernels. As the problem becomes harder, moving to Ring-Attn, longer sequences, and 8-GPU runs, the

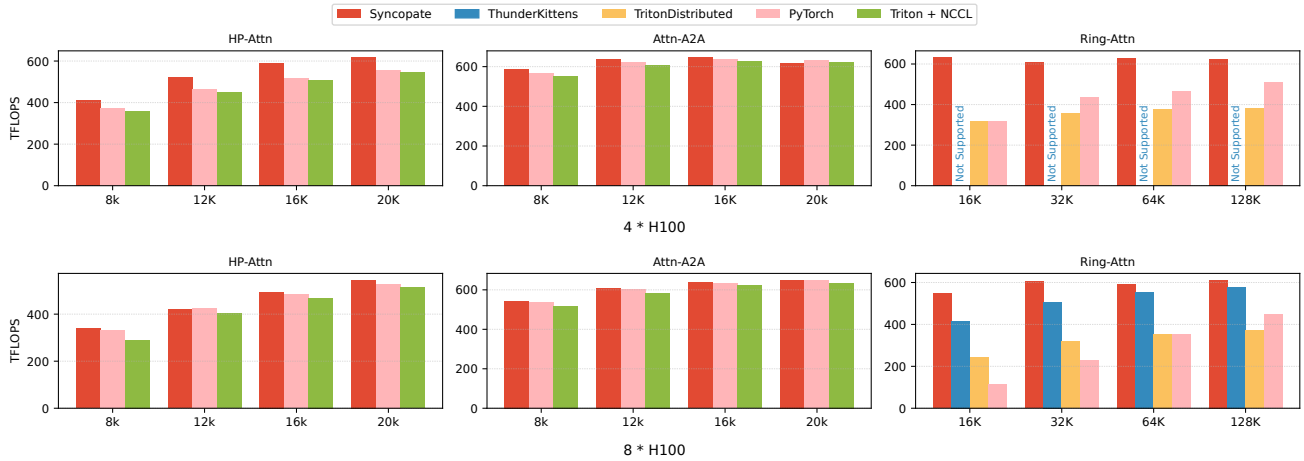


Figure 9: Performance comparison of operators optimized by AutoOverlap with SOTA baselines.

gap widens in favor of AutoOverlap. Our compiler maintains high TFLOPS while baseline kernels degrade more rapidly, since it can reshape chunks, rebalance compute and communication, and choose different backends as the sequence length and parallelism strategy change. Notably, on the most communication-intensive Ring-Attn settings, AutoOverlap delivers the best performance despite not being tailored specifically for this operator.

Integration Results. We further evaluate how AutoOverlap composes with existing automatic distributed compilers that search for the communication schedule among devices, using their communication plans as our inputs. As summarized in Fig. 10, for each of Domino, Alpa, and Mercury, we keep the original parallelization strategy and the searched communication schedule fixed, convert that schedule into our chunk-level representation, and let AutoOverlap generate the fine-grained overlapped kernels. Across both GEMM and attention workloads on 4- and 8-H100 configurations, integrating AutoOverlap consistently reduces end-to-end operator latency compared to the native implementations shipped with these systems, showing that chunk-based intra-kernel overlap exposes an additional optimization dimension on top of their global parallelization decisions.

This experiment also illustrates that AutoOverlap can be cleanly integrated with both partition-based and loop-based compiler stacks. Domino and Alpa operate on partitioned IRs and expose their communication plans as sequences of collectives between logical tensor partitions, while Mercury is built around a loop-centric IR for ring [18] and double-ring attention [12]. Because our interface only requires a well-defined, implementation-agnostic communication schedule and lightweight annotations on the local kernels, these systems can plug into AutoOverlap with minimal source modifications. This compatibility allows practitioners to reuse mature distributed compilers for global partitioning, while relying on AutoOverlap to automatically realize high-performance, fine-grained overlap within each generated operator.

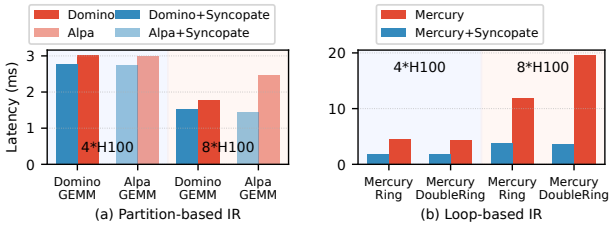


Figure 10: Evaluation lowering partitioned-based IR and loop-based IR searched by higher-level distributed compilers integrated with AutoOverlap.

6.3 Ablation and Sensitivity Studies

We next evaluate the effect of AutoOverlap's chunk-based code generation and communication-centric auto-tuning components through a series of ablation and sensitivity analyses in Fig. 11. Each subplot corresponds to one of the key design choices introduced in §5: communication backend selection and SM allocation, chunk size (split factor), and intra-tile scheduling.

Communication Backend and SM Tuning. Fig. 11(a) and (c) study how different realizations of the same logical communication schedule perform under the backends described in § 4.2, and how tuning the number of active SMs further refines this choice. In (a), for GEMM-RS and AG-GEMM, copy-engine and intra-SM TMA backends achieve the highest TFLOPS, while purely CUDA load/store realizations saturate at much lower throughput. The gap between the best and worst backend for the same logical schedule is comparable to the gap between our final implementation and several baselines in Fig. 8, indicating that backend selection alone can determine whether overlap is effective. This confirms that the ability to instantiate the same chunk schedule with different backends is crucial: no single mechanism dominates across operators, and

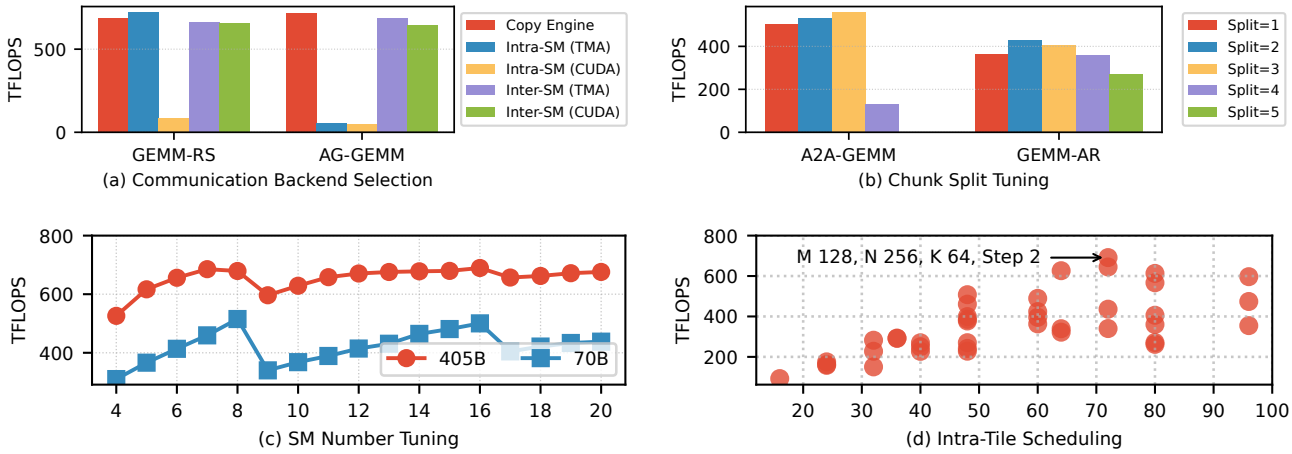


Figure 11: Ablation and sensitivity studies of AutoOverlap's auto-tuning design space.

picking a suboptimal backend can leave more than half of the available performance on the table. In (c), for a fixed backend, varying the number of SMs devoted to communication reveals a clear sweet spot where computation and communication are balanced. For both 405B and 70B GEMMs, allocating too few SMs underutilizes the link bandwidth, whereas allocating too many starves the main kernel. The optimal SM count also shifts with model size, which matches the design of our backend code generation: the autotuner treats SM allocation as a first-class knob and automatically selects a near-optimal point for each operator/hardware pair instead of relying on a single hard-coded ratio.

Chunk Size Tuning. Fig. 11(b) varies the number of chunks (split factor) used to realize the same high-level schedule for A2A-GEMM and GEMM-AR. Smaller split factors correspond to larger chunks with higher single-transfer efficiency but fewer overlap opportunities, while larger split factors increase overlap at the cost of per-chunk overhead. The curves exhibit a clear non-monotonic trend, with performance peaking at an intermediate split (e.g., 2–3 splits, about 128MB for GEMM-AR) and degrading when chunks become either too coarse or too fine. Notably, naive choices that are convenient to implement by hand (such as a single large chunk or splitting once per rank) sit far from the optimum. This behavior directly reflects the trade-off discussed in our chunk-based code generation: practitioners cannot reliably pick a single "good" chunk size by hand, whereas AutoOverlap searches this space automatically and selects the configuration that best matches the operator's compute/communication balance.

Intra-Tile Scheduling. Finally, Fig. 11(d) explores different intra-tile scheduling strategies for a representative GEMM configuration, varying the order in which tiles within each chunk are visited. Each point corresponds to a valid schedule with tile size on M, N, K dimensions and the pipeline stages that preserve program semantics but change locality and load balance. To represent different search candidates, we calculate the consumed shared memory size and plot the performance of these valid schedules. The wide spread in TFLOPS shows that tile order alone can introduce more than a 2× performance difference, reinforcing the importance of the tile-scheduler transformation in § 4.2. High-performing schedules cluster around orders that align tile waves with the communication chunk order introduced in our chunk-based code generation, whereas poorly performing ones repeatedly revisit tiles in a way that destroys locality or creates long tails of unfinished work. By generating and evaluating these schedules automatically, AutoOverlap converges on tile orders that co-optimize cache reuse and SM utilization, without requiring users to reason manually about low-level tiling and swizzling policies.

Taken together, these studies demonstrate that AutoOverlap's chunk abstraction is not only expressive but also forms a practical search space: the same logical schedule can be realized via multiple backends, chunk sizes, SM allocations, and tile orders, and the differences between reasonable but non-optimal choices and the tuned configuration are often comparable to, or larger than, the gaps between systems in our main benchmarks. The auto-tuning framework in § 4.3 systematically explores this space using the code generation mechanisms of § 4.2, turning what would otherwise be a brittle, hand-tuned process into a robust, compiler-driven optimization.

7 Conclusion

Syncopate bridges the abstraction gap between communication planning and fine-grained overlapping by introducing a chunk-based interface that unifies communication plans with tiled GPU computation. By automatically aligning tile schedules with communication progress and selecting among heterogeneous backends, Syncopate turns intra-kernel overlap into a general compiler capability rather than a hand-engineered optimization. The framework integrates cleanly with existing distributed compilers, complementing their

global parallelization strategies with backend-aware, fine-grained scheduling. This decoupled design provides a foundation for systems that co-schedule computation and communication and accommodate diverse communication backends.

References

[1] Sami Alabed, Daniel Belov, Bart Chrzaszcz, Juliana Franco, Dominik Grewe, Dougal Maclaurin, James Molloy, Tom Natan, Tamara Norman, Xiaoyue Pan, et al. Partir: Composing spmd partitioning strategies for machine learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 794–810, 2025.

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 929–947, New York, NY, USA, 2024. Association for Computing Machinery.

[3] Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, Xuanrun Zhang, Zuquan Song, Chengji Yao, Ziheng Jiang, et al. Flux: fast software-based communication overlap on gpus through kernel fusion. *arXiv preprint arXiv:2406.06858*, 2024.

[4] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 178–191, 2024.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[6] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 3393–3404, Red Hook, NY, USA, 2018. Curran Associates Inc.

[7] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[8] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[9] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 804–817, 2023.

[10] Raja Gond, Nipun Kwatra, and Ramachandran Ramjee. Tokenweave: Efficient compute-communication overlap for distributed llm inference, 2025.

[11] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[12] Diandian Gu, Peng Sun, Qinghao Hu, Ting Huang, Xun Chen, Yingtong Xiong, Guoteng Wang, Qiaoling Chen, Shangchun Zhao, Jiarui Fang, et al. Loongtrain: Efficient training of long-sequence llms with head-context parallelism. *arXiv preprint arXiv:2406.18485*, 2024.

[13] Yue Guan, Xinwei Qiang, Zaifeng Pan, Daniels Johnson, Yuanwei Fang, Keren Zhou, Yuke Wang, Wanlu Li, Yufei Ding, and Adnan Aziz. Mercury: Unlocking multi-gpu operator optimization for llms via remote memory scheduling. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 1046–1061, New York, NY, USA, 2025. Association for Computing Machinery.

[14] Ke Hong, Xiuhong Li, Minxu Liu, Qiuli Mao, Tianqi Wu, Zixiao Huang, Lufang Chen, Zhong Wang, Yichong Zhang, Zhenhua Zhu, et al. Flashoverlap: A lightweight design for efficiently overlapping communication and computation. *arXiv preprint arXiv:2504.19519*, 2025.

[15] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. *arXiv preprint arXiv:2309.14509*, 2023.

[16] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 402–416, 2022.

[17] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5:341–353, 2023.

[18] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ringattention with blockwise transformers for near-infinite context. In *The Twelfth International Conference on Learning Representations*.

[19] NVIDIA. NVIDIA H100 Tensor Core GPU Architecture. Technical report, NVIDIA, mar 2022. White paper.

[20] NVIDIA. NVIDIA Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) Rev 3.0.0, 2024. Accessed: 2025-02-10.

[21] NVIDIA Corporation. Nvidia nvlink high-speed interconnect: Application performance. Technical report, NVIDIA Corporation, 2015. Accessed: 2025-04-16.

[22] NVIDIA Corporation. *NVIDIA Collective Communications Library (NCCL)*, 2025. Version 2.26.2.

[23] OpenMP Architecture Review Board. Openmp application programming interface. https://www.openmp.org/specifications/, 2023.

[24] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025.

[25] Keshav Santhanam, Siddharth Krishna, Ryota Tomioka, Tim Harris, and Matei Zaharia. Distir: An intermediate representation and simulator for efficient neural network distribution. *arXiv preprint arXiv:2111.05426*, 2021.

[26] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems*, 37:68658–68685, 2024.

[27] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. Tensor program optimization with probabilistic programs. *Advances in Neural Information Processing Systems*, 35:35783–35796, 2022.

[28] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-tensorflow: Deep learning for supercomputers, 2018.

[29] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[30] Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré. Thunderkittens: Simple, fast, and adorable ai kernels, 2024.

[31] Stuart H. Sul, Simran Arora, Benjamin F. Spector, and Christopher Ré. Parallelkittens: Systematic and practical simplification of multi-gpu ai kernels, 2025.

[32] Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 10–19, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[34] Guanhua Wang, Chengming Zhang, Zheyu Shen, Ang Li, and Olatunji Ruwase. Domino: Eliminating communication in llm training via generic tensor slicing and overlapping, 2024.

[35] Haoran Wang, Lei Wang, Haobo Xu, Ying Wang, Yuming Li, and Yinhe Han. Primepar: Efficient spatial-temporal tensor partitioning for large transformer model training. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 801–817, New York, NY, USA, 2024. Association for Computing Machinery.

[36] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, et al. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 93–106, 2022.

[37] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, page 93–106, New York, NY, USA, 2022. Association for Computing Machinery.

[38] William Won, Midhilesh Elavazhagan, Sudarshan Srinivasan, Swati Gupta, and Tushar Krishna. Tacos: Topology-aware collective algorithm synthesizer for distributed machine learning. In *Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture*, MICRO '24, page 856–870. IEEE Press, 2024.

[39] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Man Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. Mirage: A {Multi-Level} superoptimizer for tensor programs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 21–38, 2025.

[40] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. Gspmd: General and scalable parallelization for ml computation graphs, 2021.

[41] Shulai Zhang, Ningxin Zheng, Haibin Lin, Ziheng Jiang, Wenlei Bao, Chengquan Jiang, Qi Hou, Weihao Cui, Size Zheng, Li-Wen Chang, et al. Comet: Fine-grained computation-communication overlapping for mixture-of-experts. *arXiv preprint arXiv:2502.19811*, 2025.

[42] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[43] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[44] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[45] Size Zheng, Wenlei Bao, Qi Hou, Xuegui Zheng, Jin Fang, Chenhui Huang, Tianqi Li, Haojie Duanmu, Renze Chen, Ruifan Xu, et al. Triton-distributed: Programming overlapping kernels on distributed ai systems with the triton compiler. *arXiv preprint arXiv:2504.19442*, 2025.

[46] Size Zheng, Jin Fang, Xuegui Zheng, Qi Hou, Wenlei Bao, Ningxin Zheng, Ziheng Jiang, Dongyang Wang, Jianxi Ye, Haibin Lin, et al. Tilelink: Generating efficient compute-communication overlapping kernels using tile-centric primitives. *arXiv preprint arXiv:2503.20313*, 2025.

[47] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Tian Tang, Qinyu Xu, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Nanoflow: Towards optimal large language model serving throughput, 2025.