Unit-II

CUDA in Practice

- CPUs and GPUs, both execute programs but they are different in their design goals.
- CPUs use an **MIMD (multiple instruction, multiple data)** approach.
- The CPU approach to parallelism is to execute multiple independent instruction streams. Within those instruction streams it seeks to extract **instruction level parallelism**
- it fills a very long pipeline of instructions and looks for instructions that can be sent to independent execution units.
- Execution units usually consist of :
  - **one or more floating-point units,**
  - **one or more integer units,**
  - **a branch prediction unit,**
  - **& one or more load/store units**

CUDA in Practice

- Branch prediction is an important concept.
- The problem with branching is that the single instruction stream turns into two streams:
  - the **branch taken** path
  - the **branch not taken** path.
- Programming constructs such as ***for***, ***while*** loops typically branch backwards to the start of the loop until the loop completes.
- In many cases, the branch can be predicted statically.
- Some compilers help with this in setting a bit within the branch instruction to say if the branch is likely to be met or not.
- This has the added advantage that the next instructions have typically already been prefetched into the cache

CUDA in Practice

- Along with branch prediction, a technique called ***speculative execution*** is used.
- The CPU will have predicted a branch correctly, it makes sense to start executing the instruction stream at that branch address.
- However, this adds to the cost of branch misprediction, as now the instruction stream that has been executed has to be discarded.
- The optimal model for both ***branch prediction*** and ***speculative execution*** is simply to execute both paths of the branch and then commit the results when the actual branch is known.
- As branches are often nested, in practice such an approach requires multiple levels of hardware and is therefore rarely used

CUDA in Practice

- **GPU**s use an **SIMT** (single instruction, multiple thread) model
- One of the aspects of GPU design that differs significantly from CPU design is the SIMT model of execution.
- In the MIMD model, there is separate hardware for each thread, allowing entirely separate instruction.
- If the threads are processing the same instruction flow, but with

different data, the GPU provides a single set of hardware to run N threads, where N is currently 32, the warp size

CUDA in Practice

- **SIMT** implementation in the GPU is similar to the old vector architecture **SIMD** model.
- In SIMT programmers are no longer forced to write code in which every thread follows the same execution path.
- Threads can diverge and then converge at some later point.
- Each path must be executed in turn, or serialized, until the control flow

converges once more.

- As a programmer you must be aware of this and think about it in the design of your kernels

CUDA in Practice

- On the CPU model, there is serial control flow. Executing an instruction that requires a number of cycles to complete will stall the current thread. This is one of the reasons why Intel uses **hyper threading.**
- The hardware internally switches to another thread when the current one stalls.
- GPU has a benefit in that it uses lazy evaluation.
- That is, it will not stall the current thread until there is an access to the dependent register.
- Thus, you may read a value into a register early in the kernel, and the thread will not stall until the register is actually used.
- The CPU model stalls at a memory load or long latency instruction

CUDA in Practice

- Segment 1:

```
int sum=0;
for (int i=0; i< 128; i++)
{
sum += src_array[i];
}
```

- If the above segment, the program must calculate the address of src_array[i], then load the data, and finally add it to the existing value of sum.
- Each operation **is dependent** on the **previous operation**.

CUDA in Practice

- Segment 2:

```
int sum=0;
int sum1=0, sum2=0, sum3=0, sum4=0;
for (int i=0; i< 128; i+=4)
{
		sum1 += src_array[i];
		sum2 += src_array[i+1];
		sum3 += src_array[i+2];
		sum4 += src_array[i+3];
}
sum =sum1 + sum2 + sum3 + sum4;
```

CUDA in Practice

- Second segment, we iterate in steps of four.
- Four independent sum values are used, allowing four independent summations to be computed in the hardware.
- How many operations are actually run in parallel depends on the number of execution units available on the processor.

CUDA in Practice

Segment 3:

```
int sum=0;
int sum1=0, sum2=0, sum3=0, sum4=0;
for (int i=0; i< 128; i+=4)
{
        const int a1 = src_array[i];
        const int a2 = src_array[i+1];
        const int a3= src_array[i+2];
        const int a4 = src_array[i+3];
        sum1 += a1;
        sum2 += a2;
        sum3 += a3;
        sum4 += a4;
}
sum = sum1 + sum2 + sum3+ sum4;
```

CUDA in Practice

- In the third segment, we move the load from memory operations out of the computation steps.
- The **load** operation for **a1** has **three** further load operations after it, plus some array index calculations, prior to its usage in the sum1 calculation.
- In the **eager evaluation** model used by CPUs we stall at the first read into a1, and on each subsequent read.
- With the **lazy evaluation** model used by GPUs we stall only on consumption of the data, the additions in the third code segment, if that data is not currently available

Algorithms that work best on the CPU versus the GPU

- Many parallel programs and parallel languages assume the MIMD model.
- The threads are independent and do not need to execute in groups (warps) as on the GPU.
- **MPI** and **Open MP** don't really fit well to the **GPU** model.
- **OpenMP** is perhaps the closest, in that it requires a **shared view** of memory.
- In OpenMP the compiler takes care of spawning threads that share a common data area.
- The programmer specifies which loop can be parallelized through various compiler pragmas and the compiler takes care of all that nasty "parallel stuff."

Algorithms that work best on the CPU versus the GPU

- **MPI**, on the other hand, considers all processes to be identical and is more suited to clusters of nodes than single-node machines.
- Typically, MPI is implemented as shared CPU/GPU pairs with the CPU handling the network and disk input/output (I/O).
- Implementations using GPU Direct allow transfers to certain InfiniBand network cards via a common shared-memory host page.

Algorithms that work best on the CPU versus the GPU

- With the GPU you have to consider that there are a limited number of threads that can easily work together on any given problem.
- Typically, we’re looking at up to 1024 threads on Fermi and Kepler,

  less on older hardware.
- The other major consideration for GPU algorithms is the memory available on the device.
- The largest single GPU memory space available is 6 GB on the Tesla M2090 cards.

Algorithms that work best on the CPU versus the GPU

- Many CPU algorithms make use of recursion.
- It’s convenient to break down a problem into a smaller problem that is then broken down further and so on until it becomes a trivial problem.
- Binary search is a classic example of this.
- Binary search splits a sorted list of numbers in half and simply asks the question of whether the data we’re looking for exists in the left or right set.
- It then repeats the split until either the item is found or the problem becomes just two items and is thus trivial to solve.

Algorithms that work best on the CPU versus the GPU

- Recursion is also problematic on GPUs, as it's only supported on compute 2.x GPUs, and then only for __device__ functions and not __global__ functions.
- The upcoming dynamic parallelism feature found in the Kepler K20 design will help in many respects with recursive algorithms.

Algorithms that work best on the CPU versus the GPU

- Quick sort is also a common example of an algorithm that is typically implemented recursively.
- The algorithm picks a pivot point and then sorts all items less than the pivot point to the left and less than or equal to the pivot point to the right.
- You now have 2 independent datasets that can be sorted by two independent threads.
- This then becomes 4 threads on the next iteration, then 8, then 16, and so on.

Algorithms that work best on the CPU versus the GPU

- How do you replicate such algorithms on a GPU?
- The easiest is when the parallelism scales in some known manner, as with quick sort.
- You can then simply invoke **one kernel per level** or **one kernel per N** levels of the algorithm back-to-back in a single stream.
- As one level finishes, it writes its state to global memory and the next kernel execution picks up on the next level.
- As the kernels are already pushed into a stream ready to execute, there is no CPU intervention needed to launch the next stream.

Algorithms that work best on the CPU versus the GPU

- Where the parallelism grows by some indeterminate amount per iteration, you can also store the state in global memory.
- You have to then communicate back to the host the number of the amount of parallelism that the next iteration will explore.

PROCESSING DATASETS: ballot

_ballot():

```
unsigned int __ballot(int predicate);
```

- This function evaluates the predicate value passed to it by a given thread.
- A predicate, is simply a true or false value.
- If the predicate value is nonzero, it returns a value with the Nth bit set, where N is the value of the thread (threadIdx.x).

PROCESSING DATASETS : ballot

- This atomic operation can be implemented as C source code as follows:

```
__device__ unsigned int __ballot_non_atom(int predicate)
{
        if (predicate != 0)
                return (1 << (threadIdx.x % 32));
        else
                return 0;
}
```

PROCESSING DATASETS:ballot

- The usefulness of ballot may not be immediately obvious, unless you combine it with another atomic operation, **atomicOr.**
- The prototype for this is

  int atomicOr(int * address, int val);
- It reads the value pointed to by address, performs a bitwise OR operation with the contents of val, and writes the value back to the address.
- It also returns the old value

PROCESSING DATASETS: ballot

- It can be used in conjunction with the __ballot function as follows:

```
volatile __shared__ u32 warp_shared_ballot[MAX_WARPS_PER_BLOCK];
// Current warp number - divide by 32
const u32 warp_num = threadIdx.x >> 5;
atomicOr( &warp_shared_ballot[warp_num], __ballot(data[tid] >
threshold) );
```

PROCESSING DATASETS:ballot

- we use an array that can be either in shared memory or global memory, but obviously
- shared memory is preferable due to it's speed.
- We write to an array index based on the warp number, which we implicitly assume here is 32.
- Thus, each thread of every warp contributes 1 bit to the result for that warp.
- For the predicate condition, if the value in **data[tid]**, our source data, is greater than a given threshold.
- Each thread reads one element from this dataset.
- The results of each thread are combined to form a bitwise OR of the result where thread 0 sets (or not) bit 0, thread 1 sets (or not) bit 1, etc.

PROCESSING DATASETS: _popc function

- **__popc function:** This returns the number
- of bits set within a 32-bit parameter. It can be used to accumulate a block-based sum for all warps in the block, as follows:

```
        atomicAdd(&block_shared_accumulate,
        __popc(warp_shared_ballot[warp_num]));
```

- we can accumulate for a given CUDA block the number of threads in every warp that had the condition we used for the predicate set

Locality

- The principle of locality is important in GPUs and CPUs.
- Memory closer to the device (shared memory on the GPU, cache on the CPU) is quicker to access.
- Communication within a socket (i.e., between cores) is much quicker than communication to another core in a different socket.
- Communication to a core on another node is at least an order of magnitude slower than within the node.
- Clearly, having software that is aware of this can make a huge difference to the overall performance of any system.
- Such socket-aware software can split data along the lines of the hardware layout.

MULTI-CPU SYSTEMS

- The most common multi-CPU system is the single-socket, multicore desktop.
- Almost any PC you buy today will have a multicore CPU.
- Even in laptops and media PCs, you will find multicore CPUs.
- If we look at Steam’s regular hardware (consumer/gaming) survey in mid 2012 , it reveals that :
  - 50% of users had dual-core systems
  - an additional 40% had quad-core or higher systems.

MULTI-CPU SYSTEMS

- The second type of multi-CPU systems you encounter is in workstations and low-end servers.
- These are often dual-socket machines, typically powered by multicore Xeon or Opteron CPUs.

MULTI-CPU SYSTEMS

- The final type of multi-CPU systems you come across are data center–based servers where you can have typically 4, 8, or 16 sockets, each with a multicore CPU.
- Such hardware is used to create a virtualized set of machines, allowing companies to centrally support large numbers of virtual PCs

  from one large server.

MULTI-CPU SYSTEMS

- One of the major problems you have with any multiprocessor system is memory coherency.
- Both CPUs and GPUs allocate memory to individual devices.
- In the case of GPUs, this is the global memory on each GPU card.
- In the CPU case, this is the system memory on the motherboard.

MULTI-CPU SYSTEMS

- To speed up access to memory locations, CPUs make extensive use of caches.
- Suppose two cores need to update x, because one core is assigned a debit processing task and the other a credit processing task.
- Both cores must have a consistent view of the memory location holding the parameter x.
- This is the issue of cache coherency.
- It limits the maximum number of cores that can practically cooperate on a single node.

MULTI-CPU SYSTEMS

- **Write invalidate** protocol is one solution for cache coherence.
- In this case the cores that need to access variable x need to get it from slow operating main memory.
- This will slow down the process

MULTI-CPU SYSTEMS

- In more complex coherency models, instead of invalidating x the invalidation request is replaced with an **write update** request.
- **Every write** has to be **distributed to N caches**.
- As the number of N grows the time to synchronize the caches becomes impractical.
- This often limits the practical number of nodes you can place into a symmetrical multiprocessor (SMP) system

MULTI-GPU SYSTEMS

- Similar to CPU systems, a lot of systems have multiple GPUs
- 9800GX2, GTX295, GTX590and GTX690 have dual cards
- you should always exploit to produce the best experience possible but is rather a pain if you wish to write a task that needs to have the GPUs cooperate in some way.

ALGORITHMS ON MULTIPLE GPUS

- **BOINC** is an application which allows users to donate spare computing power to solving the world's problems.
- On a multi-GPU system it spawns N tasks, where N is equal to the number of GPUs in the system.
- Each task gets a separate data packet or job from a central server.
- As GPUs finish tasks, it simply requests additional tasks from the central server (task dispatcher)

ALGORITHMS ON MULTIPLE GPUS

- Example 2:
- Encoding video is typically done by applying a JPEG-type algorithm to each individual frame and then looking for the motion vectors between frames.
- Thus, we have an operation within a frame that can be distributed to N GPUs, but then an operation that requires the GPUs to share data

  & has a dependency on the first task (JPEG compression) completing.
- The easiest is to use two passes:
  - one kernel that simply does the JPEG compression on N independent frames
  - second kernel that does the motion vector analysis–based compression

ALGORITHMS ON MULTIPLE GPUS

- We can do this because motion vector–based compression uses a finite window of frames, so frame 1 does not affect frame 1000. Thus, we can split the work into N independent jobs.

Which GPU

- How does the programmer select a GPU device?
- you need to set a device via a call to:

  cudaError_t cudaSetDevice(int device_num);

- or the simplified version often used in this text,

  CUDA_CALL(cudaSetDevice(0));

- The parameter device_num is a number from zero (the default device) to the number of devices in the system.

Which GPU

- To query the number of devices, simply use the following call:

cudaError_t cudaGetDeviceCount(int * device_count);

CUDA_CALL(cudaGetDeviceCount(&num_devices));

- CUDA_CALL macro takes the return value, checks it for an error, prints a suitable error message, and exits if there is a failure.

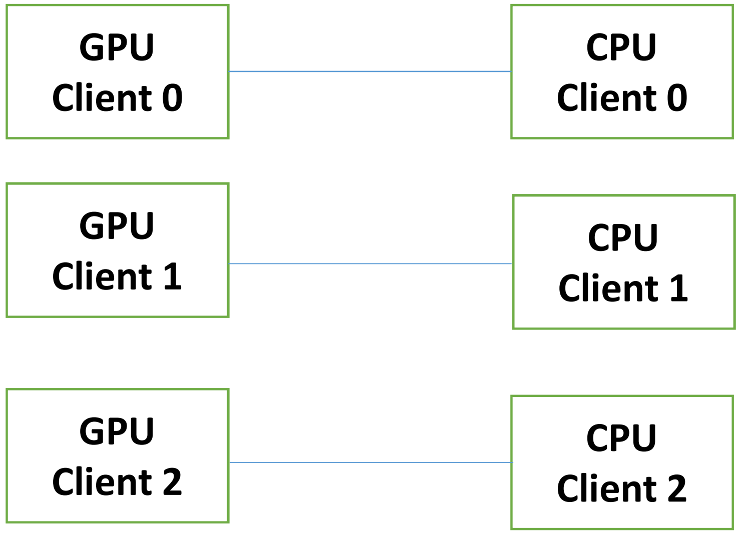
Which GPU

- How to select GPU?
- For this we need to know the details of a particular device. We can query this with the following call:

```
cudaError_t cudaGetDeviceProperties(struct cudaDeviceProp * properties, int device);
struct cudaDeviceProp device_0_prop;
CUDA_CALL(cudaGetDeviceProperties(&device_0_prop, 0));
```

Single Node System

- In versions of CUDA prior to the 4.0 SDK single-node systems were the only multi-GPU model available as shown in FIG



- In a single-node system, a CPU-based task is associated with a single-GPU context.
- The CUDA runtime binds the CPU process/thread ID to the GPU context, ensuring subsequent CUDA calls allocate memory on the bound device.
- Processes on the CPU can be likened to blocks on the GPU, where each is scheduled to run on a respective core or SM (streaming multiprocessor).
- CPU threads are similar to GPU threads but do not execute in groups (warps) like GPU threads.
- Threads in a CPU communicate and cooperate through shared memory within the same process, while processes communicate using interprocess communication.
- Communication within a CPU core, socket, node, or system, as well as between computer systems, is possible.
- Multiple CPUs using shared host memory can communicate via interprocess communication libraries.
- Multiple GPUs can communicate through host memory or directly via PCI-E bus peer-to-peer communication.

Single Node System

- A single CPU-based task would be associated with a single-GPU context.
- A task in this context would be either a process or a thread.
- Behind the scenes the CUDA runtime would bind the CPU process/thread ID to the GPU context.
- Thus, all subsequent CUDA calls (e.g., cudaMalloc) would allocate memory on the device that was bound to this context.
- From a programming perspective, the process/thread model on the host side is fragmented by the OS type.
- A **process** is a program that runs as an independent schedulable unit on a CPU and has its own data space.

Single Node System

- A thread is a much more lightweight element of the CPU scheduling.
- It shares both the code and data space used by its parent process.
- However, as with a process, each thread requires the OS to maintain a state (instruction pointer, stack pointer, registers, etc.).

Single Node System

- Threads may communicate and cooperate with other threads within the same process.
- Processes may communicate and cooperate with other processes through inter process communication.
- Such communication between processes may be within a CPU core, within a CPU socket, within a CPU node, within a rack, within a computer system, or even between computer systems.

Single Node System

- CPU threads are similar to the GPU threads.
- They don’t execute in groups or warps as the GPU.
- GPU threads communicate via shared memory and explicitly synchronize to ensure every thread has read/written to that memory.
- The shared memory is local to an SM, which means threads can only communicate with other threads within the same SM.
- A block is the scheduling unit to an SM, thread Communication is actually limited to a per-block basis.

Single Node System

- Processes on the CPU can be thought of in the same way as blocks on theGPU.
- A process is scheduled to run on one of N CPU cores.
- A block is scheduled to run one of N SMs on the GPU.
- In this sense the SMs act like CPU cores
- CPU processes can communicate to one another via host memory on the same socket.
- Since processes using a separate memory space, this can only happen with the assistance of a third-party interprocess communications library.
- It is not true for GPU blocks, as they access a common address space on the GPU global memory

Single Node System

- Systems with multiple CPUs using shared host memory can communicate with one another via this shared host memory, with the help of a interprocess communication library.

- Multiple GPUs can communicate to one another on the same host, using host memory, or, as of CUDA SDK 4.0, directly via the PCI-E bus peer-to-peer communication model.

STREAMS

- A CUDA stream is a sequence of commands that execute on the GPU in the order they are issued.
- You can think of streams as independent execution queues on the GPU, allowing multiple tasks to be processed concurrently.
- Streams are especially useful for overlapping computation with data transfers, kernel execution, and other operations to fully utilize the GPU's processing power.

STREAMS

- You can create and manage streams in CUDA using the CUDA Runtime API or the CUDA Driver API

- cudaStream_t stream;
- cudaStreamCreate(&stream);
- cudaStreamDestroy(stream); // clean up when done

STREAMS

- To ensure that operations in a stream are complete before proceeding, you can use **cudaStreamSynchronize(stream)** to synchronize with a specific stream.
- This function will block the CPU until all operations in the given stream are finished.
- The primary advantage of using streams is to overlap operations for better GPU utilization.
- Ex: you can start a data transfer operation (e.g., cudaMemcpyAsync) and a kernel execution in separate streams to overlap data transfer and computation, reducing overall execution time.

STREAMS

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

// Copy data from host to device in stream1
cudaMemcpyAsync(device_data, host_data, size,
cudaMemcpyHostToDevice, stream1);

// Execute a kernel in stream2
myKernel<<<grid, block, 0, stream2>>>(device_data);

// Synchronize with both streams when necessary
cudaStreamSynchronize(stream1);
cudaStreamSynchronize(stream2);
```

STREAMS

- Advantages:
- **Improved GPU utilization**: Streams allow you to overlap data transfer, computation, and other tasks, leading to better resource utilization.
- **Reduced latency**: Overlapping operations can reduce the overall time it takes for a GPU-accelerated application to complete.
- **Enhanced concurrency**: Streams enable concurrent execution of tasks, making better use of the GPU's parallelism.

STREAMS

- Applications:
- Stream handling is crucial in :
  - real-time graphics
  - scientific simulations
  - machine learning
  - video processing

STREAMS

```
void fill_array(u32 * data, const u32 num_elements)
{
        for (u32 i=0; i< num_elements; i++)
        {
        data[i] = i;
        }
}
void check_array(char * device_prefix, u32 * data, const u32
num_elements)
{
        bool error_found = false;
```

STREAMS

```
for (u32 i=0; i< num_elements; i++)
{
        if (data[i] !=(i*2))
        {
        printf("%s Error: %u %u", device_prefix, i, data[i]);
        error_found = true;
        }
}
if (error_found ==false)
        printf("%s Array check passed", device_prefix);
}
```

STREAMS

- In the first function we simply fill the array with a value from 0 to num_elements.
- The second function simply checks that the GPU result is what we’d expect.

STREAMS

```
// Define maximum number of supported devices
        #define MAX_NUM_DEVICES (4)
// Define the number of elements to use in the array
        #define NUM_ELEM (1024*1024*8)
// Define one stream per GPU
        cudaStream_t stream[MAX_NUM_DEVICES];
// Define a string to prefix output messages with so
// we know which GPU generated it
        char device_prefix[MAX_NUM_DEVICES][300];
// Define one working array per device, on the device
        u32 * gpu_data[MAX_NUM_DEVICES];
// Define CPU source and destination arrays, one per GPU
        u32 * cpu_src_data[MAX_NUM_DEVICES];
        u32 * cpu_dest_data[MAX_NUM_DEVICES]
```

STREAMS

```
// Generate a prefix for all screen messages
        struct cudaDevicePropdevice_prop;
        CUDA_CALL(cudaGetDeviceProperties(&device_prop, device_num));
        sprintf(&device_prefix[device_num][0], "\nID:%d %s:", device_num,
                                        device_prop.name);

// Create a new stream on that device
        CUDA_CALL(cudaStreamCreate(&stream[device_num]));

// Allocate memory on the GPU
        CUDA_CALL(cudaMalloc((void**)&gpu_data[device_num],
                                single_gpu_chunk_size));
```

STREAMS

```
// Allocate page locked memory on the CPU
        CUDA_CALL(cudaMallocHost((void **) &cpu_src_data[device_num],
                                          single_gpu_chunk_size));
        CUDA_CALL(cudaMallocHost((void **)&cpu_dest_data[device_num],
                                  single_gpu_chunk_size));

// Fill it with a known pattern
fill_array(cpu_src_data[device_num], NUM_ELEM);
```

STREAMS

```
// Copy a chunk of data from the CPU to the GPU  asynchronous
        CUDA_CALL(cudaMemcpyAsync(gpu_data[device_num],
                cpu_src_data[device_num], single_gpu_chunk_size,
                cudaMemcpyHostToDevice, stream[device_num]));
// Invoke the GPU kernel using the newly created  stream - asynchronous invocation
                gpu_test_kernel<<<num_blocks,
                num_threads,
                shared_memory_usage,
                stream[device_num]>>>(gpu_data[device_num]);

        cuda_error_check(device_prefix[device_num],
                "Failed to invoke gpu_test_kernel");
```

STREAMS

```
// Now push memory copies to the host into the streams
// Copy a chunk of data from the GPU to the CPU asynchronous
        CUDA_CALL(cudaMemcpyAsync(cpu_dest_data[device_num],
                        gpu_data[device_num],
                        single_gpu_chunk_size,
                        cudaMemcpyDeviceToHost,
                        stream[device_num]));
}
```

STREAMS

- We create a stream, or work queue, for each GPU present in the system.
- Into this stream we place a copy from the host (CPU) memory to the GPU global memory followed by a kernel call and then a copy back to the CPU.

STREAMS

```
// Process the data as it comes back from the GPUs  Overlaps CPU execution
with GPU execution
        for (int device_num=0;device_num < num_devices;device_num++)
{
// Select the correct device
CUDA_CALL(cudaSetDevice(device_num));
//Wait for all commands in the stream to complete
CUDA_CALL(cudaStreamSynchronize(stream[device_num]));
```

STREAMS

```
// GPU data and stream are now used, so  clear them up
        CUDA_CALL(cudaStreamDestroy(stream[device_num]));
        CUDA_CALL(cudaFree(gpu_data[device_num]));

// Data has now arrived in  cpu_dest_data[device_num]
        check_array( device_prefix[device_num], cpu_dest_data[device_num], NUM_ELEM);

// Clean up CPU allocations
        CUDA_CALL(cudaFreeHost(cpu_src_data[device_num]));
        CUDA_CALL(cudaFreeHost(cpu_dest_data[device_num]));

// Release the device context
        CUDA_CALL(cudaDeviceReset());
}
}
```

STREAMS

- it checks the contents and then frees the GPU and CPU resources associated with each stream.
- we need to add some timing code to see how long each kernel takes in practice.
- Add events to the work queue.
- Now events are special in that we can query an event regardless of the currently selected GPU
- we need to declare a start and stop event:

```
// Define a start and stop event per stream
        cudaEvent_t kernel_start_event[MAX_NUM_DEVICES];
        cudaEvent_t memcpy_to_start_event[MAX_NUM_DEVICES];
        cudaEvent_t memcpy_from_start_event[MAX_NUM_DEVICES];
        cudaEvent_t memcpy_from_stop_event[MAX_NUM_DEVICES];
```

STREAMS

- Finally, we need to get the elapsed time and print it to the screen:

```
// Wait for all commands in the stream to complete
        CUDA_CALL(cudaStreamSynchronize(stream[device_num]));

// Get the elapsed time between the copy  and kernel start
        CUDA_CALL(cudaEventElapsedTime(&time_copy_to_ms,memcpy_to
        _start_event[device_num], kernel_start_event[device_num]));

// Get the elapsed time between the kernel start and copy back start
        CUDA_CALL(cudaEventElapsedTime(&time_kernel_ms,
        kernel_start_event[device_num],memcpy_from_start_event[device
                _num]));
```

STREAMS

```
// Get the elapsed time between the copy back start  and copy back start
        CUDA_CALL(cudaEventElapsedTime(&time_copy_from_ms,memcpy_from_
        start_event[device_num], memcpy_from_stop_event[device_num]));

// Get the elapsed time between the overall start and stop events
        CUDA_CALL(cudaEventElapsedTime(&time_exec_ms,
        memcpy_to_start_event[device_num],
        memcpy_from_stop_event[device_num]));

// Print the elapsed time
        const float gpu_time = (time_copy_to_ms þ time_kernel_ms +
        time_copy_from_ms);
        printf("%sCopy To : %.2f ms",
        device_prefix[device_num], time_copy_to_ms);
        printf("%sKernel : %.2f ms",
        device_prefix[device_num], time_kernel_ms);
```

STREAMS

```
printf("%sCopy Back : %.2f ms",  device_prefix[device_num],
                 time_copy_from_ms);
printf("%sComponent Time : %.2f ms", device_prefix[device_num],
                 gpu_time);
printf("%sExecution Time : %.2f ms", device_prefix[device_num],
time_exec_ms);
printf("\n");
```

STREAMS

- When we run the program we see the following result:
- ID:0 GeForce GTX 470:Copy To : 20.22 ms
- ID:0 GeForce GTX 470:Kernel : 4883.55 ms
- ID:0 GeForce GTX 470:Copy Back : 10.01 ms
- ID:0 GeForce GTX 470:Component Time : 4913.78 ms
- ID:0 GeForce GTX 470:Execution Time : 4913.78 ms
- ID:0 GeForce GTX 470:Array check passed

- ID:1 GeForce 9800 GT:Copy To : 20.77 ms
- ID:1 GeForce 9800 GT:Kernel : 25279.57 ms
- ID:1 GeForce 9800 GT:Copy Back : 10.02 ms
- ID:1 GeForce 9800 GT:Component Time : 25310.37 ms
- ID:1 GeForce 9800 GT:Execution Time : 25310.37 ms
- ID:1 GeForce 9800 GT:Array check passed

STREAMS

- ID:2 GeForce GTX 260:Copy To : 20.88 ms
- ID:2 GeForce GTX 260:Kernel : 14268.92 ms
- ID:2 GeForce GTX 260:Copy Back : 10.00 ms
- ID:2 GeForce GTX 260:Component Time : 14299.80 ms
- ID:2 GeForce GTX 260:Execution Time : 14299.80 ms
- ID:2 GeForce GTX 260:Array check passed

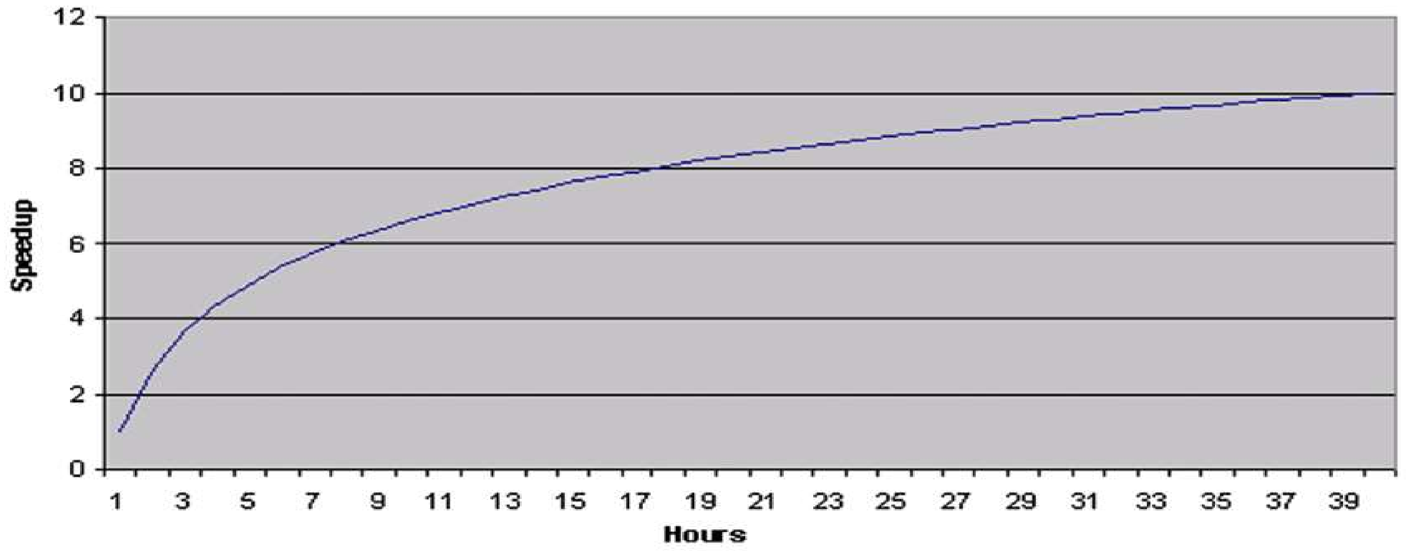
- ID:3 GeForce GTX 460:Copy To : 20.11 ms
- ID:3 GeForce GTX 460:Kernel : 6652.78 ms
- ID:3 GeForce GTX 460:Copy Back : 9.94 ms
- ID:3 GeForce GTX 460:Component Time : 6682.83 ms
- ID:3 GeForce GTX 460:Execution Time : 6682.83 ms
- ID:3 GeForce GTX 460:Array check passed

PARALLEL/SERIAL GPU/CPU PROBLEM BREAKDOWN

TIME

- what is an “acceptable” time period is for the execution time of the algorithm

as a software professional, your time costs money.



- The faster a program needs to execute, the more effort is involved in making this happen

TIME

- **Program Performance vs. Execution Speed**:
  - Program performance refers to how efficiently a piece of software accomplishes its tasks.
  - One crucial aspect of performance is the speed at which a program executes its operations.
  - Faster execution generally means that a program can complete its tasks more quickly and respond to user inputs or system events with less delay.
- **Effort in Software Development**:
  - Developing software involves various stages, including designing, coding, testing, debugging, and optimizing.
  - Each of these stages requires effort from developers.
  - Effort can be in the form of time, resources, and expertise.

TIME

- **Complexity of Optimization**:
  - The effort required to optimize a program for speed increases as the need for speed becomes more critical.
  - If a program has a loose performance requirement, developers may not invest as much effort in optimization, as long as it meets basic usability standards.
  - However, if a program is required to execute very quickly, developers must invest more time and resources in fine-tuning and improving its performance.

TIME

- **Trade-offs**:
- There are often trade-offs to consider when optimizing for speed.
- For example, making a program faster may lead to increased memory usage, making it less efficient in terms of resource consumption.
- Developers need to strike a balance between execution speed and other factors like memory footprint, code maintainability, and development time.

TIME

- In setting a suitable speedup goal, you have to be aware of what is reasonable, given a set of hardware.
- Internet search engines have to return a set of search results to the user with in seconds.
- At the same time, it used to be “acceptable” for their indexes to take several days to updated i.e the time taken for them to pick up new content.
- Thus what is acceptable today may not be acceptable tomorrow, next month, or next year.

TIME

- In considering what the acceptable time is, check how far you are away currently are from this.
- If it’s a factor of two or less, often it will be worth spending time optimizing the CPU implementation.
- Creating an entirely new, parallel approach to the problem may cause, problems like Multiple threads introduce:
    - Dependencies
    - Deadlock
    - Synchronization
    - Debugging, etc.
- If you can live with the serial CPU version, this may be a better solution in the short term.

TIME

- A high clock rate means high power consumption.
- The processor manufacturers have already abandoned that route in favour of multicore as the only long-term solution to providing more compute power.
- If you decide to go down the GPU route, which for many problems is a very good solution, then you should typically set your design goal to be around a 10 (ten times) improvement in execution time of the program.
- At least a 2 or 3 speedup is a relatively easy goal, even for those new to CUDA

Problem decomposition

- Can the problem be broken down into chunks that can run in parallel; is there an opportunity to exploit concurrency in the problem?
- If the answer is no, then the GPU is not the answer for you.
- One of the main limiting factors with CPU parallelization is that there is often just not enough parallel work to be done.
- GPUs run thousands of threads, so the problem needs to be decomposed into thousands of blocks, not just a handful of concurrent tasks as with the CPU

Problem decomposition

- problem decomposition should always start with the data first and the tasks to be performed second.
- You should try to represent the problem in terms of the output dataset.
- One of the issues with this type of approach is that you need to fully understand the problem for the best benefit.
- The real benefit of this approach comes from making the chain from the input data points to the output data points completely parallel.

Problem decomposition

- There are some problems where this single-output data point view is not practical video encoding
- In this particular problem, there are a number of stages defined, each

  of which defines a variable-length output data stream.
- Here the destination pixel is a function of N source pixels.
- This analogy works well in many scientific problems.
- Where the input set is very large, simply apply a threshold or cutoff point such that those input data points that contribute very little are excluded from the dataset.
- This will contribute a small amount of error, but in some problems allows a huge section of the dataset to be eliminated from the calculation.

Problem decomposition

- Optimize the operations or functions being performed on the data.
- However, as compute capacity has increased hugely in comparison to memory bandwidth, it’s now the data that is the primary consideration.
- GPUs have on the order of 5 to 10 times the memory bandwidth of CPUs, you have to decompose the problem such that this bandwidth can be used
- if you plan to use multiple GPUs or multiple GPU nodes communication between nodes will be very expensive in terms of computation cycles so it needs to be minimized and overlapped with computation.

Dependencies

- A dependency is where some calculation requires the result of a previous calculation.
- In either case, the dependency causes a problem in terms of parallel execution.
- Dependencies are seen in two main forms:
  - where one element is dependent on one or more elements around it
  - where there are multiple passes over a dataset and there exists a dependency from one pass to the next.

Dependencies

```
extern int a,c,d;
extern const int b;
extern const int e;
void some_func_with_dependencies(void)
{
a = b * 100;
c = b * 1000;
d = (a + c) * e;
}
```

Dependencies

- **a** and **c** have a dependency on **b**.
- **d** has **a** dependency on both **a** and **c**.
- The calculation of **a** and **c** can be done in parallel, but the calculation of **d** requires the calculation of both **a** and **c** to have completed.

Dependencies

- In a superscalar CPU, there are multiple independent pipelines.
- The independent calculations of a and c be dispatched to separate execution units that would perform the multiply.
- The results of those calculations would be needed prior to being able to compute the addition operation for a and c.
- The result of this addition operation would also need to be available before the final multiplication operation could be applied.

Dependencies

- This type of code arrangement allows for little parallelism and causes a number of stalls in the pipeline, as the results from one instruction must feed into the next.
- While stalled, the CPU and GPU would otherwise be idle.
- Clearly this is a waste
- On the CPU side, instruction streams from other virtual CPU cores fill in the gaps in the instruction pipeline (e.g., hyperthreading).
- However, this requires that the CPU know from which thread the instruction in the pipeline belongs, which complicates the hardware.

Dependencies

- On the GPU, multiple threads are also used, but in a time-switching manner, so the latency of the arithmetic operations is hidden with little or no cost.
- In fact, on the GPU you need around 20 clocks to cover such latency. However, this latency need not come from another thread.

Dependencies

```
extern int a,c,d,f,g,h,i,j;
extern const int b;
extern const int e;
void some_func_with_dependencies(void)
{
a = b * 100;
c = b * 1000;

f = b * 101;
g = b * 1001;

d = (a + c) * e;
h = (f + g) * e;

i =d * 10;
J =h * 10;
}
```

Dependencies

- Here the code has been rearranged and some new terms introduced.
- insert some independent instructions between the calculation of a and c and their use in d
- allow these calculations more time to complete before the result is obtained.
- The calculations of f, g, and h in the example are also overlapped with the d calculation.
- In effect, you are hiding the arithmetic execution latency through overlapping nondependent instructions.

Dependencies

- **loop fusion:**

```
void loop_fusion_example_unfused(void)
{
unsigned int i,j;
a = 0;
for (i=0; i<100; i++) /* 100 iterations */
{
a += b * c * i;
}
d = 0;
for (j=0; j<200; j++) /* 200 iterations */
{
D+= e * f
}
}
```

Dependencies

```
void loop_fusion_example_fused_01(void)
{
unsigned int i; /* Notice j is eliminated */
a = 0;
d = 0;
for (i=0; i<100; i++) /* 100 iterations */
{
a +=b * c * i;
d += e * f * i;
}
for (i+100; i<200; i++) /* 100 iterations */
{
d += e * f * i;
}
}
```

Dependencies

```
void loop_fusion_example_fused_02(void)
{
unsigned int i; /* Notice j is eliminated */
a = 0;
d =0;
for (i=0; i<100; i++) /* 100 iterations */
        {
        a += b * c * i;
        d += e * f * i;
        d += e * f * (i*2);
        }
}
```

Dependencies

- we have two independent calculations for results a and d.
- The number of iterations required in the second calculation is more than the first.
- However, the iteration space of the two calculations overlaps.
- You can, therefore, move part of the second calculation into the loop body of the first, as shown in function **loop_fusion_example_fused_01**.
- This has the effect of introducing additional, independent instructions, plus reducing the overall number of iterations, in this example, by one-third.

Dependencies

- In the loop_fusion_example_fused_02 we can further fuse the two loops by eliminating the second loop and fusing the operation into the first, adjusting the loop index accordingly.

Dependencies

- in the GPU it’s likely these loops would be unrolled into threads and a single kernel would calculate the value of a and d.
- There are a number of solutions, but the most likely is one block of
- 100 threads calculating a with an additional block of 200 threads calculating d.
- By combining the two calculations, you eliminate the need for an additional block to calculate d

Memory Considerations

Memory Bandwidth

- Bandwidth refers to the amount of data that can be moved to or from a given destination.
- In the GPU case we’re concerned primarily about the global memory bandwidth.
- Latency refers to the time the operation takes to complete.

Memory Bandwidth

- Memory latency is designed to be hidden on GPUs by running threads from other warps.
- When a warp accesses a memory location that is not available, the hardware issues a read or write request to the memory.
- This request will be automatically combined or coalesced with requests from other threads in the same warp, provided the threads access adjacent memory locations .
- f threads did not access consecutive memory addresses, it led to a rapid drop off in memory bandwidth

Memory Bandwidth

- Fermi, unlike compute 1.x devices, fetches memory in transactions of either 32 or 128 bytes.
- By default every memory transaction is a 128-byte cache line fetch.
- Access by a stride other than one, but within 128 bytes, now results in cached access instead of another memory fetch

Memory Bandwidth

- One of the key areas to consider is in the number of memory transactions.
- Each memory transaction feeds into a queue and is individually executed by the memory subsystem.
- There is a certain amount of overhead with this.
- It's less expensive for a thread to issue a read of four floats or four integers in one pass than to issue four individual reads

Source of limit

- Kernels are typically limited by two key factors, memory latency/bandwidth and instruction latency/bandwidth.
- which of these two key factors is limiting performance is critical to knowing where to direct your efforts.

Source of limit

- The simplest way is to see where the balance of the code lies is to simply comment out all the arithmetic instructions and replace them with a straight assignment to the result.
- Arithmetic instructions include any calculations, branches, loops, etc.
- If you have a one-to-one mapping of input values to calculated outputs, this is very simple and a one-to-one assignment works well.
- Where you have a reduction operation of one form or another, simply replace it with a sum operation

Source of limit

- Include all the parameters read from memory into the final output or the compiler will remove the apparently redundant memory reads/writes.
- Retime the execution of the kernel and you will see the approximate percentage of time that was spent on the arithmetic or algorithmic part.
- If this percentage is very high, you are arithmetically bound.
- Otherwise if very little changed on the overall timing, you are memory bound

Source of limit

- For Memory bound kernels rearrange the memory pattern so the GPU can coalesce the memory access pattern by thread.
- Thread 0 has to access address 0, thread 1 address 1, thread 2 address 2, and so on.
- Ideally, your data pattern should generate a column-based access pattern by thread, not a row-based access.
- If you can’t easily rearrange the data pattern, rearrange the thread pattern such that you can use them to load the data into shared memory before accessing the data

Source of limit

- With arithmetic-bound kernels, translate the the source code into assembly (PTX) code.
- Array indexes can often be replaced with pointer-based code, replacing slow multiplies with much faster additions.
- Divide or multiply instructions that use a power of 2 can be replaced with much faster right and left shift operations, respectively.
- Anything that is constant within a loop body, an invariant, should be moved outside the loop body.
- Adopt loopunrolling

Memory organization

- Getting the correct memory pattern for a GPU is often the key consideration in many applications.
- CPU programs typically arrange the data in rows within memory.
- Arrange the memory pattern such that access to it by consecutive threads will be in columns.
- This is true of both global memory and shared memory.
- This means for a given warp (32 threads)
  - thread 0 should access address offset 0,
  - thread 1 address offset 1,
  - thread 2 address offset 2, etc.

Memory organization

- Assuming you have an aligned access, 128 bytes of data will come in from global memory at a time.
- With a single float or integer per thread, all 32 threads in the warp will be given exactly one element of data each.
- ***cudaMalloc*** function will allocate memory in 128-byte aligned blocks.
- if using a structure that would straddle such a boundary
- Two approaches:
  - First, you can either add padding bytes/words explicitly to the structure.
  - Alternatively, you can use the cudaMallocPitch function

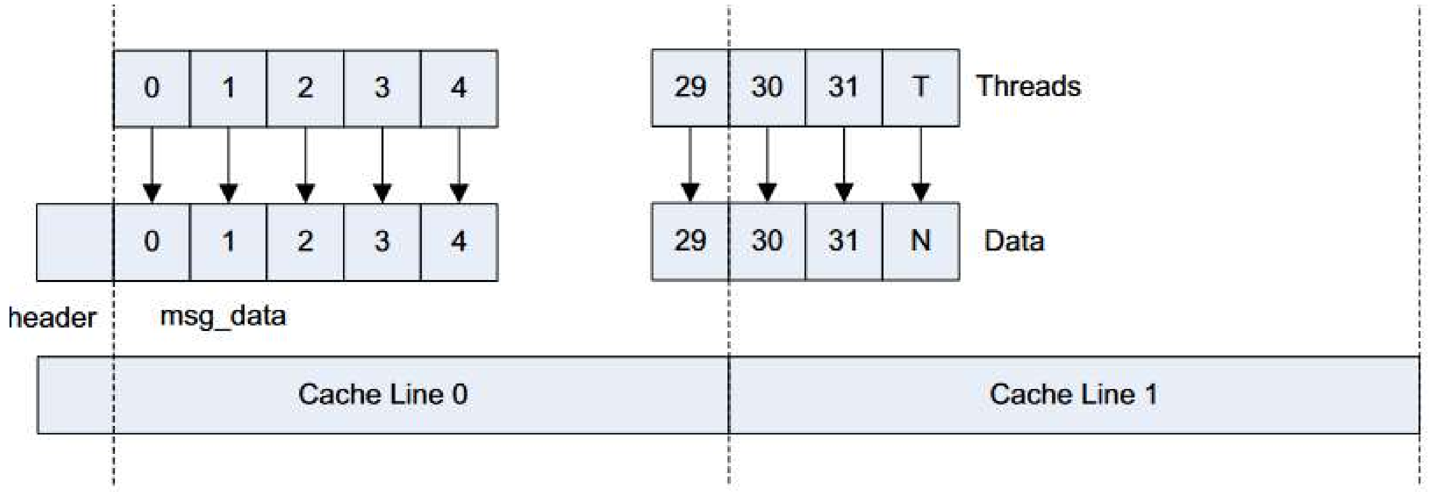
Memory organization

- alignment is a key criteria as to whether one or two memory transactions, or cache lines need to be fetched.
- suppose thread 0 accesses address offset 2 instead of 0.
- You are accessing some data structure that has a header at the start, such as

```
#define MSG_SIZE 4096
typedef struct
{
u16 header;
u32 msg_data[MSG_SIZE];
} MY_STRUCT_T;
```

Memory organization

- If the kernel processes msg_data, then threads 30 and 31 of the warp cannot be served by the single memory fetch.
- In fact, they generate an additional 128-byte memory transaction
- Any subsequent warps suffer from the same issue. You are halving your memory bandwidth, just by having a 2-byte header at the start of the data structure.



Memory organization

- Loading the header into a separate chunk of memory somewhere else allows for aligned access to the data block.
- (OR) Manually insert padding bytes into the structure definition to ensure that msg_data is aligned to a 128-byte boundary.
- Reordering the structure elements to move ‘header’ after msg_data will also work, providing the structure is not subsequently used to create an array of structures.

TRANSFERS

Pinned memory

- Pinned memory, also known as locked memory, is a concept used in both GPU (Graphics Processing Unit) and CPU (Central Processing Unit) systems to improve data transfer and reduce latency.
- It involves allocating memory in such a way that it remains fixed in physical RAM and cannot be swapped out to disk by the operating system's virtual memory management.
- The primary purpose of using pinned memory is to enable faster data transfers between the GPU and the CPU.

Pinned memory

- In GPU computing, pinned memory is particularly important when dealing with data transfers between the CPU and GPU.

- When data is transferred between the CPU and GPU, it usually involves several steps, such as copying data from the CPU's memory to the GPU's memory, processing the data on the GPU, and then transferring the results back to the CPU.

- During these data transfers, standard memory (also known as pageable memory) may be subject to swapping, which can introduce additional latency and decrease performance

Pinned memory

- Pinned memory eliminates this swapping overhead because it remains in the RAM and cannot be moved by the operating system.
- As a result, data transfers between the CPU and GPU can be more efficient and streamlined, especially for large datasets and frequent data exchanges.
- By using pinned memory in the GPU, data can be transferred directly from the CPU's pinned memory to the GPU without the need for an intermediate copy.
- This results in faster data transfer rates and reduces the CPU overhead.

Pinned memory

- We can allocate pinned memory using functions like cudaMallocHost or cudaHostAlloc to allocate pinned memory on the CPU side.
- Similarly, other frameworks and libraries may have their own methods for allocating pinned memory.

Pinned memory

- Applications:
  - machine learning,
  - scientific simulations,
  - any other tasks that involve significant data transfers between the CPU and GPU.

- However, it's important to note that pinned memory consumes a portion of the system's RAM.
- It should be used judiciously and only for data that truly requires it.

Zero-copy memory

- It is a technique used in GPU systems to improve data transfer efficiency between the CPU and the GPU.
- In traditional systems, when data is sent from the CPU to the GPU, it usually involves two separate memory copies:
- CPU to GPU memory copy: The data is first transferred from the CPU's main memory (RAM) to a separate memory location in the GPU's memory (VRAM).
- GPU to CPU memory copy: After processing the data on the GPU, the results are transferred back from the GPU's memory to the CPU's main memory.

Zero-copy memory

- These memory copies can lead to significant overhead and can slow down the overall performance, especially when dealing with large datasets.

- Zero-copy memory addresses this issue by allowing direct access to the same memory location from both the CPU and the GPU.

- This means that data can be accessed and manipulated by both the CPU and GPU without any additional memory copies.

Zero-copy memory

- Here's how zero-copy memory works:
- Unified Memory: Modern GPU systems employ unified memory architecture, where both the CPU and GPU can access the same virtual memory address space.
- This unified memory approach removes the need for explicit memory copies between CPU and GPU.

- Page Faulting: When the CPU accesses data that resides in the GPU memory, a page fault occurs.
- This causes the GPU driver to transfer the required data from the GPU memory to the CPU memory on-the-fly, enabling seamless data access by the CPU.
- Similarly, when the GPU needs data from the CPU memory, a page fault occurs, and the data is transferred to the GPU memory.

Zero-copy memory

- Zero-copy memory simplifies memory management for developers, as they don't need to explicitly manage data transfers between the CPU and GPU, making the code more readable and maintainable.
- The effectiveness of zero-copy memory depends on various factors, such as the GPU architecture, the type of data being processed, and the specific use case.
- While it can be beneficial in some scenarios, in others, explicit data transfers between CPU and GPU may still be more efficient. As GPU technologies evolve, the trade-offs and advantages of zero-copy memory may change, so it's essential to consider the latest hardware and software developments for optimal performance.

Zero-copy memory

- To allocate memory on the host such that it can be mapped into device memory.
- This is done with an additional flag cudaHostAllocMapped to the cudaHostAlloc function.

```
// Allocate zero copy pinned memory
CUDA_CALL(cudaHostAlloc((void **) &host_data_to_device, size_in_bytes,
                    cudaHostAllocWriteCombined | cudaHostAllocMapped));
```

Zero-copy memory

- we need to convert the host pointer to a device pointer, which is done with the cudaHostGetDevicePointer function as follows:

```
// Convert to a GPU host pointer
CUDA_CALL(cudaHostGetDevicePointer( &dev_host_data_to_device,
                    host_data_to_device, 0));\
```

- To free the memory later, an operation performed on the host, the existing call remains the same:

```
// Free pinned memory
CUDA_CALL(cudaFreeHost(host_data_to_device));
```

Bandwidth Limitations

- In GPU-CPU systems, bandwidth limitations refer to the constraints on data transfer between the CPU and the GPU.
- This may affect the overall performance of the system, particularly in tasks that heavily rely on data movement between the CPU and GPU, such as graphics rendering, deep learning, scientific simulations, and other parallel computing workloads.

Bandwidth Limitations

- factors that can contribute to bandwidth limitations in GPU-CPU systems:

- Memory bandwidth:
- Both the CPU and GPU have their dedicated memory (RAM), and data needs to be transferred between these memories when data is shared or when tasks are offloaded between the two processing units.
- The memory bandwidth represents the rate at which data can be read from or written to the memory.
- If the bandwidth is limited, it can cause data transfer bottlenecks and slow down the overall system performance.

Bandwidth Limitations

- PCI-E bandwidth:
- In systems where the GPU is connected to the CPU via the Peripheral Component Interconnect Express (PCI-E) interface, the bandwidth of this connection can also be a limiting factor.

- PCI-E is used to communicate between various components in the system, and if the available bandwidth is insufficient, it can hinder data transfers between the CPU and GPU.

Bandwidth Limitations

- Data transfer overhead:
- The process of data transfer between the CPU and GPU incurs some overhead due to various reasons:
  - protocol conversion
  - memory alignment
  - synchronization.

- These overheads can add up and further reduce the effective bandwidth between the two processing units.

Bandwidth Limitations

- Memory access patterns:
- The performance of data transfer can also depend on the access patterns used by the CPU and GPU.
- Ex: random memory accesses might be less efficient than sequential accesses due to the nature of memory hierarchies and cache systems.
- Memory contention:
- In some cases, both the CPU and GPU may compete for access to system memory simultaneously, leading to contention and reduced overall bandwidth.

Bandwidth Limitations

- To mitigate bandwidth limitations in GPU-CPU systems, several strategies can be employed:

- Data locality optimization:
- Designing algorithms and data structures that minimize data movement between CPU and GPU memory can help reduce the impact of bandwidth limitations.

- Data compression:
- Compressing data before transferring it between the CPU and GPU can reduce the amount of data to be transferred, potentially alleviating bandwidth constraints.

Bandwidth Limitations

- Asynchronous data transfers:
- In some cases, it's possible to overlap data transfers with other computations to better utilize available bandwidth.

- Improved memory hierarchies:
- Modern GPU and CPU architectures often feature advanced memory hierarchies and cache systems designed to optimize data movement and minimize the impact of bandwidth limitations.

- .

Bandwidth Limitations

- System-level optimization:
- Ensuring that the overall system configuration, including memory configurations and PCIe connections, is properly balanced to match the requirements of the workloads being executed can also help address bandwidth limitations

Single GPU & Multi GPU Timing

- Single GPU timing and multi-GPU timing refer to the performance metrics associated with running computational tasks on a single graphics processing unit (GPU) versus using multiple GPUs in parallel to perform the same task.
- These metrics are essential for assessing the efficiency and scalability of GPU-accelerated computations.

Single GPU & Multi GPU Timing

- Single GPU Timing:
- Single GPU timing measures the time it takes for a computational task to be executed on a single GPU.
- The time is typically measured in milliseconds (ms) or seconds (s).
- The lower the timing, the faster the GPU can process the task.
- Single GPU timing is crucial for evaluating the performance of a specific GPU model, its architecture, and its processing power.

Single GPU & Multi GPU Timing

- Multi-GPU Timing:
- Multi-GPU timing, refers to the time taken to complete a computation when multiple GPUs are working together in parallel.
- This approach is commonly referred to as GPU parallelism or GPU scaling.
- The time taken to execute the task with multiple GPUs is again measured in milliseconds or seconds.

Single GPU & Multi GPU Timing

- The goal of using multiple GPUs is to accelerate the computation significantly compared to using just one GPU or relying solely on the central processing unit (CPU).
- Multi-GPU timing is essential for understanding the efficiency of GPU parallelism and assessing the speedup achieved by employing multiple GPUs for a given task.

Single GPU & Multi GPU Timing

- If a computation can be perfectly parallelized and all GPUs work together without communication overhead, the multi-GPU timing would be nearly equal to the single GPU timing divided by the number of GPUs used. This concept is known as **linear scaling.**
- In practice, some computational tasks may not be fully parallelizable, and there may be communication overhead and synchronization challenges when using multiple GPUs.
- As a result, the actual multi-GPU timing might not perfectly follow linear scaling, but it should still be faster than the single GPU timing.

Overlapping GPU transfers

- Overlapping GPU transfers is a technique used to improve the overall performance and efficiency of GPU-based computations.
- They are connected to the CPU and host memory through a shared bus, which can introduce bottlenecks when transferring data between the GPU and the rest of the system.
- Modern GPUs and their associated software frameworks have implemented techniques to overlap data transfers with computation.
- One common approach is known as "asynchronous data transfers" or "pipelining," which allows the GPU to continue processing data while data transfers are taking place in the background.

Overlapping GPU transfers

- overlapping GPU transfers working:

- Initiate data transfer: The CPU or the host system initiates data transfer between the host memory and the GPU's memory. This can involve transferring input data to the GPU or retrieving results from previous computations.

Overlapping GPU transfers

- Asynchronous transfer:
- Instead of waiting for the data transfer to complete before starting computation, the GPU allows the CPU to continue processing other tasks while the data transfer is happening in the background.

- Computation begins:
- As soon as the GPU receives a portion of the data, it can start processing it in parallel with the ongoing data transfer.

Overlapping GPU transfers

- Pipelining:
- While the GPU is computing on the data, the CPU can already initiate the next data transfer or computation.
- This overlapping process helps maximize the utilization of both the GPU and the CPU, reducing idle time.

- Synchronization:
- At some point, the GPU may need to wait for all the required data to be available before it can complete the computation.
- In such cases, the GPU and CPU need to synchronize to ensure the correct results are obtained.

Overlapping GPU transfers

- The ability to overlap GPU transfers with computations can significantly enhance the overall throughput and performance of GPU-based applications.
- It requires careful consideration of the data transfer size, synchronization points, and the specific capabilities and limitations of the GPU and its software framework.

Thread usage

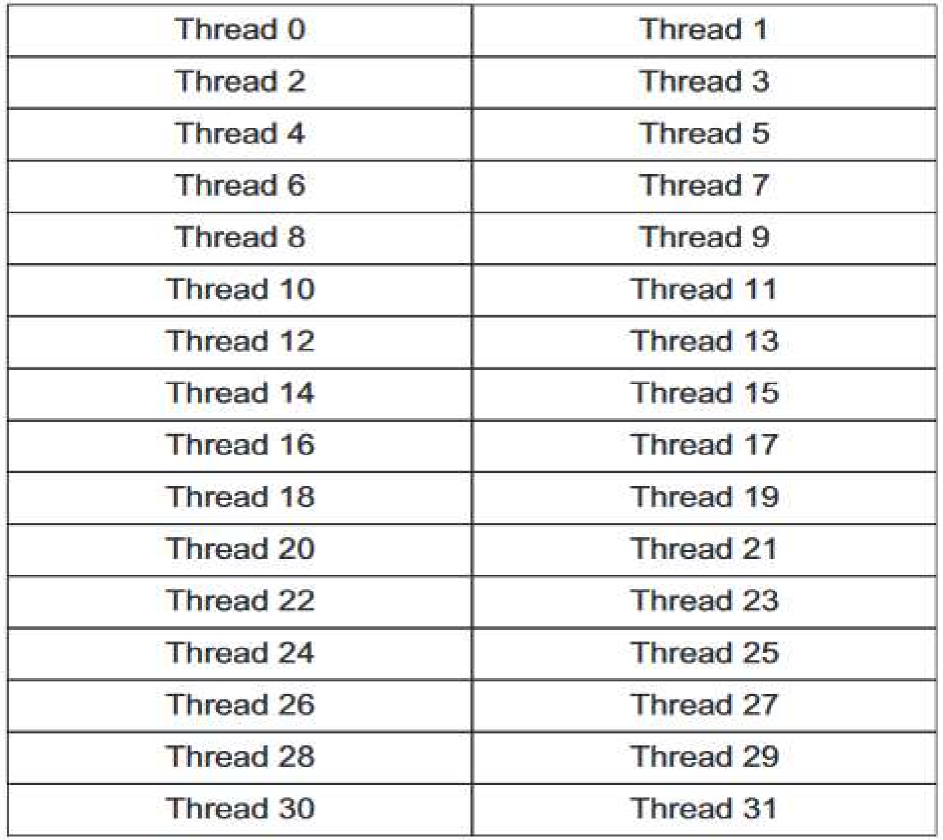
Thread usage

- The layout of threads in a GPU system can significantly impact performance, especially when accessing memory.
- In this case, we are comparing a 2 x 32 layout of threads versus a 32 x 2 layout of threads. Let's explore how each layout affects memory access and performance:

Thread usage

- In the 2 x32 example, thread 0 cannot be coalesced with any other thread than thread 1.
- In this case the hardware issues a total of 16 memory fetches.
- The warp cannot progress until at least the first half-warp has acquired all the data it needs.
- At least eight of these very long memory transactions need to complete prior to any compute activity on the SM

Thread usage



Thread 0	Thread 1
Thread 2	Thread 3
Thread 4	Thread 5
Thread 6	Thread 7
Thread 8	Thread 9
Thread 10	Thread 11
Thread 12	Thread 13
Thread 14	Thread 15
Thread 16	Thread 17
Thread 18	Thread 19
Thread 20	Thread 21
Thread 22	Thread 23
Thread 24	Thread 25
Thread 26	Thread 27
Thread 28	Thread 29
Thread 30	Thread 31

Thread usage

- On Fermi, the first of these would cause a read miss in the L1 cache. The L1 cache would request the minimum size of data possible, 128 bytes from the L2 cache.
- When data is moved from the L2 cache to the L1 cache, just 3.125% of the data moved is consumed by thread 0.
- On the first run through the code the L2 cache is unlikely to contain the data.
- It issues a 128-byte fetch also to slow global memory.
- This latency-expensive operation is finally performed and 128 bytes arrive at the L2 cache.

Thread usage

- We issue 1 coalesced read for 128 bytes instead of 16 separate reads.
- There’s a factor of 16:1 improvement in both the number of

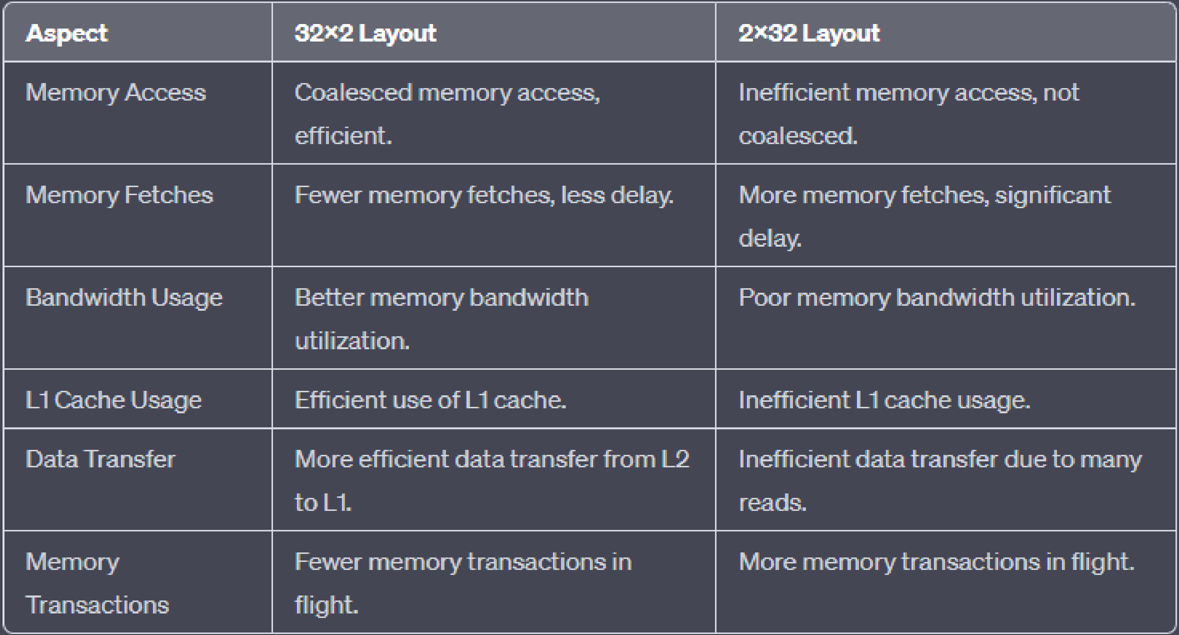
memory transactions in flight and also bandwidth usage.

- Data can be moved from the L2 to the L1 cache in just one transaction, not 16.

Inactive threads

Underutilization of GPU Cores:

- Each core is capable of executing multiple threads simultaneously.
- When some threads become inactive, meaning they have completed their tasks or waiting for dependencies, those cores remain idle or underutilized.
- This leads to wasted computational power and reduced overall performance
- GPUs are most efficient when they can keep all cores busy with active threads, allowing for maximum parallelism and throughput.
- Inactive threads can create bottlenecks, limiting the GPU's ability to fully exploit its processing capabilities.



Aspect	32×2 Layout	2×32 Layout
Memory Access	Coalesced memory access, efficient.	Inefficient memory access, not coalesced.
Memory Fetches	Fewer memory fetches, less delay.	More memory fetches, significant delay.
Bandwidth Usage	Better memory bandwidth utilization.	Poor memory bandwidth utilization.
L1 Cache Usage	Efficient use of L1 cache.	Inefficient L1 cache usage.
Data Transfer	More efficient data transfer from L2 to L1.	Inefficient data transfer due to many reads.
Memory Transactions	Fewer memory transactions in flight.	More memory transactions in flight.

Inactive threads

Occupied Memory Resources:

- Threads in a GPU system typically require memory resources to store their data and intermediate results.
- When threads become inactive, their allocated memory remains reserved and unavailable for other threads that might need it.

Thread Divergence Reduction:

- Thread divergence occurs when threads in the same GPU core execute different code paths based on conditional statements. This can lead to some threads becoming inactive while others continue processing.
- By optimizing the code to minimize thread divergence, developers can increase the chances of keeping all cores busy.

Inactive threads

- **Memory Management**:
- Efficient memory allocation and deallocation strategies can help prevent memory wastage caused by inactive threads. Releasing memory occupied by completed threads and managing memory fragmentation can improve overall GPU performance.

Arithmetic density

- Arithmetic density is a term that measures the relative number of calculations per memory fetch.
- Kernel that fetches two values from memory, multiplies them, and stores the result back to memory has very low arithmetic density.

```
C[z] = A[y] * B[x]
```

- The real work being done is the multiplication.
- With only one operation being performed per three memory transactions (two reads and one write), the kernel is very much memory bound.

Arithmetic density

- The total execution time is

  T = read time(A) + reads time(B)+ arithmetic time(M)+ store time(C)

  or

  T = A + B + M + C
- The individual read times are not easy to predict.
- Fetching of A may also bring into the cache B, so the access time for B is considerably less than A. Writing C may evict from the cache A or B.

Arithmetic density

- When looking at the arithmetic density, our goal is to increase the ratio of useful work done relative to memory fetches and other overhead operations.

Transcendental operations

- The GPU hardware is aimed at speeding up gaming environments.
- There are certain accelerators built into the GPU hardware. These are dedicated sections of hardware designed for a single purpose.
- GPUs have the following such accelerators:

  - Division
  - Square root
  - Reciprocal square root
  - Sine
  - Cosine
  - Log 2
  - Base 2 exponent Ex

Transcendental operations

- These various instructions perform operations to 24-bit accuracy, in line with the typical 24-bit RGB setup used in many game environments.
- If you'd like the faster but less precise operation, you have to enable them using either the compile switch (-use_fast_math) or explicitly using intrinsic operations.

Loop invariant analysis

- Loop invariant analysis looks for expressions that are constant within the loop body and moves them outside the loop body.

Example:

```
for (int j=0;j<100;j++)
{
for (int i=0; i<100; i++)
{
const int b = j * 200;
q[i]=b;
}
}
```

Loop invariant analysis

- he parameter j is constant within the loop body for parameter i.
- Thus, the compiler can easily detect this and will move the calculation of b outside the inner loop.

Loop invariant analysis

```
for (int j=0;j<100;j++)
{
const int b = j * 200;
for (int i=0; i<100; i++)
{
q[i]= b;
}
}
```

- This optimized code removes thousands of unnecessary calculations of b, where j, and thus b, are constant in the inner loop.

Loop unrolling

- Loop unrolling is a technique that seeks to ensure you do a reasonable number of data operations for the overhead of running through a loop. Take the following code:

```
{
for (i=0;i<100;i++)
q[i]=i;
}
```

Loop unrolling

- In terms of assembly code, this will generate:
- A load of a register with 0 for parameter i.
- A test of the register with 100.
- A branch to either exit or execute the loop.
- An increment of the register holding the loop counter.
- An address calculation of array q indexed by i.
- A store of i to the calculated address.
- Only the last of these instructions actually does some real work.
- The rest of the instructions are overhead

Loop unrolling

We can rewrite this C code as

```
{
for (i=0;i<25;i+=4)
q[i]=i;
q[i+1]=i+1;
q[i+2]=i+2;
q[i+3]=i+3;
}
```

Loop unrolling

- The NVCC compiler supports the **#pragma unroll** directive, which will automatically unroll fully such loops when the iteration count is constant or silently do nothing when it’s not
- You can also specify **#pragma unroll 4** where four is replaced by any number the programmer wishes

Loop peeling

- Loop peeling is an enhancement to the loop unrolling, when the number of iterations is not an exact multiple of the loop unrolling size.
- Here the last few iterations are peeled away and done separately,
- and then the main body of the loop is unrolled.

Loop peeling

- For example, if we have 101 loop iterations and plan to use four levels of loop unrolling, the first 100 iterations of the loop are unrolled
- The final iteration is peeled away to allow the bulk of the code to operate on the unrolled code.
- The final few iterations are then handled as either a loop or explicitly

Divergence

- GPUs execute code in blocks, or warps. A single instruction is decoded once and dispatched to a warp scheduler.
- It will be in a queue until the warp dispatcher dispatches it to a set of 32 execution units.
- It is similar to the old vector machines.
- The main difference is that CUDA does not require that every instruction execute in this way.
- If there is a branch in the code and only some instructions follow this branch, those instructions diverge while the others wait at the point of divergence.

Divergence

- The single fetch/decode logic then fetches the instruction stream for the divergent threads and the other threads simply ignore it.
- Each thread within the warp has a mask that enables its execution or not.
- Those threads not following the divergence have the mask cleared.
- Those following the branch have the bit set.
- This type of arrangement is called predication.
- A predicate is created, which results in a single bit being set for those threads within a warp that follow the branch.
- Most PTX op-codes support an optional predicate allowing selective threads to execute an instruction