CS 610: GPU Architecture and CUDA Programming

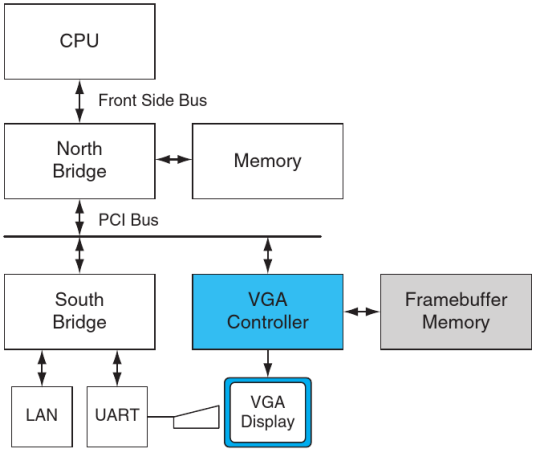
Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2024-25-I

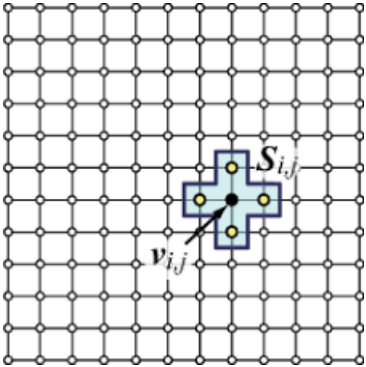
Rise of GPU Computing

- Rise of graphical OS in late 80s created a market for a new compute device
- Display accelerators (e.g., VGAs) offered hardware-assisted bitmap operations
- Silicon Graphics popularized use of 3D graphics
  - Released OpenGL as a programming interface to its hardware
- Popularity of first-person games in mid-90s accelerated the need for dedicated graphics accelerators that evolved from VGA controllers



Need for GPU Computing Support

- Many real-world applications are compute-intensive and data-parallel
- Need to process a lot of data, mostly floating-point operations
- Examples are
  - Real-time high-definition graphics applications such as your favorite video games
  - Iterative kernels which update elements according to some fixed pattern called a stencil



Rise of GPGPU Computing

- Researchers tricked GPUs to perform non-rendering computations, often referred to as general-purpose GPU (GPGPU) computations
- Programming initial GPU devices for other purposes was very convoluted
  - Programming model was very restrictive
  - Limited input colors and texture units, writes to arbitrary locations, floating-point computations
- This spurred the need for a generic highly-parallel computational device with high computational power and memory bandwidth
  - CPUs are more complex devices catering to a wider audience

Rise of GPGPU Computing

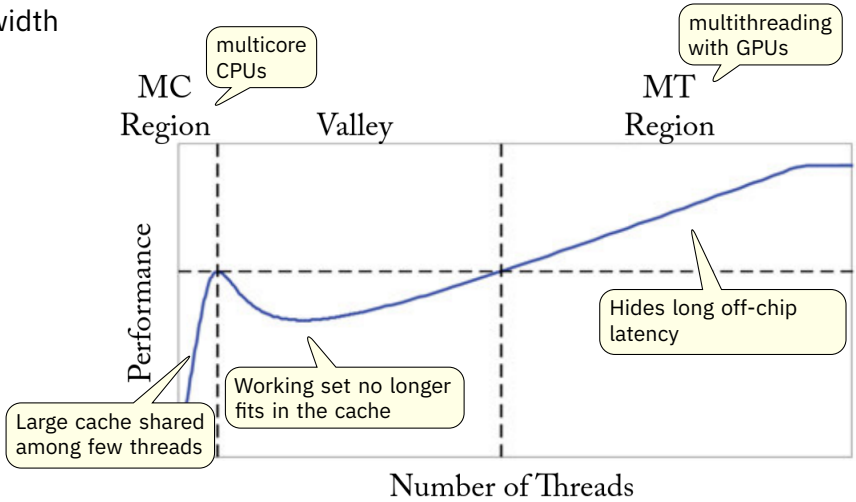
- NVIDIA released GeForce 8800 GTX in 2006 with CUDA architecture
  - + General-purpose ALU and instruction set for general-purpose computation
  - + Allowed arbitrary reads and writes to shared memory
  - + IEEE compliance for single-precision floating-point arithmetic
- Introduced CUDA C and the toolchain for ease of development with the CUDA architecture
- GPUs are now used in different applications
  - For example, game effects, computational science simulations, image processing, machine learning, and linear algebra
  - Modern GPUs serve both as a programmable graphics processor and a scalable parallel computing platform
- There are several GPU vendors like NVIDIA, AMD, Intel, Qualcomm, and ARM

GPU Architecture

Philosophy and Design Goals

Analytical Model to Compare CPU and GPU Performance

Simple cache model where threads do not share data and there is infinite off-chip memory bandwidth

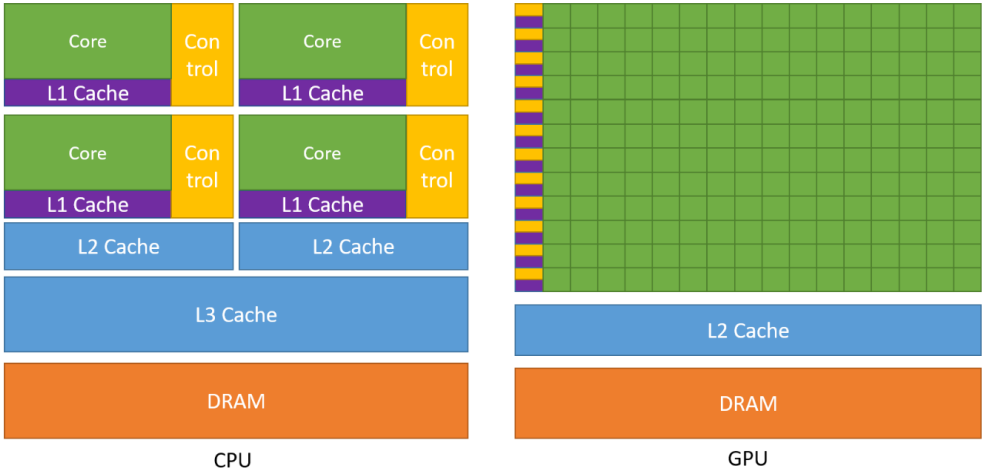


Key Insights in GPU Architecture

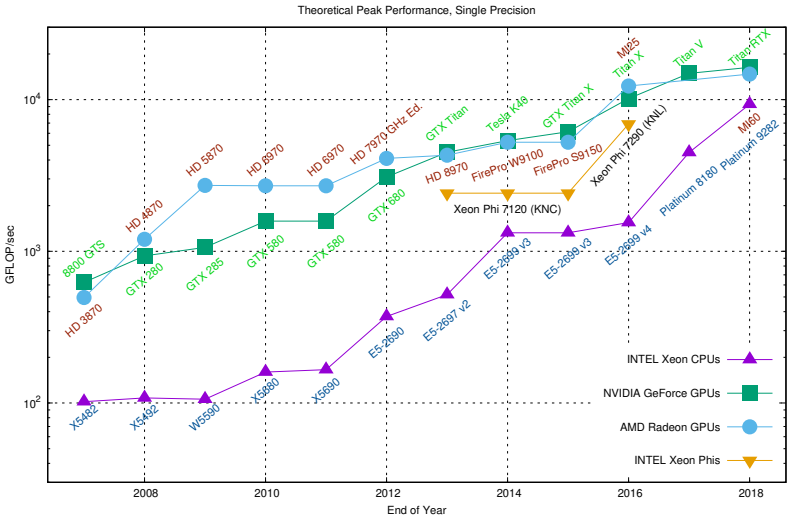
- GPUs are suited for **compute-intensive data-parallel** applications
  - The same program is executed for each data element
- Many-core chip
  - Multi-threaded execution on a single core (multiple threads executed concurrently by a core)
  - SIMD execution within a single core (many ALUs performing the same instruction)
- The focus is on overall **computing throughput** rather than on the speed of an individual core
  - GPUs do not reduce latency, they aim to **hide latency**
  - High arithmetic intensity and large number of schedulable units to hide latency of memory accesses

Key Insights in GPU Architecture

Much more transistors or real-estate is devoted to computation rather than data caching and control flow

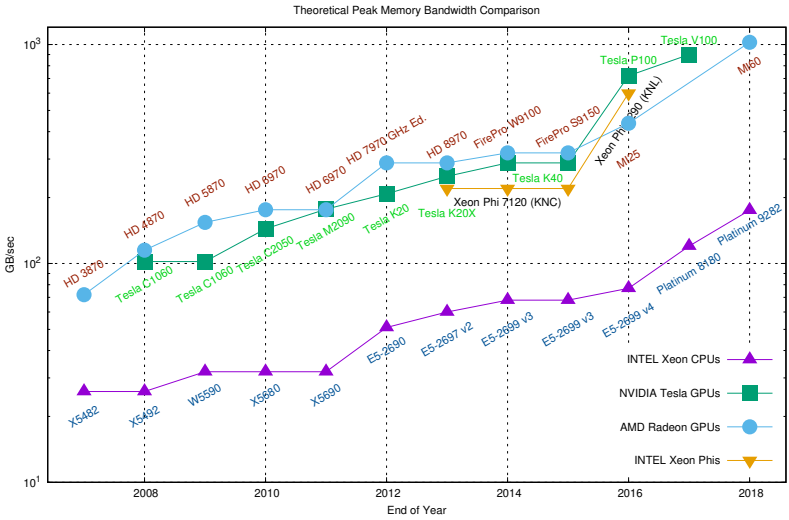


Comparing FLOPS



CPU, GPU and MIC Hardware Characteristics over Time

Memory Bandwidth for CPU and GPU



CPU, GPU and MIC Hardware Characteristics over Time

Compare GPU to CPU Architecture

- CPUs aim to **reduce memory latency** with increasingly large and complex memory hierarchy
- Disadvantages
  - The Intel I7-920 processor has some 8 MB of internal L3 cache, almost **30%** of the size of the chip
  - Larger cache structures increases the physical size of the processor
  - Implies more expensive manufacturing costs and increases likelihood of manufacturing defects
- Effect of larger, progressively more inefficient caches ultimately results in higher costs to the end user

Advantages of a GPU

Performance

- Comparing Xeon 8180M and Titan V (based on peak values)
  - 3.4X operations executed per second compared to the CPU
  - 5.5X bytes transferred from main memory per second compared to the CPU
  - Cost- and energy-efficiency
    - 15X as much performance per dollar
    - 2.8X as much performance per watt

Energy

- GPU's higher performance and energy efficiency are due to different allocation of chip area
  - High degree of SIMD parallelism, simple in-order cores, less control/sync. logic, less cache/scratchpad capacity
  - SIMD is more energy-efficient than MIMD since a single instruction can launch many data operations
  - Simpler pipeline with no support for restartable instructions and precise exceptions

Limitations of GPUs

Should we not use GPUs ALL the time?

- GPUs can only execute some types of code fast
  - SIMD parallelism is not well suited for all algorithms
  - Need lots of data parallelism, data reuse, and regularity
- GPUs are harder to program and tune than CPUs because of their architecture
  - Fewer tools and libraries exist

Role of CPUs

- CPU is responsible for initiating computation on the GPU and transferring data to and from the GPU
- Beginning and end of the computation typically require access to input/output (I/O) devices
- There are ongoing efforts to develop APIs providing I/O services directly on the GPU
  - GPUs are not standalone yet, assumes the existence of a CPU

CPUs vs GPUs

CPU

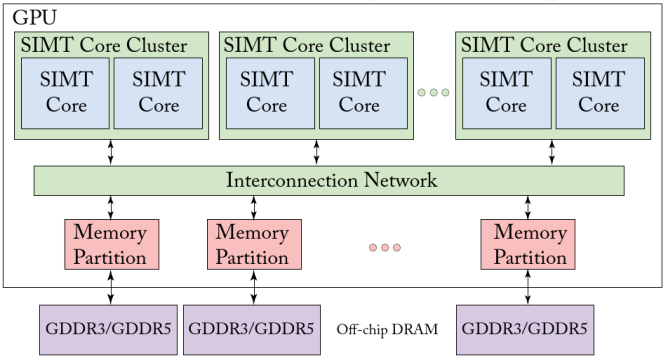
- Designed for running a few potentially complex tasks
  - Tasks may be unconnected
  - Suitable to run system software like the OS and applications
- Small number of registers per core private to a task
  - Context switch between tasks is expensive in terms of time
  - Register set must be saved to memory and the next one restored from memory

GPU

- Designed for running large number of simple tasks
  - Suitable for data parallelism
- Has a single set of registers but with multiple banks
  - A context switch involves setting a bank selector to switch in and out the current set of registers
  - Orders of magnitude faster than having to save to RAM

GPU Architecture

- GPUs consist of Streaming Multiprocessors (SMs)
  - NVIDIA calls these streaming multiprocessors and AMD calls them compute units
- SMs contain Streaming Processors (SPs) or Processing Elements (PEs)
  - Each core contains one or more ALUs and FPUs
- GPU can be thought of as a multi-multicore (manycore) system



Ampere Architecture

Ampere Architecture

Ampere Architecture

- A GA102 GPU includes 28.3 billion transistors with a die size of 628.4 mm^2
- Includes 7 Graphics Processing Clusters (GPCs), 42 (7*6) Texture Processing Clusters (TPCs), and 84 (7*12) Streaming Multiprocessors (SMs)
- Each SM in GA10x GPUs contain
  - 128 CUDA cores for a total of 84*128=10752 cores
  - 256 (4*16384*32 bits) KB register file
  - Combined 128 KB L1 data cache/shared memory subsystem
- In addition, there are 84 RT Cores, 336 Tensor Cores, 168 FP64 units (two per SM)
- The memory subsystem consists of twelve 32-bit memory controllers (384-bit total)
  - 512 KB of L2 cache is paired with each 32-bit memory controller, for a total of 6144 KB
- Includes PCIe Gen4 providing up to 16 Gigatransfers/second bit rate

NVIDIA Ampere GA102 GPU Architecture: Second-Generation RTX

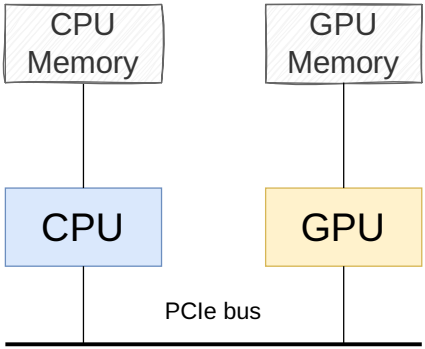
Compute Capability (CC)

- When programming with CUDA, it is very important to be aware of the differences among different versions of hardware
- In CUDA, CC refers to architecture features
  - For example, number of registers and cores, cache and memory size, and supported arithmetic instructions
- For example, CC 1.x devices have 16 KB local memory per thread, and 2.x and 3.x devices have 512 KB local memory per thread

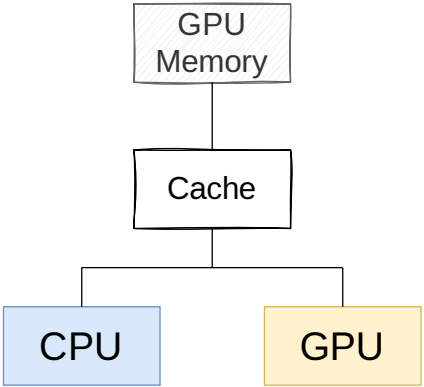
CUDA: Version features and specifications

Discrete vs Integrated GPUs

Discrete



Integrated



Discrete vs Integrated GPUs

Discrete

- More performant, consumes more energy
- Cost of PCIe transfers influences the granularity of offloading and the performance

Integrated

- Less performant because of energy considerations
- CPU and GPU share physical memory (DRAM or LLC) and can avoid the cost of data transfers over a PCIe bus

CUDA Programming

Programming API for NVIDIA GPUs

CUDA Philosophy

Massively parallel

The computations can be broken down into hundreds or thousands of independent units of work

Computationally intensive

The time spent on computation significantly exceeds the time spent on transferring data to and from GPU memory

Single Instruction Multiple Thread (SIMT)

CUDA Programming Model

Allows fine-grained data parallelism and thread parallelism nested within coarse-grained data parallelism and task parallelism

(i) Partition the problem into coarse sub-problems that can be solved independently

(ii) Assign each sub-problem to a “block” of threads to be solved in parallel

(iii) Each sub-problem is also decomposed into finer work items that are solved in parallel by all threads within the block

Hello World with CUDA

```
#include <stdio.h>
#include <cuda.h>

__global__ void hwkernel() {
  printf("Hello world!\n");
}

int main() {
  hwkernel<<<1, 1>>>();
}
```

```
$ nvcc hello-world.cu
$ ./a.out

$
```

hello-world.cu

Makefile

Hello World with CUDA

```
#include <stdio.h>
#include <cuda.h>

__global__ void hwkernel() {
  printf("Hello world!\n");
}

int main() {
  hwkernel<<<1, 1>>>();
  cudaDeviceSynchronize();
}
```

```
$ nvcc hello-world.cu
$ ./a.out
Hello world!

$
```

- CPU thread returns immediately after launching the kernel
- Use `cudaDeviceSynchronize()` (or its variants) to block the caller CPU thread

Function Declarations

	Executed on	Callable from
`__device__ float deviceFunc()`	Device	Device
`__global__ void kernelFunc()`	Device	Host[§]
`__host__ float hostFunc()`	Host	Host

- `__global__` define a kernel function, must return `void`
- `__device__` functions can have return values
- `__host__` is default, and can be omitted
- Prepending `__host__ __device__` causes the system to compile separate host and device versions of the function

[§] A kernel function can also be called from the device if dynamic parallelism is enabled

Classical Execution Model

(i) Prepare input data in CPU memory
(ii) Copy data from CPU to GPU memory (e.g., `cudaMemcpy()`)
(iii) Launch GPU kernel and execute, caching data on chip for performance
(iv) Copy results from GPU memory to CPU memory (e.g., `cudaMemcpy()`)
(v) Use the results on the CPU
(vi) Repeat the above steps based on the application logic

CUDA Extensions for C/C++

Kernel launch	● Calling functions on GPU
Memory management	● GPU memory allocation, copying data to/from GPU
Declaration qualifiers	● `__device__, __shared, __local, __global__, __host__`
Special instructions	● Barriers, fences, atomics
Keywords	● `threadIdx, blockIdx, blockDim`

Kernels

- Kernels are special functions that a CPU can call to execute on the GPU
  - Executed `N` times in parallel by `N` different CUDA threads
  - Cannot return a value
  - Each thread will execute `VecAdd()`
- Each thread has a unique thread ID that is accessible within the kernel through the built-in `threadIdx` variable

```
// Kernel definition
__global__ void VecAdd(float* A, float* B,
    float* C) {
  int i = threadIdx.x;
  ...
}
int main() {
  ...
  // Kernel invocation with N threads
  VecAdd<<<1, N>>>(A, B, C);
}
```

Kernels

```
KernelName<<<m, n>>>(arg1, arg2, ...)
```

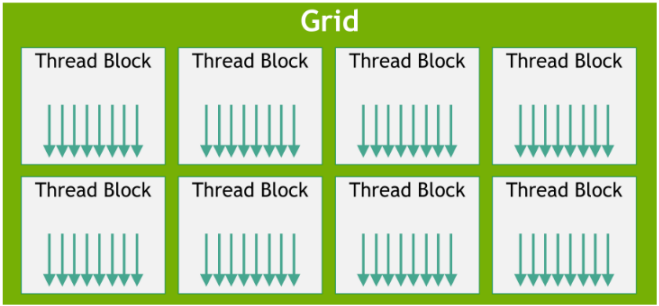
GPU spawns `m` blocks with `n` threads that run a copy of the same function

- CPU can continue processing while GPU runs the kernel
- Kernel call returns when all threads have terminated

```
kernel1<<<X,Y>>>(...);
// kernel starts execution, CPU continues to next statement
kernel2<<<X,Y>>>(...);
// kernel2 placed in queue, will start after kernel1 finishes,
// CPU continues
cudaMemcpy(...);
// CPU blocks until memory is copied, memory copy starts after all
// preceding CUDA calls finish
```

Thread Hierarchy

- A kernel executes in parallel across a set of parallel threads
- All threads that are generated by a kernel launch are collectively called a **grid**
- Threads are organized in **thread blocks**
- A thread block is a set of concurrently executing threads that can cooperate among themselves through shared memory and barrier synchronization
- A grid is an array of thread blocks that execute the same kernel



Dimension and Index Variables

Type is `dim3`

Dimension

- `gridDim` specifies the number of blocks in the grid
- `blockDim` specifies the number of threads in each block

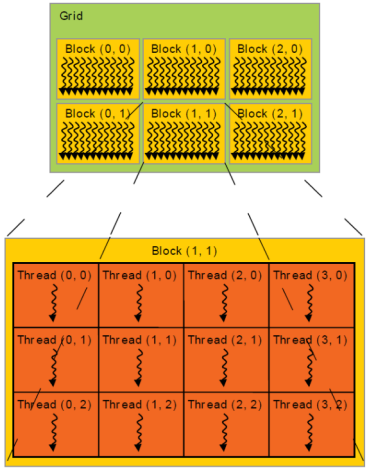
Index

- `blockIdx` gives the index of the block in the grid
- `threadIdx` gives the index of the thread within the block

- Keywords are 3-component vectors
  - For example, `threadIdx.[x,y,z]` gives the index of the thread in a block, in the particular direction
  - Thread index can be 1D, 2D, or 3D

Thread Hierarchy

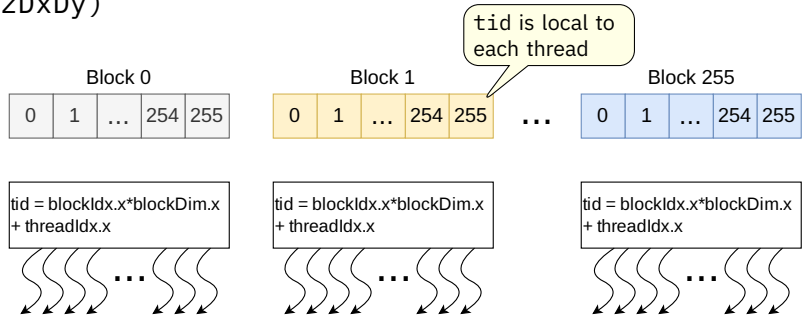
- Threads in a block reside on the same core, max 1024 threads in a block
- Thread blocks are organized into 1D, 2D, or 3D grids
  - Also called cooperative thread array (CTA)
  - Grid dimension is given by `gridDim` variable
- Identify block within a grid with the `blockIdx` variable
- Block dimension is given by `blockDim` variable



Finding Thread IDs

How to find out the relation between thread IDs and `threadIdx`?

- 1D: `tid = threadIdx.x`
- 2D block of size `(Dx, Dy)`: thread ID of a thread of index `(x, y)` is `(x + yDx)`
- 3D block of size `(Dx, Dy, Dz)`: thread ID of a thread of index `(x, y, z)` is `(x + yDx + zDxDy)`



thread-id.cu

Device Management

Query device information

- The application can query and select GPUs
  - `cudaGetDeviceCount(int *count)`
  - `cudaSetDevice(int device)`
  - `cudaGetDevice(int *device)`
  - `cudaGetDeviceProperties(cudaDeviceProp *prop, int device)`
- Multiple host threads can share a device
- A single host thread can manage multiple devices
  - `cudaSetDevice(i)` to select current device
  - `cudaMemcpy(...)` for peer-to-peer copies

Launching Kernels

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  C[i][j] = A[i][j] + B[i][j];
}
int main() {
  ...
  // Kernel invocation
  dim3 threadsPerBlock(16, 16);
  dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
  MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
  ...
}
```

Execution Configuration Uses Integer Arithmetic

- Assume data is of length `N`, and say the kernel execution configuration is `<<<N/TPB, TPB>>>`
  - Each block has `TPB` threads and there are `N/TPB` blocks
- Dimension variables are vectors of integral type

Suppose `N = 64` and `TPB = 32`

Implies there are 2 blocks of 32 threads

Suppose `N = 65` and `TPB = 32`

Implies there are 2 blocks of 32 threads

Execution Configuration Uses Integer Arithmetic

- Assume data is of length `N`, and say the kernel execution configuration is `<<<N/TPB, TPB>>>`
  - Each block has `TPB` threads and there are `N/TPB` blocks
- Dimension variables are vectors of integral type

- Ensure that the grid covers the array length by rounding up the number of blocks from `N/TPB` to `(N+TPB-1)/TPB`
- Use a control statement in the kernel to ensure that the thread index is within the maximum array index

Choosing Optimal Execution Configuration

- The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system
  - It is okay to have a greater number of threads
- No fixed rule, needs exploration and experimentation
- Choose number of threads in a block to be a multiple of 32

Timing a CUDA Kernel

```
float memsettime;
cudaEvent_t start, stop;
// initialize CUDA timers
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start,0);
// CUDA Kernel
...
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&memsettime,start,stop); // in milliseconds
cout << "Kernel execution time: " << memsettime << "\n";
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

How to Implement Performance Metrics in CUDA C/C++

Reporting Errors

- All CUDA API calls return an error code of type `cudaError_t`
  - Error in the API call itself or error in an earlier asynchronous operation (e.g. kernel)
- Get the error code for the last error with `cudaGetLastError()`
- Get a string to describe the error with `char *cudaGetErrorString(cudaError_t)`

What is the canonical way to check for errors using the CUDA runtime API?

Dynamic Parallelism

- Data intensive irregular applications can result in load imbalance across kernel threads, potentially under-utilizing the GPU
- It is possible to launch kernels from other kernels
- Calling `__global__` functions from the device is referred to as dynamic parallelism
  - Requires CUDA devices of CC 3.5 (Kepler microarchitecture) and CUDA 5.0 or higher

Warp Scheduling

SIMT Architecture

- GPUs employ SIMD hardware to exploit the data-level parallelism
  - In SIMD, we program with the vector width in mind
  - In vectorization, users program the SIMD hardware directly, or uses auto-vectorization or intrinsics
- SIMT can be thought of as SIMD with multithreading
  - Software analog compared to the hardware perspective of SIMD
  - For example, we rarely need to know the number of cores with CUDA

SIMT Architecture

- CUDA also features a MIMD-like programming model
  - Launch large number of threads
  - Each thread can have its own execution path and access arbitrary memory locations
- This execution model is called single-instruction multiple-thread (SIMT)
- Two levels of parallelism
  - Independent grids (i.e., kernels) or concurrent thread blocks represent coarse-grained data parallelism or task parallelism
  - Concurrent threads/warps represent fine-grained data parallelism or thread parallelism

SIMD vs SPMD

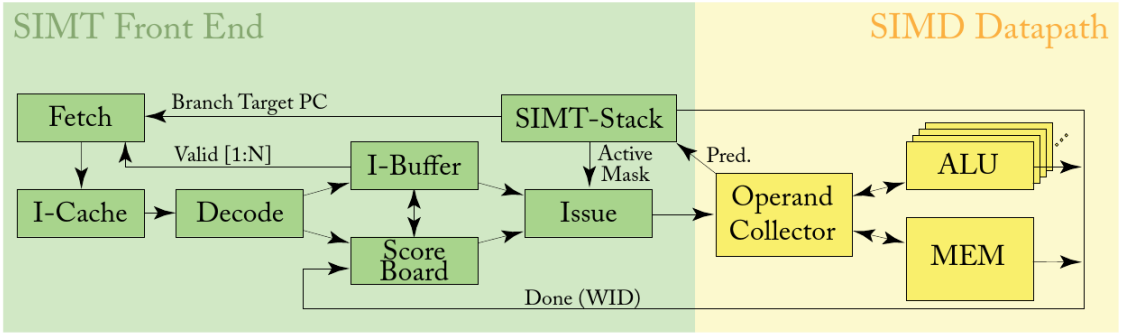
SIMD

- Processing units are executing the same instruction at any instant

SPMD

- Parallel processing units execute the same program on multiple parts of the data
- All the processing units may not execute the same instruction at the same time

Core Microarchitecture



Mapping Blocks and Threads

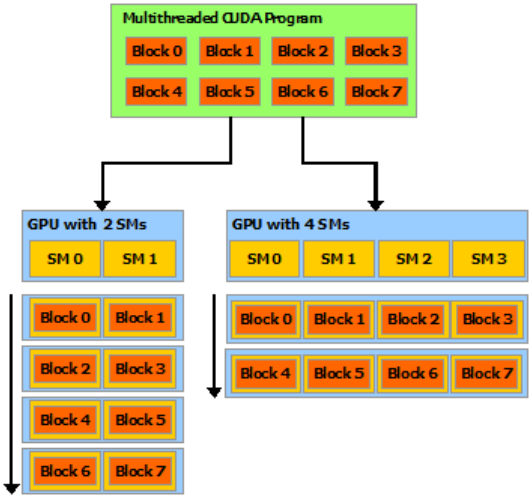
- A GPU executes one or more kernel grids
- A kernel is partitioned into thread blocks that execute independently of each other
- When a CUDA kernel is launched, the thread blocks are distributed to SMs in any order
  - Multiple thread blocks, up to a limit, can execute concurrently on one SM
    - Not all blocks may be resident at the same time
  - For example, a CUDA device may allow up to eight blocks to be assigned to each SM
  - CUDA cores in the SM execute threads of a block
- A block begins execution only when it has secured all execution resources necessary for all the threads
- As thread blocks terminate, new blocks are launched on the vacated SMs

CUDA runtime can execute blocks in any order

- Blocks are mostly **not** supposed to synchronize with each other
  - Allows for simple hardware support for data parallelism

Block Scalability

- A kernel with enough blocks scales across GPUs
- A GPU with more SMs will automatically execute the program in less time than a GPU with fewer SMs

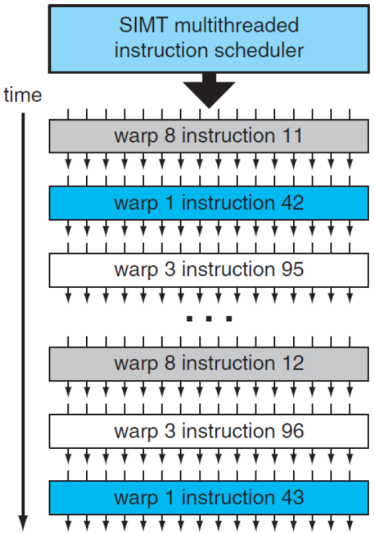


Thread Warps

- Conceptually, threads in a block can also execute in any order
- However, sharing a control unit reduces hardware complexity, cost, and power consumption
- A set (currently 32) of consecutive threads that execute in SIMD fashion is called a **warp**
  - Called wavefront (with 64 threads) on AMD GPUs
- Warps are scheduling units in an SM
  - Implementation detail, not part of the programming model

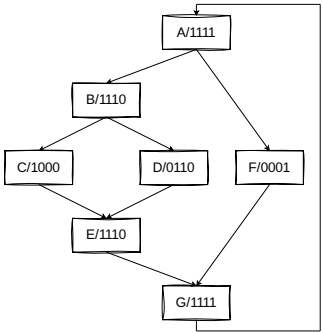
Lockstep Execution

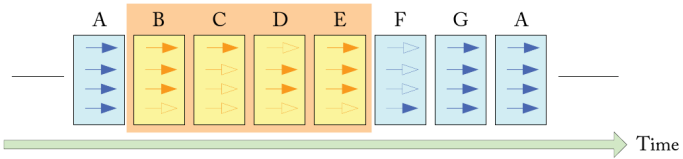
- All threads in a warp run in lockstep
- Warps share an instruction stream, i.e., same instruction is fetched for all threads in a warp during the instruction fetch cycle
  - Prior to Volta, warps used a single shared program counter
- In the execution phase, each thread will either execute the instruction or will execute nothing
- Individual threads in a warp have their own instruction address counter and register state
- Warp threads are fully synchronized, i.e., there is an implicit barrier after each step/instruction



Thread Divergence

- If some threads take the then branch and other threads take the else branch, they cannot operate in lockstep
  - Some threads must wait for the others to execute, serializes execution at that point
- The programming model does not prevent thread divergence
- Divergence occurs only within a warp





SIMT Stack

How does GPU hardware enable threads within a warp to follow different code paths?

TOS	Reconv PC	Next PC	Active Mask
	–	G	1111
	G	B	1110
→	G	F	0001

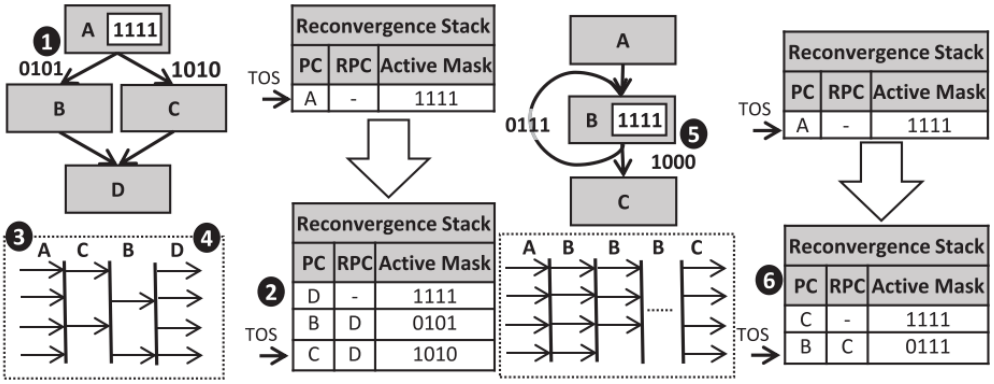
TOS	Reconv PC	Next PC	Active Mask
	–	G	1111
	G	E	1110
	E	D	0110
→	E	C	1000

TOS	Reconv PC	Next PC	Active Mask
	–	G	1111
→	G	E	1110

Is the order important?

Threads that diverge can be forced to continue executing in lockstep from a reconvergence point

Stack-Based Reconvergence



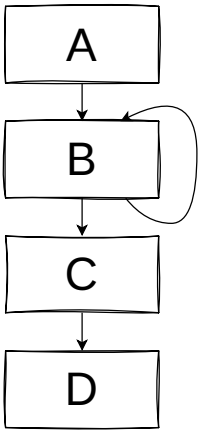
(a) If-else Example (b) Loop Example

A. ElTantawy and T. M. Aamodt. MIMD Synchronization on SIMT Architectures, MICRO'16.

SIMT Deadlock

Stack-based implementation of SIMT execution can lead to a deadlock

```
A: *mutex=0;
B: while(!atomicCAS(mutex, 0, 1)) {}
C: // critical section
D: atomicExch(mutex, 0);
```

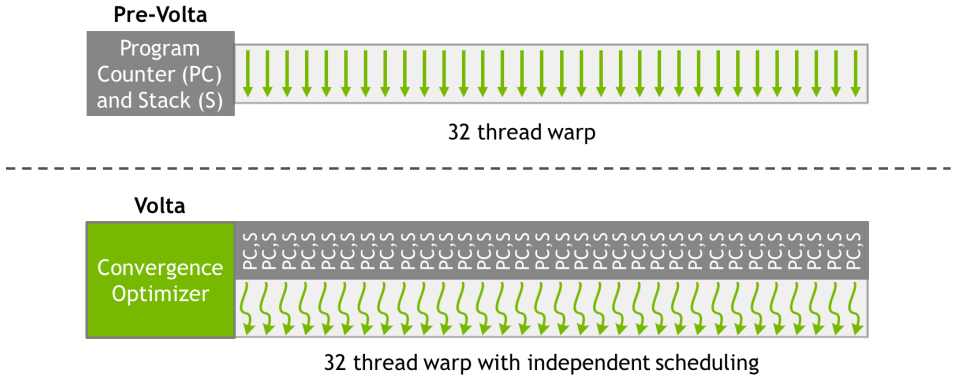


Stack-based SIMT execution constrains thread scheduling

- Serialization—The threads in the taken branch block until the threads in the not-taken branch reach the reconvergence point (or vice versa)
- Forced reconvergence—When the threads in the taken branch reach the reconvergence point, they block waiting for the threads in the not-taken branch to reach the reconvergence point (or vice versa)

Volta SIMT Model

- Architectures older than Volta maintained a shared program counter (PC) and call stack per warp
- Volta onward, the execution state (i.e., PC and call stack) is maintained per thread



Inside Volta: The World's Most Advanced Data Center GPU

Independent Thread Scheduling (ITS)

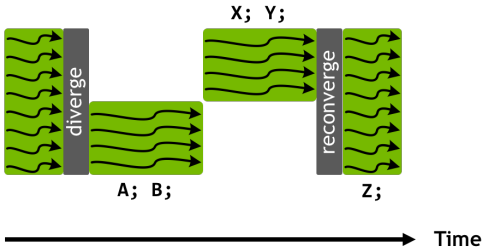
- Volta architecture introduces ITS among threads in a warp
- ITS allows intra-warp synchronization which was previously not possible
- Threads can now diverge and reconverge at sub-warp granularity
- ITS makes it easy to implement complex, fine-grained algorithms and data structures

- The SIMT stack is replaced with per warp convergence barriers
  - The metadata includes barrier participation mask, barrier convergence state, and per-thread states like PC and active status

SIMT Models across NVIDIA GPU Architectures

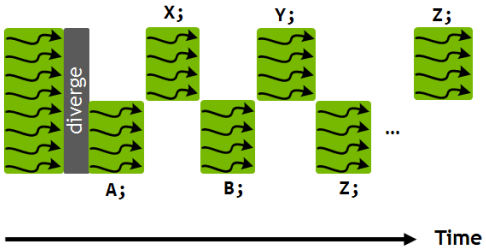
Till Pascal

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



Volta onward

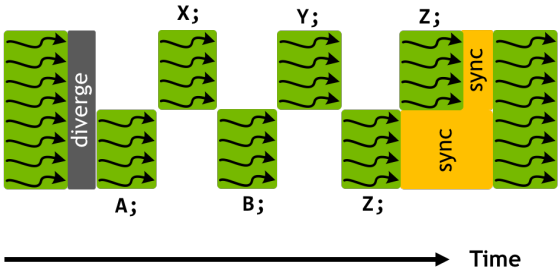
```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



Intra-warp Synchronization

Use `__syncwarp()` to reconverge threads within a warp

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```



Scheduling Thread Warps

- Each SM launches warps of threads, and executes warps on a time-sharing basis
  - Time-sharing is implemented in hardware, not software
- SM schedules and executes warps that are ready to run
  - Warps run for fixed-length time slices like processes
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Selection of ready warps for execution does not introduce any idle time into the execution timeline, called zero-overhead scheduling
  - If more than one warp is ready for execution, a priority mechanism is used to select one for execution

Scheduling Thread Warps

- Suppose an instruction executed by a warp has to wait for the result of a previously initiated long-latency operation
  - The warp is not selected for execution, another warp that is not waiting for results is selected for execution
- Goal is to have enough threads and warps around to utilize hardware in spite of long-latency operations
  - GPU hardware will likely find a warp to execute at any point in time
  - Hides latency of long operations with work from other threads, called latency tolerance or latency hiding
- Thread blocks execute on an SM, thread instructions execute on a core
- CUDA virtualizes the physical hardware
  - Thread is a virtualized scalar processor (registers, PC, state)
  - Block is a virtualized multiprocessor (threads, shared memory)
- As warps and thread blocks complete, resources are freed

Warp Scheduling

- For an ideal memory system, increasing the number of warps can increase the throughput
  - If the number of warps in a core multiplied by the issue time of each warp exceeds the memory latency, the execution units in the core will always remain busy
  - Round-robin scheduling of warps would suffice
  - However, locality property either favors or discourages round-robin scheduling
- Increasing the number of warps is impractical because of the need to maintain execution state in hardware (i.e., to achieve zero-overhead scheduling)
- Number of threads that can be simultaneously tracked and scheduled in hardware is bounded
  - Requires resources for an SM to maintain execution status of threads
- Up to 2048 threads can be assigned to each SM on recent CUDA devices
  - For example, 8 blocks of 256 threads, or 4 blocks of 512 threads
- Assume a CUDA device with 28 SMs
  - Each SM can accommodate up to 2048 threads
  - The device can have up to 57344 threads simultaneously residing in the device for execution

Memory Hierarchy

Memory Access Efficiency

Compute-to-global-memory access ratio is the number of floating-point operations performed for each access to global memory

```
for (int i = 0; i < N; i++)
  tmp += A[i*N+K]*B[k*N+j];
```

Assume a GPU device with 1 TB/s global memory bandwidth and peak single-precision performance of 12 TFLOPS

- What is the performance we expect with an access ratio of 1?
- We can do 1000/4 GFLOPS, which is only $\sim$ 2% of the peak performance

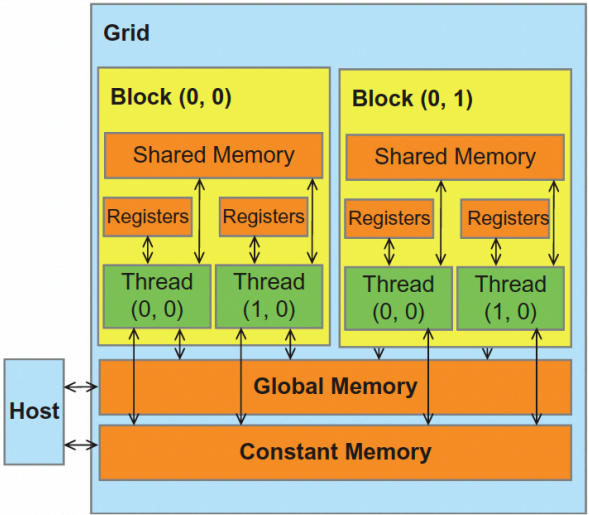
Memory Hierarchy in CUDA

Device code can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

Host code can

- Transfer data to/from per grid global and constant memories



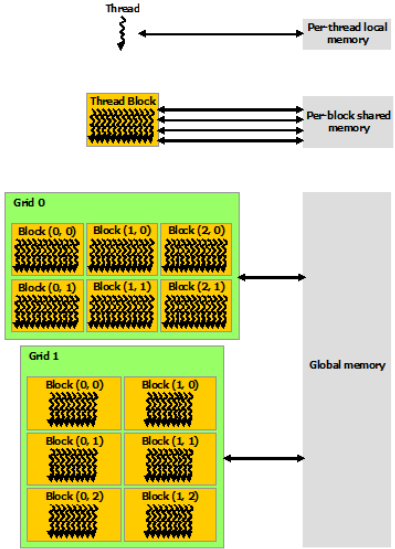
Variable Type Qualifiers in CUDA

	Memory	Scope	Lifetime
`int localVar`	Register	Thread	Kernel
`__device__ __local__ int localVar`	Local	Thread	Kernel
`__device__ __shared__ int sharedVar`	Shared	Block	Kernel
`__device__ int globalVar`	Global	Grid	Application
`__device__ __constant__ int constVar`	Constant	Grid	Application

- `__device__` is optional when used with `__local__`, `__shared__`, or `__constant__` except arrays that reside in local memory
- Pointers can only point to memory allocated or declared in global memory

Memory Organization

- Host and device maintain their own separate memory spaces
  - A variable in CPU memory may not be accessed directly in a GPU kernel
- It is the programmer's responsibility to keep them in sync
  - A programmer needs to maintain copies of variables



Registers

- 64K 32-bit registers (or 256 KB register file) per SM
  - A CPU in contrast has a few (1–2 KB) per core
- Up to 255 registers per thread (compute capability 3.5+)
- If a code uses the maximum number of registers per thread (255) and an SM has 64K registers, then the SM can support a maximum of 256 threads
- If we use the maximum allowable number of threads per SM (2048), then each thread can use at most 32 registers per thread

Registers

- 64K 32-bit registers (or 256 KB register file) per SM
  - A CPU in contrast has a few (1–2 KB) per core
- Up to 255 registers per thread (compute capability 3.5+)
- If a code uses the maximum number of registers per thread (255) and an SM has 64K registers, then the SM can support a maximum of 256 threads
- If we use the maximum allowable number of threads per SM (2048), then each thread can use at most 32 registers per thread

What if each thread uses 33 registers?

Registers

- If we use the maximum allowable number of threads per SM (2048), then each thread can use at most 32 registers per thread
- What if each thread uses 33 registers?
  - Fewer threads $\Rightarrow$ fewer warps
- There is a big difference between “fat” threads which use lots of registers, and “thin” threads that require very few!

Shared Memory

- Shared memory (also called scratchpad memory) is used for efficient communication among threads in a block
  - Usually 16–64 KB of storage that can be accessed efficiently by all threads in a block
- Each SM contains a single shared memory structure that acts as a software-managed cache
  - Resides adjacent to an SM on chip
  - The space is shared among all blocks running on that SM

- Variable in shared memory is allocated using the `__shared__` specifier
  - Latency is comparable to accessing registers
- Amount of shared memory per block limits occupancy

Say an SM with 4 thread blocks has 16 KB of shared memory

```
__shared__ float min[256];
__shared__ float max[256];
__shared__ float avg[256];
__shared__ float stdev[256];
```

Global Variables

- Variable `lock` can be accessed by both kernels
  - Resides in global memory space
  - Can be both read and modified by all threads

```
__device__ int lock=0;
__global__ void kernel1(...) {
  // Kernel code
}
__global__ void kernel2(...) {
  // Kernel code
}
```

Global Memory

- On-device memory accessed via 32, 64, or 128 B transactions
- A warp executes an instruction that accesses global memory
  - The addresses are coalesced into transactions
  - Number of transactions depend on the access size and distribution of memory addresses
  - More transactions mean less throughput
    - For example, if 32 B transaction is needed for a thread's 4 B access, throughput is essentially 1/8th

Constant Memory

- Used for data that will not change during kernel execution
  - Accessible from all threads within a grid
  - Constant memory is 64 KB

```
// Global scope
__constant__ float d_filter[FILTER_WIDTH];
// Initialize array in constant memory
cudaMemcpyToSymbol(d_filter, h_filter, FILTER_WIDTH * sizeof(
    float));
```

- Constant memory is aggressively cached
  - Each SM has a read-only constant cache that is shared by all cores in the SM
  - Used to speed up reads from the constant memory space which resides in device memory
  - Read from constant memory incurs a memory latency on a miss
  - Otherwise, it is a read from constant cache, which is almost as fast as registers

Local Memory

- Local memory is off-chip per-thread memory
  - More like thread-local global memory, so it requires memory transactions and consumes bandwidth
- Automatic variables are placed in local memory
  - (i) Arrays when it is not known whether indices are constant quantities
  - (ii) Large structures or arrays that consume too much register space
  - (iii) In case of register spilling
- Inspect PTX assembly code (compile with `-ptx`)
  - Check for `ld.local` and `st.local` mnemonic

Device Memory Management

- Global device memory can be allocated with `cudaMalloc()`
- Freed by `cudaFree()`
- Data transfer between host and device is with `cudaMemcpy()`
- Initialize memory with `cudaMemset()`
- There are asynchronous versions

GPU Caches

- GPUs have L1 and L2 data caches on devices with CC 2.x and higher
  - Texture and constant cache are available on all devices
- L1 cache is write-through, and per SM
  - Shared memory is partitioned out of unified data cache and its size can be configured, remaining portion is the L1 cache
  - Can be configured as 48 KB of shared memory and 16 KB of L1 cache, or 16 KB of shared memory and 48 KB of L1 cache, or 32 KB each
  - L1 caches are 16–48 KB
- L2 cache is shared by all SMs
- L1 cache lines are 128 B wide in Fermi onward, while L2 lines are 32 B

CPU Caches vs GPU Caches

CPU Cache

- Data is automatically moved by hardware between caches
  - Association between threads and cache does not have to be exposed to programming model
- Caches are generally coherent

GPU Cache

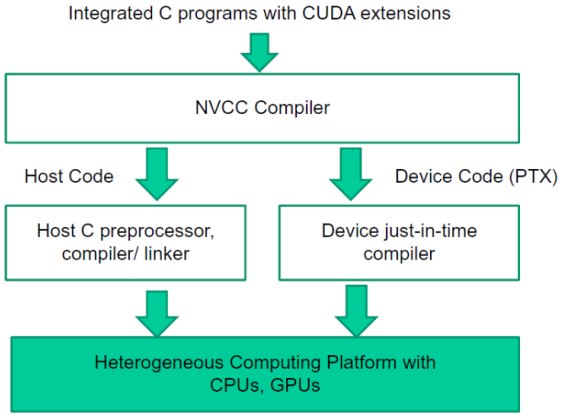
- Data movement must be orchestrated by programmer
  - Association between threads and storage is exposed to programming model
- L1 cache is not coherent, L2 cache is coherent

CUDA Compilation

Binary compatibility of GPU applications is not guaranteed across different generations

How NVCC works

- Nvcc is a driver program based on LLVM
  - Compiles and links all input files
  - Requires a general-purpose C/C++ host compiler
  - Uses GCC and G++ by default on Linux platforms



NVIDIA CUDA Compiler Driver NVCC

Selected NVCC Options and File Types

Options	Description
`-std {c++11|...}`	Select a particular C++ dialect
`-m {32|64}`	Specify the architecture
`-G`	Generate debug information for device code, turns off device optimizations
`-arch ARCH`	Specify the **virtual** GPU architecture (`sm_52` is the default)
`-code CODE`	Specify the real GPU architecture to assemble and optimize for

File Type	Description
`.cu`	CUDA source file
`.c,.cpp,.cxx,.cc`	C/C++ source files
`.ptx`	PTX intermediate assembly
`.cubin`	CUDA device binary code for a single GPU architecture
`.fatbin`	CUDA fat binary file that may contain multiple PTX and CUBIN files

CUDA Compilation Trajectory

(i) Input program is preprocessed for device compilation

(ii) It is compiled to a CUDA binary (.cubin) and/or PTX (Parallel Thread Execution) intermediate code which are encoded in a fatbinary

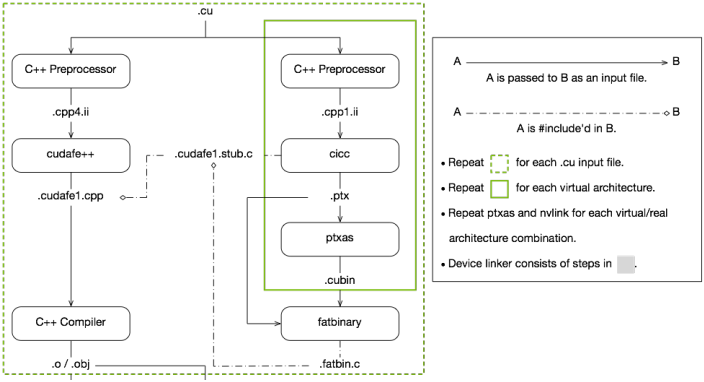
(iii) Input program is processed for compilation of the host code

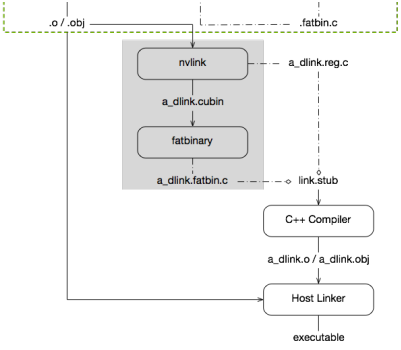
(iv) CUDA-specific C++ constructs are transformed to standard C++ code

(v) Synthesized host code and the embedded fatbinary are linked together to generate the executable

- A compiled CUDA device binary includes
  - Program text (instructions)
  - Information about the resources required
    - N threads per block
    - X bytes of local data per thread
    - M bytes of shared space per block

CUDA Compilation Trajectory

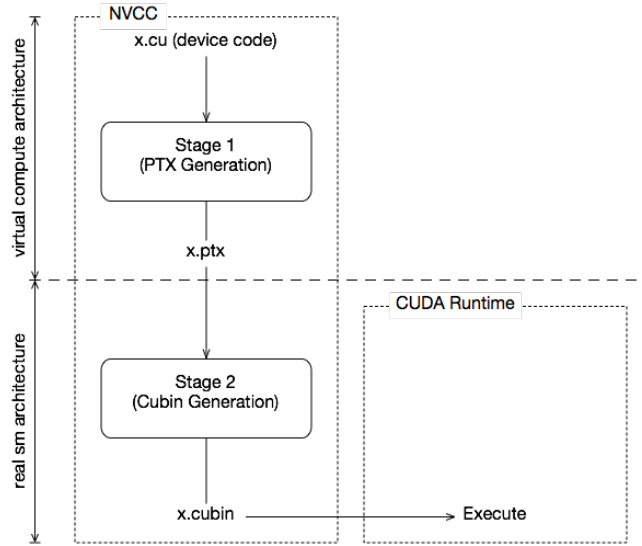




Binary Application Compatibility

- NVIDIA does not guarantee binary compatibility of GPU applications across different generations
  - For example, a CUDA application compiled for a Fermi GPU will very likely not run on a Kepler GPU (and vice versa)
  - Instruction set and instruction encodings of a generation is different from those of other generations
- nvcc relies on a two stage compilation model for ensuring application compatibility across GPU generations
  - Code is compiled to a virtual assembly called PTX
  - PTX code is assembled for a real GPU architecture
- Recommendation
  - Specify a lower virtual architecture to improve portability and a higher real architecture for improved performance

Two-Staged Compilation with Virtual and Real Architectures



PTX Assembly and SASS

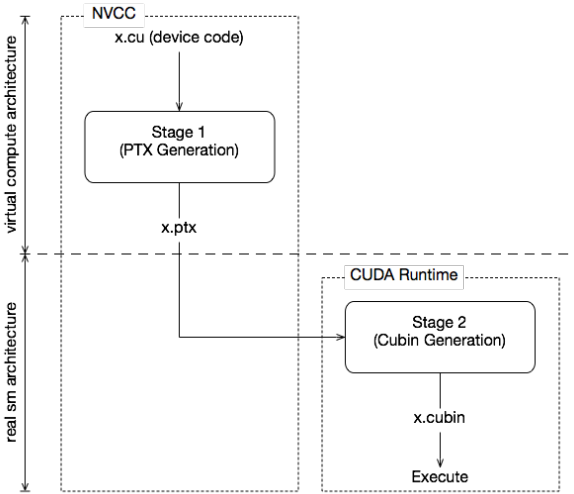
- GPU compilation is performed via an intermediate representation PTX
  - PTX is an acronym for Parallel Thread eXecution
  - Goal is to provide an architecture-indent ISA for compilers
  - Can be considered as assembly for a virtual GPU architecture

- SASS is the low-level assembly that compiles to binary microcode which executes natively on NVIDIA GPUs
  - SASS is an acronym for Source and ASSembly

Parallel Thread Execution ISA Version 8.5

Improving Application Portability with JIT Compilation

- Two-stage compilation does not help improve application portability
  - Generated code is bound to one GPU generation (e.g., `sm_53`)
- There are two strategies to improve portability
  - (i) nvcc will postpone assembly of PTX code until application run time if only a virtual GPU architecture is specified
  - (ii) Generate multiple translations for different architectures and embed the CUDA binaries in a fat binary



NVCC Examples

`nvcc –arch=compute_30 –code=sm_52 hello-world.cu`

Generate PTX assuming CC 3.0 and generate binary SASS code compliant with CC 5.2

`nvcc –arch=compute_30 –code=sm_30,sm_52 hello-world.cu`

Generate binary SASS for two GPU architectures and embed the cubin files in the executable

`nvcc -arch=compute_50 hello-world.cu`

Implies `nvcc -arch=compute_50 -code=compute_50 hello-world.cu`

`nvcc -arch=sm_52 hello-world.cu`

- Implies `nvcc -arch=compute_52 -code=sm_52,compute_52 hello-world.cu`
- Embed both the PTX and the SASS code in the final binary

Synchronization in CUDA

Race Conditions and Data Races

- A race condition occurs when program behavior depends upon relative timing of two (or more) event sequences
  - (i) Read value at address c
  - (ii) Add sum to value
  - (iii) Write result to address c
- There can be intra-warp, inter-warp, and inter-block races
  - Intrawarp races occur when threads from the same warp write to the same memory location

```
__global__ void intrawarp ( unsigned * data ) {
  data [0] = threadIdx .x;
}
```

`__syncthreads()`

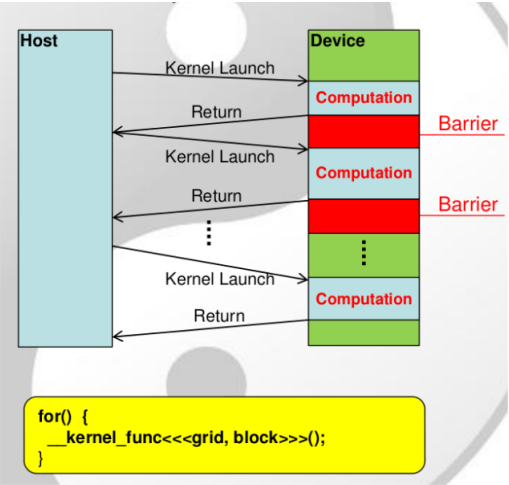
- `__syncthreads()` synchronizes threads within a block
- All global and shared memory accesses made by the participating threads prior to `__syncthreads()` are visible to all threads in the block

- A `__syncthreads()` statement must be executed by **all** threads in a block
- If `__syncthreads()` is in an `if` statement, then either all threads in the block execute the `then` path that includes the `__syncthreads()` or none of them does
- If `__syncthreads()` statement is in each path of an `if-then-else` statement, then either all threads in a block execute the `__syncthreads()` on the `then` path or all of them execute the `else` path
  - The two `__syncthreads()` are different barrier synchronization points

Grid-level Synchronization

- `cudaDeviceSynchronize()` synchronizes all threads in a grid
  - There are other variants

- For threads from different grids, writes from a kernel happen before reads from subsequent grid launches



Atomic Operations

- Atomic operations allow performing atomic read-modify-write (RMW) operations on data residing in global or shared memory
  - Prevent data races associated with multiple threads concurrently accessing a global or shared memory variable
  - Will give predictable results when simultaneous access to memory is required
  - For example, `atomicAdd()`, `atomicSub()`, `atomicMin()`, `atomicMax()`, `atomicInc()`, `atomicDec()`, `atomicExch()`, `atomicCAS()`

Scoped Synchronization

- CUDA provides scope qualifiers that limit the subset of the threads that are guaranteed to observe the synchronization
- CUDA exposes three scopes: block, device, and system
- For example, an atomic RMW operation with block scope is only visible to threads within the same thread block (e.g., `atomicAdd_block()`)

CUDA Memory Model

- A **memory consistency model** is a set of rules that govern how systems process memory operation requests from multiple processors
  - Determines the order in which memory operations appear to execute
  - Specifies the allowed behaviors of multithreaded programs executing with shared memory
  - Memory models are defined both for hardware and programming languages

- CUDA implements a weakly-ordered memory model
- The order in which a thread writes data is not necessarily the order in which the data is observed being written by another CUDA or host thread
- The behavior on concurrent and conflicting accesses is undefined

Weakly-Ordered Memory Model in CUDA

```
__device__ int X = 1, Y = 2;
```

Thread 1

```
__device__ void writeXY() {
  X = 10;
  Y = 20;
}
```

Thread 2

```
__device__ void readXY() {
  int B = Y;
  int A = X;
}
```

- A racy program has undefined behavior, and has no defined semantics
- The resulting values for A and B can be anything

Memory Fences

Semantics of `__threadfence_block()`

- All writes made by the calling thread before the call are observed by all threads in the block as occurring before all writes made by the calling thread after the call
- All reads from all memory made by the calling thread before the call are ordered before all reads from all memory made by the calling thread after the call

```
__device__ int X = 1, Y = 2;
```

Thread 1

```
__device__ void writeXY() {
  X = 10;
  __threadfence();
  Y = 20;
}
```

Thread 2

```
__device__ void readXY() {
  int B = Y;
  __threadfence();
  int A = X;
}
```

What are possible outcomes?

`__syncthreads()` vs `__threadfence()`

`__syncthreads()`

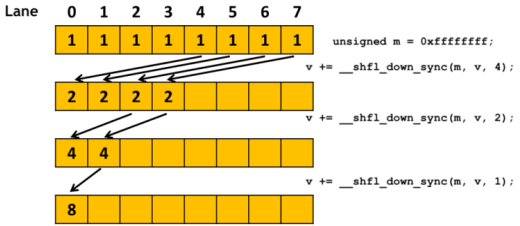
- Establishes **visibility** of memory operations across threads
- Includes a barrier plus memory fence functionality (i.e., `__threadfence_block()`)

`__threadfence()`

- Ensures **ordering** of memory operations by a thread
- `__threadfence_block()` does not imply `__syncthreads()`

Warp-Level Primitives

- `_ballot_sync(mask, predicate)`
  - Evaluate `predicate` for all non-exited threads in `mask` and return an integer whose N^{th} bit is set if `predicate` evaluates to non-zero for the N^{th} thread
- `__activemask()`
  - Returns a 32-bit integer mask of all currently active threads in the calling warp



Copy a variable from a warp lane with higher ID relative to caller

- `__syncwarp(mask)`
  - Guarantees memory ordering among threads participating in the barrier
  - Threads within a warp that wish to communicate via memory can store to memory, execute `__syncwarp()`, and then safely read values stored by other threads

Concurrency and CUDA Streams

Overlap host and device computation with data transfers

Classic Copy-then-Execute Model

```
1 cudaMemcpy(d_a, h_a, numBytes, cudaMemcpyHostToDevice);
2 kernel<<<1,N>>>(d_a);
3 cudaMemcpy(h_res, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- Data transfer on line 1 is blocking or synchronous
  - Host thread cannot launch the kernel until the copy is done
- Kernel launch on line 2 is asynchronous
- Data transfer on line 3 cannot begin until the kernel completes due to device-side ordering

How to Overlap Data Transfers in CUDA C/C++

GPU Pro Tip: CUDA 7 Streams Simplify Concurrency

Overlap Host and Device Computation

```
1 cudaMemcpy(d_a, h_a, numBytes, cudaMemcpyHostToDevice);
2 kernel<<<1,N>>>(d_a);
3 cudaMemcpy(h_res, d_a, numBytes, cudaMemcpyDeviceToHost);
```

```
1 cudaMemcpy(d_a, h_a, numBytes, cudaMemcpyHostToDevice);
2 kernel<<<1,N>>>(d_a);
3 // Host gets work done
4 h_func(h_b);
5 cudaMemcpy(h_res, d_a, numBytes, cudaMemcpyDeviceToHost);
```

Utilize GPU Hardware

- Overlap kernel execution with memory copy between host and device
- Overlap execution of multiple kernels if there are enough resources
- Recent GPUs support overlapped execution
  - Check the fields `asyncEngineCount`, `concurrentKernels`, and `deviceOverlap` from the `cudaDeviceProp` structure

CUDA Streams

- A stream is a sequence of operations that execute on the device in the order in which they were issued by the host
  - Operations across streams can interleave and run concurrently
- All GPU device operations run in a stream
- The default “null” stream is used if no custom stream is specified, the default stream is synchronizing
  - No operation in the default stream will begin until all previously issued operations in any stream have completed
  - An operation in the default stream must complete before any other operation in any stream will begin

```
// Both launches are on the default stream
kernel1<<< blocks, threads, bytes >>>();    // default stream
kernel2<<< blocks, threads, bytes, 0 >>>(); // stream 0
```

How to Overlap Data Transfers in CUDA C/C++

Using a Non-default Stream

Manipulate non-default streams from the host

```
cudaStream_t stream1;
cudaError_t result;
result = cudaStreamCreate(&stream1);
result = cudaStreamDestroy(stream1);
```

Issue a data transfer to a non-default stream

```
result = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1);
```

Specifying a stream during kernel launch is optional

```
kernel1<<<blocks, threads, bytes>>>(); // default/NULL stream
kernel2<<<blocks, threads, bytes, stream1>>>();
```

How to Overlap Data Transfers in CUDA C/C++

Non-default Streams

- Operations in a non-default stream are non-blocking with the host
- Use `cudaDeviceSynchronize()`
  - Blocks host until all previously issued operations on the device have completed
- Cheaper alternatives
  - `cudaStreamSynchronize(), cudaEventSynchronize(),...`

```
1 cudaStream_t stream1;
2 cudaError_t res;
3 res = cudaStreamCreate(&stream1);
4 res = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1);
5 increment<<<1,N,0,stream1>>>(d_a);
6 // Blocks the host thread
7 cudaStreamSynchronize(stream1);
8 res = cudaStreamDestroy(&stream1);
```

Overlapping Kernel Execution and Data Transfers

```
1 for (int i = 0; i < nStreams; ++i) {
2   int offset = i * streamSize;
3   cudaMemcpyAsync(&d_a[offset], &h_a[offset], streamBytes,
        cudaMemcpyHostToDevice, stream[i]);
4   kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
5   cudaMemcpyAsync(&h_a[offset], &d_a[offset], streamBytes,
        cudaMemcpyDeviceToHost, stream[i]);
6 }
```

How to Overlap Data Transfers in CUDA C/C++

Overlapping Kernel Execution and Data Transfers

```
1  for (int i = 0; i < nStreams; ++i) {
2    int offset = i * streamSize;
3    cudaMemcpyAsync(&d_a[offset], &h_a[offset], streamBytes,
         cudaMemcpyHostToDevice, stream[i]);
4  }
5  for (int i = 0; i < nStreams; ++i) {
6    int offset = i * streamSize;
7    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
8  }
9  for (int i = 0; i < nStreams; ++i) {
10   int offset = i * streamSize;
11   cudaMemcpyAsync(&h_a[offset], &d_a[offset], streamBytes,
         cudaMemcpyDeviceToHost, stream[i]);
12 }
```

Streams and Concurrency in CUDA 7+

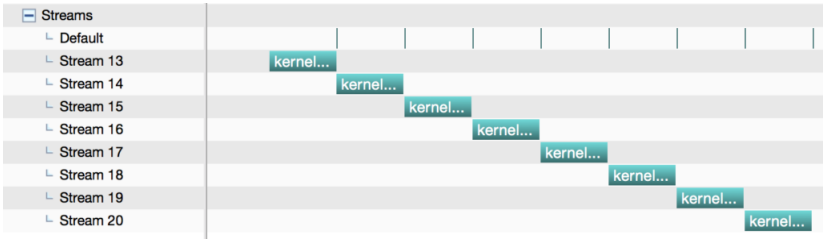
- Prior to CUDA 7, all host threads shared the default stream
  - The default stream implicitly synchronizes with all other streams on the device
  - Two commands from different streams cannot run concurrently if the host thread issues any CUDA command to the default stream between them

- CUDA 7+ provides an option to have a per-host-thread default stream
  - Commands issued to the default stream by different host threads can run concurrently
  - Commands in the default stream may run concurrently with commands in non-default streams
  - Pass the option `–default-stream per-thread` to nvcc to enable per-thread default streams in CUDA 7+

GPU Pro Tip: CUDA 7 Streams Simplify Concurrency

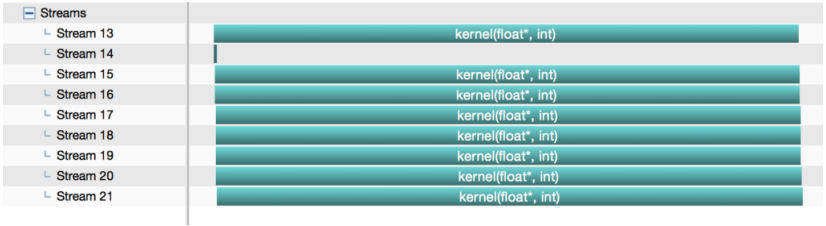
Multi-Stream Example: Legacy Behavior

```
for (int i = 0; i < num_streams; i++) {
  cudaStreamCreate(&streams[i]);
  cudaMalloc(&data[i], N * sizeof(float));
  // launch one worker kernel per stream
  kernel<<<1, 64, 0, streams[i]>>>(data[i], N);
  // launch a dummy kernel on the default stream
  kernel<<<1, 1>>>(0, 0);
}
```



Multi-Stream Example: Per-Thread Default Stream

```
1 for (int i = 0; i < num_streams; i++) {
2   cudaStreamCreate(&streams[i]);
3   cudaMalloc(&data[i], N * sizeof(float));
4   // launch one worker kernel per stream
5   kernel<<<1, 64, 0, streams[i]>>>(data[i], N);
6   // launch a dummy kernel on the default stream
7   kernel<<<1, 1>>>(0, 0);
8 }
```



Achievable Concurrency

Is it possible to launch two kernels that do independent tasks concurrently?

```
1  // host and device initialization
2  .......
3  // launch kernel1
4  myMethod1 <<<.... >>> (params);
5  // launch kernel2
6  myMethod2 <<<.....>>> (params);
```

- We can launch concurrent kernels in different streams
- There must be resources available while one kernel is running to run concurrent kernels
- The maximum concurrency is 16 kernels on Fermi, 32 on Kepler, and 128 on Turing and Ampere

Compute Capabilities: Features and Technical Specifications

Achievable Concurrency

Can we run two CUDA kernels from two different applications concurrently?

- A kernel from one CUDA context (analogue of host processes for the device) cannot execute concurrently with a kernel from another CUDA context
- The GPU may time slice to provide forward progress to each context
- Must enable Multi-Process Service (MPS) to run kernels from multiple process simultaneously

Performance Bottlenecks in CUDA

Differences between Host and Device

Host

- Limited amount of concurrent threads
- Context switches of threads are heavyweight
- Designed to minimize latency

Device

- Massive number of concurrently active threads
- Context switches are lightweight
  - Resources stay allocated to a thread till it completes
- Designed to maximize throughput

Key Ideas for Extracting Performance

- Desired application characteristics for device execution
  - (i) Large data-parallel computation
  - (ii) Complex computation kernel to justify the data movement costs
    - Keep data on the device to avoid repeated transfers
- Try and reduce resource consumption
- Exploit SIMT, reduce thread divergence in a warp
- Strive for good locality, use tiling to exploit shared memory
  - Improve throughput by reducing global memory traffic
  - Copy blocks of data from global memory to shared memory and operate on them (e.g., matrix multiplication kernel)
- Optimize memory accesses

  Global memory memory coalescing
  Shared memory avoid bank conflicts

What can we say about this code?

```
__global__ void dkernel(float *vector, int vectorsize) {
  int id = blockIdx.x * blockDim.x + threadIdx.x;
  switch (id) {
    case 0: vector[id] = 0; break;
    case 1: vector[id] = vector[id] * 10; break;
    case 2: vector[id] = vector[id - 2]; break;
    case 3: vector[id] = vector[id + 3]; break;
    ...
    case 31: vector[id] = vector[id] * 9; break;
  }
}
```

Dealing with Thread Divergence

- Thread divergence renders execution sequential because the SIMD hardware takes multiple passes through the divergent paths

```
if (threadIdx.x / WARP_SIZE > 2) {}
```

- Condition evaluating to different truth values is not bad
  - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path

```
if (threadIdx.x > 2) {}
```

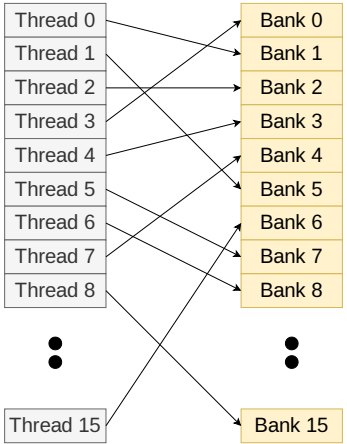
- Conditions evaluating to different truth-values for threads in a warp is bad
  - Creates two different control paths for threads in a block; branch granularity < warp size; threads 0 and 1 follow different path than the rest of the threads in the first warp

Parallel Memory Architecture

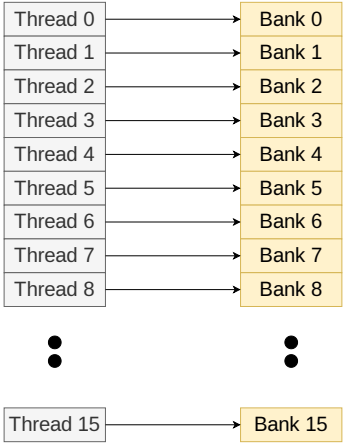
- In a parallel architecture, many threads access memory
- Memory is divided into **banks** to achieve high bandwidth
  - Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized

Example of Bank Addressing

Random access

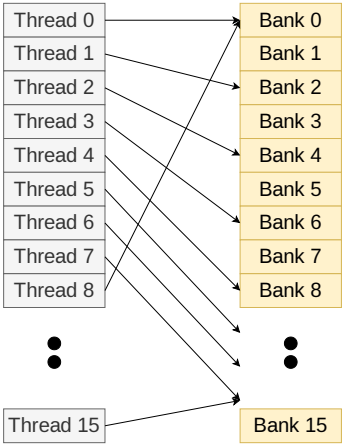


Linear access, stride=1

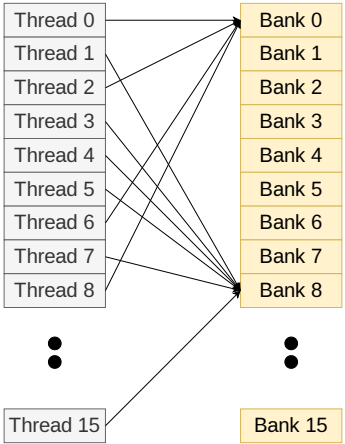


Example of Bank Addressing

Linear access, stride=2



Linear access, stride=8



Bank Conflicts in Shared Memory

Shared memory is as fast as registers if there are **no** bank conflicts

Fast case
- If all threads of a warp access different banks, there is no bank conflict
- If all threads of a warp access an identical address, there is no bank conflict (broadcast)

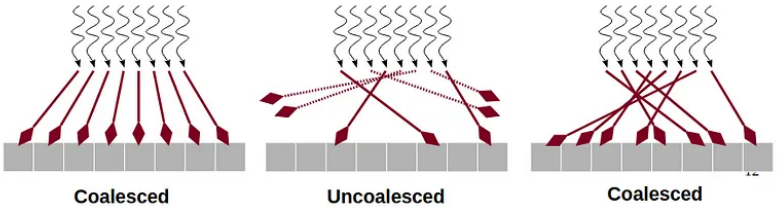
Slow case
- Bank Conflict: multiple threads in the same half-warp access the same bank
- Must serialize the accesses
- Cost = max # of simultaneous accesses to a single bank

Give low priority to fix low-degree bank conflicts since resolving it will increase instructions

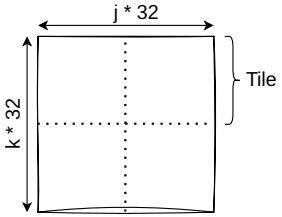
Using Shared Memory in CUDA C/C++

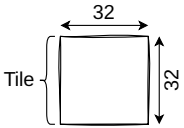
Memory Coalescing

- Issuing a memory instruction from a single warp can generate up to 32 data cache accesses
- Coalescing reduces the number of memory requests by merging accesses from multiple lanes into cache-line-sized chunks when there is spatial locality across the warp
  - Coalesced memory accesses imply a warp accesses adjacent data in a cache line
  - In the best case, this results in one memory transaction
- Uncoalesced memory accesses imply a warp accesses scattered data in different cache lines leading to memory divergence
  - This may result in 32 different memory transactions



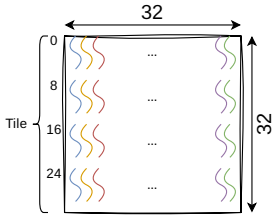
Copying a Matrix





TPB = 32 x 8

NB = j x k




```
1  __global__ void copy(float *odata, const float *idata) {
2    int x = blockIdx.x * TILE_DIM + threadIdx.x;
3    int y = blockIdx.y * TILE_DIM + threadIdx.y;
4    int width = gridDim.x * TILE_DIM;
5    for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
6      odata[(y+j)*width + x] = idata[(y+j)*width + x];
7  }
```

Matrix Transpose

```
1 __global__ void transposeNaive(float *odata, const float *idata) {
2   int x = blockIdx.x * TILE_DIM + threadIdx.x;
3   int y = blockIdx.y * TILE_DIM + threadIdx.y;
4   int width = gridDim.x * TILE_DIM;
5   for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
6     odata[x*width + (y+j)] = idata[(y+j)*width + x];
7 }
```

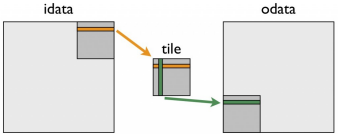
reads from idata are coalesced, but writes to odata have a stride of j*32

An Efficient Matrix Transpose in CUDA C/C++

Optimizing Matrix Transpose

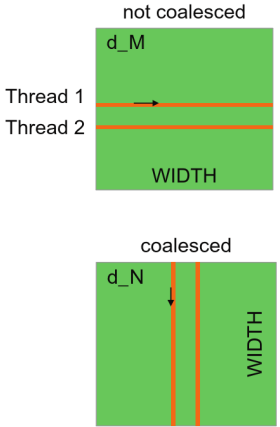
```
__global__ void transposeCoalesced(float *odata, const float *idata) {
  __shared__ float tile[TILE_DIM][TILE_DIM];
  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;
  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
    tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];
  __syncthreads();
  x = blockIdx.y * TILE_DIM + threadIdx.x; // transpose block offset
  y = blockIdx.x * TILE_DIM + threadIdx.y;
  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
    odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

Source file



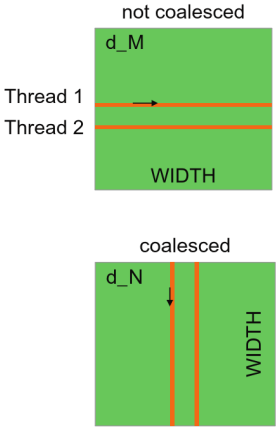
Matrix Multiplication Example

```
1  __global__ void matmulKernel(float* A, float* B, float*
       C) {
2    int row = blockIdx.y * blockDim.y + threadIdx.y;
3    int col = blockIdx.x * blockDim.x + threadIdx.x;
4    float tmp = 0;
5    if (row < N && col < N) {
6      // Each thread computes one element of the matrix
7      for (int k = 0; k < N; k++) {
8        tmp += A[row * N + k] * B[k * N + col];
9      }
10   }
11   C[row * N + col] = tmp;
12 }
```

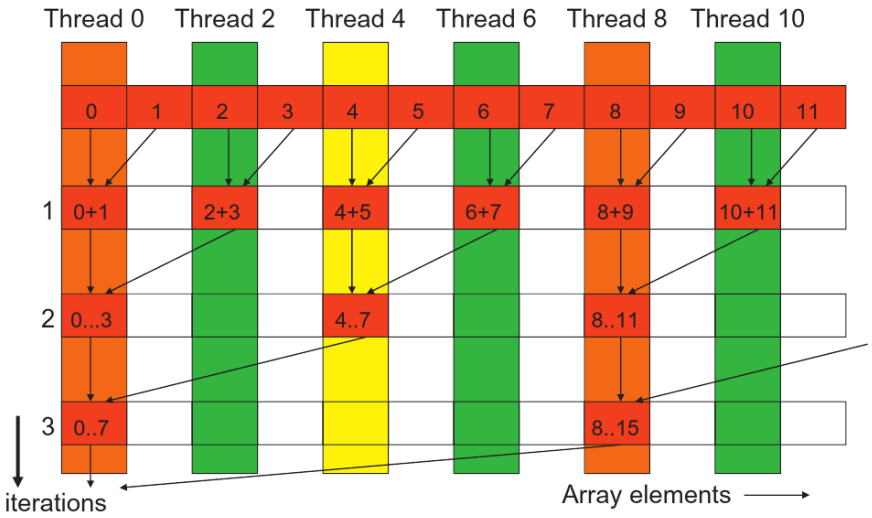


Optimizing Global Memory Accesses

- Try to ensure that memory requests from a warp can be coalesced
  - Using optimizations like tiling to make use of the faster shared memory
  - Stride-one access across threads in a warp is good
  - Use structure of arrays rather than array of structures



Implementing a Reduction Kernel in CUDA



Reduction Kernel

```
1  __shared__ float partialSum[];
2  partialSum[threadIdx.x] = X[blockIdx.x*blockDim.x+threadIdx.x];
3  __syncthreads();
4  unsigned int t = threadIdx.x;
5  for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
6    if (t % (2*stride) == 0)
7      partialSum[t] += partialSum[t+stride];
8    __syncthreads();
9  }
```

only even threads are active

Possible Optimizations on the Reduction Kernel

(i) basic implementation with modulo operator

(ii) strided access starting with each thread accessing two adjacent locations

(iii) strided access with reversed loop index

(iv) halve the number of threads

(v) unroll the last few loop iterations and avoid synchronization

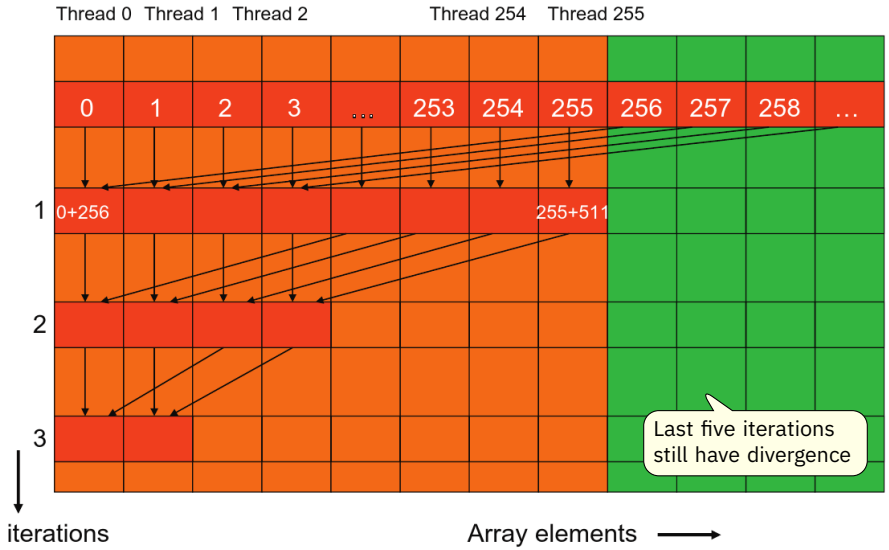
(vi) ...

M. Harris. Optimizing Parallel Reduction in CUDA.

Reduction Kernel

```
__shared__ float partialSum[];
partialSum[threadIdx.x] = X[blockIdx.x*blockDim.x+threadIdx.x];
__syncthreads();
unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x/2; stride >= 1; stride /= 2) {
  if (t < stride)
    partialSum[t] += partialSum[t+stride];
  __syncthreads();
}
```

Execution of the Revised Reduction Kernel



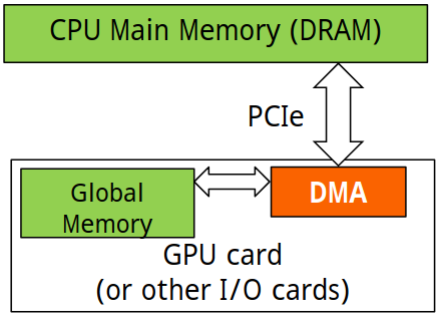
Avoid Divergence in the Last Few Iterations

```
1  for (unsigned int stride = blockDim.x/2; stride > 32; stride /= 2) {
2    if (t < stride)
3      partialSum[t] += partialSum[t+stride];
4    __syncthreads();
5  }
6  if (tid < 32) {
7    partialSum[tid] += partialSum[tid+32];
8    partialSum[tid] += partialSum[tid+16];
9    partialSum[tid] += partialSum[tid+8];
10   partialSum[tid] += partialSum[tid+4];
11   partialSum[tid] += partialSum[tid+2];
12   partialSum[tid] += partialSum[tid+1];
13 }
```

Efficient Data Management

Data Transfer Between CPU and GPU

- DMA is a hardware unit specialized to transfer bytes between physical memory address spaces
  - Uses system interconnect, typically PCIe in today's systems
- DMA (Direct Memory Access) hardware is used by `cudaMemcpy()`



Challenges with Virtual Memory

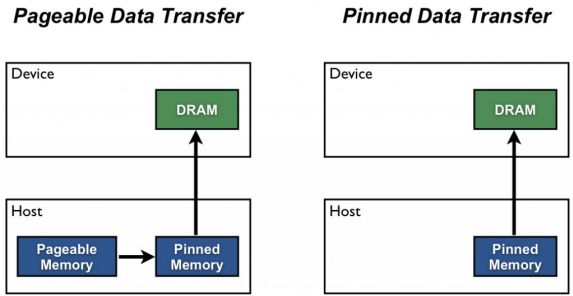
- Virtual memory support complicates data transfer
  - Pages in virtual address space are mapped into and out of the physical memory
  - The presence of a data (i.e., page) in the physical memory is checked during address translation
- `cudaMemcpy()` copies data as one or more DMA transfers
  - Address is translated, and page presence is checked for the entire source and destination regions at the beginning of each DMA transfer
- The OS could accidentally page-out the data that is being accessed by a DMA and page-in another virtual page into the same physical location

Pinned Memory

- Pinned memory are virtual memory pages that are specially marked so that they cannot be paged out
  - Also called Page Locked Memory or Locked Pages
- CPU memory that serves as the source or destination of a DMA transfer must be allocated as pinned memory
- If the source or destination of `cudaMemcpy()` in the host is not pinned, it needs to be first copied to a pinned memory leading to extra overhead
  - `cudaMemcpy()` is faster if the host memory source or destination is allocated in pinned memory since no extra copy is needed

Pinned Memory

- Allocate and free pinned memory with `cudaHostAlloc()` and `cudaFreeHost()` with the option `cudaHostAllocDefault`
- Pinned memory is a limited resource—over-subscription can have serious consequences



How to Optimize Data Transfers in CUDA C/C++

cudaMemCpy() with Pageable Memory

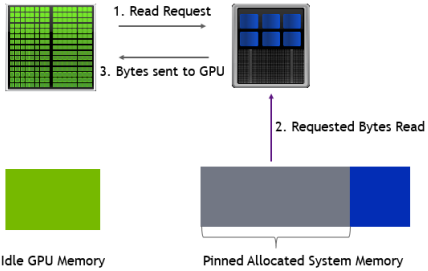
```
swarnendu@nilgiri:~/cuda-examples/src/vector$ nvprof ./vector-addition.out
==1941000== NVPROF is profiling process 1941000, command: ./vector-addition.out
Time taken (ms): h2d: 237.56 Kernel: 8.40618 d2h: 220.426
==1941000== Profiling application: ./vector-addition.out
==1941000== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   67.51%  474.62ms         2  237.31ms  237.16ms  237.46ms  [CUDA memcpy HtoD]
                   31.30%  220.01ms         1  220.01ms  220.01ms  220.01ms  [CUDA memcpy DtoH]
                    1.19%  8.3743ms         1  8.3743ms  8.3743ms  8.3743ms  vecAdd(float const *, float const *, float
*, unsigned int)
      API calls:   82.20%  695.45ms         3  231.82ms  220.42ms  237.64ms  cudaMemcpy
                   16.13%  136.47ms         3  45.490ms  111.40us  136.24ms  cudaMalloc
                    1.00%  8.4519ms         3  2.8173ms  2.8090us  8.3707ms  cudaEventSynchronize
                    0.64%  5.3911ms         3  1.7970ms  758.27us  2.5189ms  cudaFree
                    0.02%  142.30us       101  1.4080us     123ns  59.020us  cuDeviceGetAttribute
                    0.00%  39.372us         6  6.5620us  1.7480us  18.458us  cudaEventRecord
                    0.00%  32.013us         1  32.013us  32.013us  32.013us  cudaLaunchKernel
                    0.00%  22.907us         1  22.907us  22.907us  22.907us  cuDeviceGetName
                    0.00%  9.5030us         1  9.5030us  9.5030us  9.5030us  cuDeviceGetPCIBusId
                    0.00%  9.2820us         2  4.6410us     865ns  8.4170us  cudaEventCreate
                    0.00%  5.1920us         3  1.7300us     787ns  2.4430us  cudaEventElapsedTime
                    0.00%  1.5260us         3     508ns     180ns  1.1610us  cuDeviceGetCount
                    0.00%     601ns         1     601ns     601ns     601ns  cuDeviceTotalMem
                    0.00%     598ns         2     299ns     125ns     473ns  cuDeviceGet
                    0.00%     218ns         1     218ns     218ns     218ns  cuDeviceGetUuid
swarnendu@nilgiri:~/cuda-examples/src/vector$
```

cudaMemCpy() with Pinned Memory

```
swarnendu@nilgiri:~/cuda-examples/src/vector$ nvprof ./vector-addition-hostalloc-default.out
==1941051== NVPROF is profiling process 1941051, command: ./vector-addition-hostalloc-default.out
Time taken (ms): h2d: 87.2645 Kernel: 8.52237 d2h: 81.3356
==1941051== Profiling application: ./vector-addition-hostalloc-default.out
==1941051== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   66.03%  174.50ms         2  87.249ms  87.246ms  87.252ms  [CUDA memcpy HtoD]
                   30.77%  81.317ms         1  81.317ms  81.317ms  81.317ms  [CUDA memcpy DtoH]
                    3.20%  8.4589ms         1  8.4589ms  8.4589ms  8.4589ms  vecAdd(float const *, float const *, float
*, unsigned int)
      API calls:   68.93%  1.46052s         3  486.84ms  437.10ms  585.73ms  cudaHostAlloc
                   18.30%  387.85ms         3  129.28ms  125.50ms  136.48ms  cudaFreeHost
                   12.08%  255.92ms         3  85.306ms  81.333ms  87.328ms  cudaMemcpy
                    0.40%  8.4608ms         3  2.8203ms  3.1570us  8.4543ms  cudaEventSynchronize
                    0.25%  5.2468ms         3  1.7489ms  627.15us  2.5135ms  cudaFree
                    0.03%  555.08us         3  185.03us  106.57us  322.72us  cudaMalloc
                    0.01%  144.27us       101  1.4280us     166ns  57.635us  cuDeviceGetAttribute
                    0.00%  57.403us         1  57.403us  57.403us  57.403us  cudaLaunchKernel
                    0.00%  38.628us         6  6.4380us  1.7470us  26.804us  cudaEventRecord
                    0.00%  15.428us         2  7.7140us  1.1860us  14.242us  cudaEventCreate
                    0.00%  13.944us         1  13.944us  13.944us  13.944us  cuDeviceGetName
                    0.00%  10.629us         1  10.629us  10.629us  10.629us  cuDeviceGetPCIBusId
                    0.00%  4.1260us         3  1.3750us     769ns  2.4630us  cudaEventElapsedTime
                    0.00%  3.1800us         3  1.0600us     241ns  2.3700us  cuDeviceGetCount
                    0.00%     945ns         1     945ns     945ns     945ns  cuDeviceTotalMem
                    0.00%     812ns         2     406ns     156ns     656ns  cuDeviceGet
```

Zero-Copy Memory

- Zero copy memory is pinned memory that is mapped into the device address space
  - Both host and device have fine-grained direct access
  - Can leverage host memory when there is insufficient device memory
  - Avoids explicit data transfers between host and device
  - Should be used for occasional accesses when the data is read-only or the GPU memory is really scarce



Improving GPU Memory Oversubscription Performance

Zero-Copy Memory

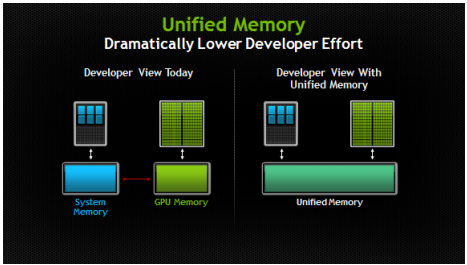
- Speed is limited by the interconnect (PCIe or NVLink) and it is not possible to take advantage of data locality
- Pinned memory does not imply zero-copy memory (GPU may not be able to access it)
- Unified virtual addressing (UVA) enables zero-copy memory
  - Provides a single virtual memory address space for all memory in the system, enables pointers to be accessed from GPU code

cudaMemCpy() with Zero-Copy Memory

```
swarnendu@nilgiri:~/cuda-examples/src/vector$ nvprof ./vector-addition-hostalloc-mapped.out
==1948543== NVPROF is profiling process 1948543, command: ./vector-addition-hostalloc-mapped.out
Invocation i: 0 Time taken (ms): kernel: 378.814
Invocation i: 1 Time taken (ms): kernel: 373.741
Invocation i: 2 Time taken (ms): kernel: 373.747
Invocation i: 3 Time taken (ms): kernel: 373.714
Invocation i: 4 Time taken (ms): kernel: 373.742
Invocation i: 5 Time taken (ms): kernel: 373.717
Invocation i: 6 Time taken (ms): kernel: 376.482
Invocation i: 7 Time taken (ms): kernel: 373.599
Invocation i: 8 Time taken (ms): kernel: 376.099
Invocation i: 9 Time taken (ms): kernel: 376.226
==1948543== Profiling application: ./vector-addition-hostalloc-mapped.out
==1948543== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  3.74542s        10  374.54ms  373.60ms  376.60ms  vecAdd(float const *, float const *, float
*, unsigned int)
      API calls:   57.77%  3.76791s        10  376.79ms  375.56ms  378.79ms  cudaEventSynchronize
                   30.61%  1.99639s         3  665.46ms  583.01ms  742.25ms  cudaHostAlloc
                    6.53%  425.82ms         3  141.94ms  141.12ms  143.34ms  cudaFreeHost
                    5.07%  330.84ms         1  330.84ms  330.84ms  330.84ms  cudaSetDeviceFlags
                    0.01%  447.18us        10  44.718us  27.583us  94.893us  cudaLaunchKernel
                    0.00%  207.81us        20  10.390us  2.7920us  30.620us  cudaEventRecord
                    0.00%  139.87us       101  1.3840us     140ns  55.603us  cuDeviceGetAttribute
                    0.00%  59.182us        10  5.9180us  3.9300us  14.372us  cudaEventElapsedTime
                    0.00%  41.062us         2  20.531us     940ns  40.122us  cudaEventCreate
```

Managed Memory

- Unified Virtual Memory (UVM) provides a single memory space accessible by all GPUs and CPUs in the system
- Use `cudaMallocManaged()` to allocate data in unified memory or use `__managed__` keyword in the global scope
  - Returns a pointer that can be accessed from both host and device code



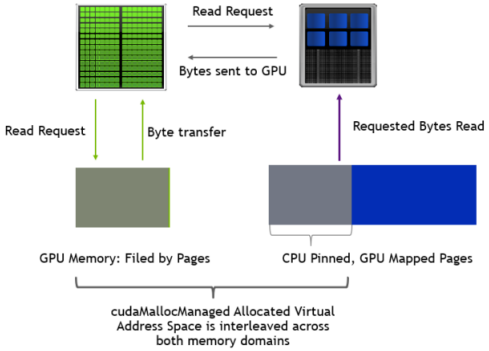
Unified Memory in CUDA 6

An Even Easier Introduction to CUDA

Unified Memory for CUDA Beginners

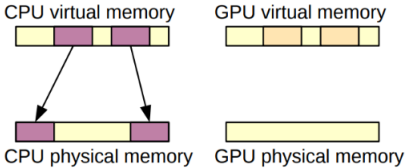
Managed Memory

- UVM features have been evolving starting from CUDA 6+
  - 6.x: use a single pointer in both CPU functions and GPU kernels
  - 8.x: added 49-bit virtual addressing and on-demand page migration
- CUDA runtime **automatically migrates data** allocated in Unified Memory between host and device, different from UVA

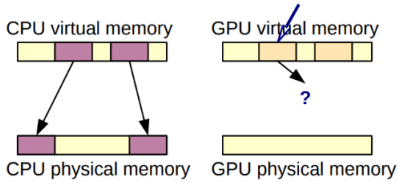


Improving GPU Memory Oversubscription Performance

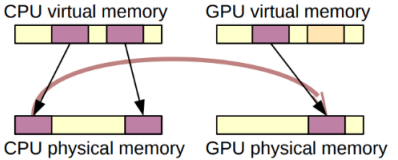
On-demand Paging



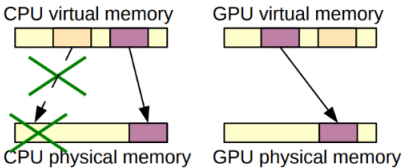
1 Data allocated in CPU memory



2 GPU touches unallocated page, triggers page fault



3 Page fault handler allocates page in GPU mem, copies contents



4 If GPU modifies page contents, invalidate CPU copy. Next CPU access will cause data to be copied back from GPU mem.

cudaMemcpy() with Managed Memory

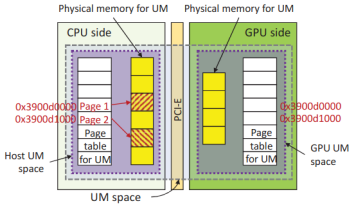
```
swarnendu@nilgiri:~/cuda-examples/src/vector$ nvprof ./vector-addition-managed.out
==1948976== NVPROF is profiling process 1948976, command: ./vector-addition-managed.out
Invocation: 0 Time taken (ms): kernel: 1219.01
==1948976== Profiling application: ./vector-addition-managed.out
==1948976== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  1.21679s         1  1.21679s  1.21679s  1.21679s  vecAdd(float const *, float const *, float*, int)
      API calls:   73.02%  1.21899s         1  1.21899s  1.21899s  1.21899s  cudaEventSynchronize
                   17.84%  297.79ms         3  99.265ms  23.511us  297.70ms  cudaMallocManaged
                    9.12%  152.21ms         3  50.737ms  50.422ms  50.941ms  cudaFree
                    0.01%  142.95us       101  1.4150us     145ns  55.431us  cuDeviceGetAttribute
                    0.00%  75.898us         1  75.898us  75.898us  75.898us  cudaLaunchKernel
                    0.00%  59.680us         2  29.840us  5.5550us  54.125us  cudaEventCreate
                    0.00%  46.808us         2  23.404us  3.9890us  42.819us  cudaEventRecord
                    0.00%  30.918us         1  30.918us  30.918us  30.918us  cuDeviceGetName
                    0.00%  9.5370us         1  9.5370us  9.5370us  9.5370us  cuDeviceTotalMem
                    0.00%  9.3320us         1  9.3320us  9.3320us  9.3320us  cuDeviceGetPCIBusId
                    0.00%  7.1650us         1  7.1650us  7.1650us  7.1650us  cudaEventElapsedTime
                    0.00%  2.6680us         3     889ns     183ns  1.3890us  cuDeviceGetCount
                    0.00%     883ns         2     441ns     367ns     516ns  cuDeviceGet
                    0.00%     350ns         1     350ns     350ns     350ns  cuDeviceGetUuid

==1948976== Unified Memory profiling result:
Device "Quadro RTX 5000 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
   36250  54.919KB  4.0000KB  0.9961MB  1.898617GB  224.7973ms  Host To Device
   18408  170.67KB  4.0000KB  0.9961MB  2.996094GB  261.0419ms  Device To Host
    4693         -         -         -           -  539.8935ms  Gpu page fault groups
Total CPU Page faults: 15349
```

cudaMemcpy() with Managed Memory

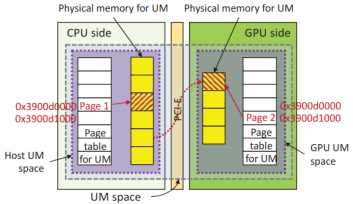
```
~/cuda-examples/src/vector$ nvprof --print-gpu-trace ./vector-addition-managed.out 2>&1 | tee output.txt
...
==1949017== Profiling result:
   Start  Duration            Grid Size      Block Size     Regs*    SSMem*    DSMem*           Device   Context
Stream        Unified Memory  Virtual Address  Name
499.24ms         -                    -               -         -         -         -                -
-         -         PC 0xe0f0cdc8   0x7fd774000000  [Unified Memory CPU page faults]
499.73ms         -                    -               -         -         -         -                -
-         -         PC 0xe0f0cdc8   0x7fd774010000  [Unified Memory CPU page faults]
499.76ms         -                    -               -         -         -         -                -
-         -         PC 0xe0f0cdc8   0x7fd774020000  [Unified Memory CPU page faults]
...
1.59851s         -                    -               -         -         -         -                -
-         -         PC 0xe0f0cdf0   0x7fd76fe80000  [Unified Memory CPU page faults]
1.59862s         -                    -               -         -         -         -                -
-         -         PC 0xe0f0cdf0   0x7fd76ff00000  [Unified Memory CPU page faults]
1.60122s  1.22684s        (1048576 1 1)       (256 1 1)        16        0B        0B  Quadro RTX 5000
1         7                     -                 -  vecAdd(float const *, float const *, float*, int) [117]
1.60123s  334.53us                -               -         -         -         -  Quadro RTX 5000
-         -                     8   0x7fd730000000  [Unified Memory GPU page faults]
1.60142s  10.016us                -               -         -         -         -  Quadro RTX 5000
-         -           96.000000KB   0x7fd730000000  [Unified Memory Memcpy HtoD]
1.60143s  4.6720us                -               -         -         -         -  Quadro RTX 5000
-         -           32.000000KB   0x7fd730018000  [Unified Memory Memcpy HtoD]
...
```

Unified Virtual Memory (UVM)

- On pre-Pascal GPUs, `cudaMallocManaged()` allocated managed space on the device that was active at the time of the call
- Pascal onward, `cudaMallocManaged()` does not immediately allocate physical memory
  - Pages and page table entries are not created until the first access by the GPU or the CPU
- Hardware supports page faulting and migration
  - The GPU stalls the accessor threads when they access absent pages
  - The Page Migration Engine migrates pages to the device before resuming the threads



(a) After the host has accessed page 1 and page 2.



(b) After the GPU has accessed page 2.

J. Jung et al. Overlapping Host-to-Device Copy and Computation using Hidden Unified Memory. PPoPP'20.

Pre-Pascal Behavior of cudaMallocManaged()

```
__global__ void add(int n, float* x, float* y) {
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
  }
int main(void) {
  int N = (1 << 20);
  float *x, *y;
  cudaMallocManaged(&x, N * sizeof(float));
  cudaMallocManaged(&y, N * sizeof(float));
  for (int i = 0; i < N; i++)
    x[i] = 1.0f; y[i] = 2.0f;
  int blockSize = 256;
  int numBlocks = (N + blockSize - 1) / blockSize;
  add<<<numBlocks, blockSize>>>(N, x, y);
  cudaDeviceSynchronize();
  ...
}
```

x and y are allocated on GPU memory, driver sets up page table entries

Page fault on CPU, x and y are copied to CPU memory

Lacked support for page faults, so all data is copied to GPU before kernel launch

Moves data back to CPU memory

Unified Memory for CUDA Beginners

Pre-Pascal Behavior of cudaMallocManaged()

```
__global__ void add(int n, float* x, float* y) {
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
  }
int main(void) {
  int N = (1 << 20)
  float *x, *y;
  cudaMallocManaged
  cudaMallocManaged
  for (int i = 0; i
    x[i] = 1.0f; y[
  int blockSize = 2
  int numBlocks = (N + blockSize - 1) / blockSize;
  add<<<numBlocks, blockSize>>>(N, x, y);
  cudaDeviceSynchronize();
  ...
}
```

- No concurrent access
- No on-demand migration to GPU
- No oversubscription

Moves data back to CPU memory

Lacked support for page faults, so all data is copied to GPU before kernel launch

Unified Memory for CUDA Beginners

Behavior of cudaMallocManaged() for Pascal+

```
__global__ void add(int n, float* x, float* y) {
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
  }
int main(void) {
  int N = (1 << 20);
  float *x, *y;
  cudaMallocManaged(&x, N * sizeof(float));
  cudaMallocManaged(&y, N * sizeof(float));
  for (int i = 0; i < N; i++)
    x[i] = 1.0f; y[i] = 2.0f;
  int blockSize = 256;
  int numBlocks = (N + blockSize - 1) / blockSize;
  add<<<numBlocks, blockSize>>>(N, x, y);
  cudaDeviceSynchronize();
  ...
}
```

Page migration cost is now part of the kernel execution time

Physical memory for x and y are not immediately allocated, allocated only on first access or prefetch

Page fault on CPU, x and y are allocated on CPU memory

Supports page faults, so no data migration overhead before kernel launch

Unified Memory for CUDA Beginners

Profiling with nvprof on Turing GPU

Page Migration: High-Level Mechanism

- Assume a Pascal+ GPU accesses a page is not present in the local GPU memory
- Address translation for the faulting page generates a fault message and locks the TLBs for the corresponding SM
  - On some architectures, two SMs share a TLB, so both are locked
  - Locking implies outstanding translations can proceed but new translations will be stalled until all faults are resolved
- GPU can generate many faults concurrently for the same page
- UVM driver processes the faults, remove duplicates, updates mappings and transfers the data
- Fault handling adds significant overhead to streaming performance of UVM

Maximizing Unified Memory Performance in CUDA

Ways to Reduce Page Migration Overhead

Initialize data on the device

```
__global__ void init(int n, float *x, float *y) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
}
```

Unified Memory for CUDA Beginners

Ways to Reduce Page Migration Overhead

- Use warm-up iterations or take the average of kernel multiple runs
- Prefetch data on the GPU with `cudaMemPrefetchAsync()`

```
int device = -1;
cudaGetDevice(&device);
cudaMemPrefetchAsync(x, N*sizeof(float), device, NULL);
cudaMemPrefetchAsync(y, N*sizeof(float), device, NULL);
```

Explicit Memory Hints in UVM

- Advise runtime on expected memory access behaviors
  - cudaMemAdvise(ptr, count, hint, device)
- Possible hints:

cudaMemAdviseSetReadMostly	Specify read duplication
cudaMemAdviseSetPreferredLocation	Suggest best location
cudaMemAdviseSetAccessedBy	Suggest mapping

- Hints do not trigger data movement by themselves

Explicit Memory Hints in UVM

cudaMemAdviseSetReadMostly

- Data will usually be read-only
- UM system will make a “local” copy of the data for each processor that touches it
- If a processor writes to it, this invalidates all copies except the one written

cudaMemAdviseSetPreferredLocation

- Suggests which processor is the best location for the data
- Data will be migrated to the preferred processor on-demand (or if prefetched)
- If possible, data mappings will be provided when other processors touch it
- If mapping is not possible, data is migrated

Explicit Memory Hints in UVM

cudaMemAdviseSetAccessedBy

- Does not cause movement or affect location of data
- Indicated processor receives a mapping to the data
- If the data is migrated, mapping is updated
- Objective: provide access without incurring page faults

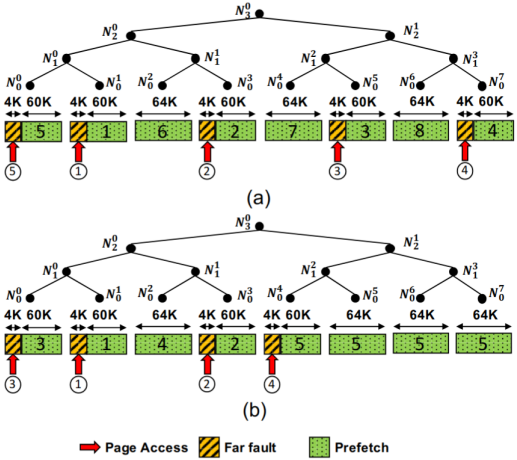
Hardware Prefetching

- Providing user hints can be complicated and error-prone
- Hardware can implement different prefetch policies: (i) random, (ii) sequential, or (iii) locality-aware
  - For example, “random” prefetches a random 4KB page from the 2MB large page boundary along with the 4KB faulting page
- Nvidia implements a locality-aware tree-based neighborhood prefetcher GeForce GTX 1080ti
  - Migrate multiples of 64KB basic blocks contiguous in the virtual address space grouped in a single transfer
  - All pages being prefetched are local to the current faulty pages and are within 2MB large page boundary

D. Ganguly et al. Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory. ISCA’19.

Tree-based Neighborhood Prefetcher

- Allocation with cudaMallocManaged() is logically divided into 2MB large pages
- The 2MB pages are further divided into logical 64KB basic blocks to create a full binary tree
- If the user-specified allocation request is not a multiple of 2MB, the remainder allocation is rounded up to the next $2i * 64$ KB
  - If the requests are for 4MB and 192KB, then two 2MB trees and 1 256KB trees are created



D. Ganguly et al. Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory. ISCA'19.

Note about UVM

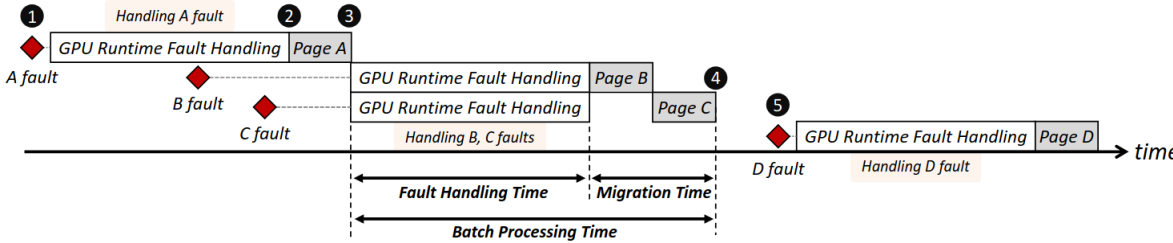
- UVM provides a coherent view of a single virtual address space between CPUs and GPUs with automatic data migration via demand paging
  - + Allows GPUs to access a page that resides in the CPU memory as if it were in the GPU memory
  - + Allows applications to run without worrying about the device memory capacity

```
struct dataElem {
  int prop1;
  int prop2;
  char *name;
}
```

- Primary goal for UVM is to improve programmer productivity
  - + Code is less verbose, makes it easy to work with nested data structures (think of C++ classes with dynamically allocated attributes)
- UVM kernels may have poorer performance
  - – Negative is the substantial cost of address translation overhead and demand paging

Handling GPU Page Faults

- When a GPU tries to access a physical memory page that is not currently resident in device memory, a page fault is raised and GPU runtime migrates the requested page to the GPU memory
- Page fault handling is expensive because it requires long latency communications between the CPU and GPU over the PCIe bus
  - The GPU runtime processes a group of page faults together to amortize overhead



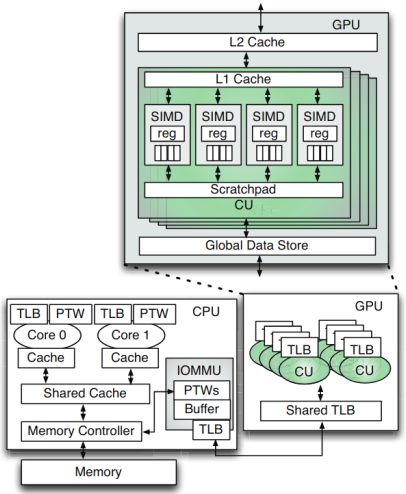
H. Kim et al. Batch-Aware Unified Memory Management in GPUs for Irregular Workloads. ASPLOS'20

Memory Access Divergence

- An SIMD memory instruction cannot complete until data for all threads are available
  - Problematic for irregular applications with little scope for coalescing
  - Execution of one instruction requires multiple cache accesses when accesses fall on distinct cache lines and multiple virtual-to-physical address translations when accesses fall on distinct pages
- Negative impact from divergence can impact address translation more than cache access
  - Irregular memory accesses can lead from 1 to warp size (32/64) address translation requests, most will miss in the TLB
  - A page table walk on a TLB miss can take up to four memory accesses, for a total of 128–256 memory accesses per instruction
  - GPUs employ physical caches which makes address translation the bottleneck

Address Translation Request

- IOMMU is a hardware component on the CPU that services address translation requests for accesses to the DRAM by any accelerator (e.g., GPU)
  - IOMMU supports multiple page table walkers (e.g., 8–16) to concurrently service multiple page table walk requests (TLB misses)
  - IOMMU employs small page walk caches for the first three levels of the page tables
- GPU multiprocessors (compute units) share a private L1 TLB across SIMD units
- The GPU's L1 TLBs are backed by a larger L2 TLB that is shared across all the CUs in the GPU



S. Shin et al. Scheduling Page Table Walks for Irregular GPU Applications. ISCA'18.

Address Translation Request

(i) An address translation request is generated by a SIMD load/store instruction

(ii) A hardware coalescer merges multiple requests to the same page generated by the same SIMD instruction

(iii) The coalesced translation request looks up the GPU's L1 TLB and then the GPU's shared L2

(iv) On a miss in the GPU's L2 TLB, the request is sent to the IOMMU

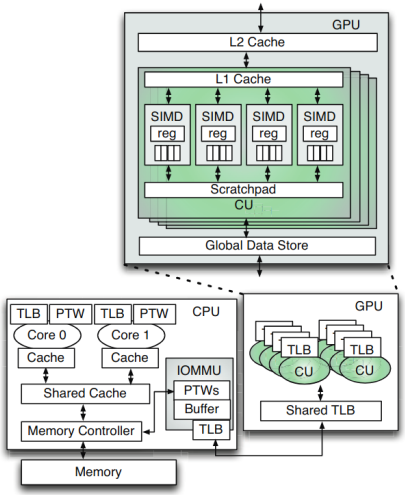
(v) At the IOMMU, the request looks up the IOMMU's TLBs

(vi) On a miss, the request queues up as a page walk request in the IOMMU buffer

(vii) When an IOMMU's page table walker becomes free, it selects a pending request from the IOMMU buffer in some order

(viii) The page table walker first performs a PWC lookup and then completes the walk of the page table, generating one to four memory accesses

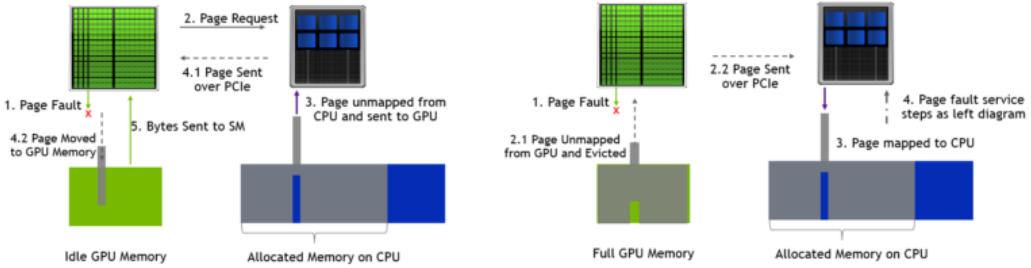
(ix) On finishing a walk, the desired translation is returned to the TLBs and ultimately to the GPU SIMD unit that requested it



S. Shin et al. Scheduling Page Table Walks for Irregular GPU Applications. ISCA'18.

Memory Oversubscription

- UVM support allows GPUs to oversubscribe memory
- With oversubscription, a memory page is first evicted from GPU memory to system memory, followed by transfer of requested memory from CPU to GPU



Improving GPU Memory Oversubscription Performance

Prefetching with Memory Oversubscription

- Aggressive prefetching under memory constraint can be counter-productive, may cause displacement of heavily-accessed pages
- CUDA drivers implement LRU 4KB page replacement policy
- Penalty of faults are greater under oversubscription
  - Threads need to be stalled for writing back pages along with the latency to migrate new pages
- An option is to disable prefetching on oversubscription

References

- D. Kirk and W. M. Hwu. Programming Massively Parallel Processors. Chapters 1–5, 13, 20, 3rd edition, Morgan Kaufmann.
- T. Aamodt et al. General-Purpose Graphics Processor Architectures. Chapters 1–4, Springer Cham.
- D. Patterson and J. Hennessy. Computer Organization and Design. Appendix C, 5th edition, Morgan Kaufmann.
- J. Hennessy and D. Patterson. Computer Architecture: A Quantitative Approach. Section 4.4, 6th edition, Morgan Kaufmann.
- NVIDIA Corporation. CUDA C++ Programming Guide.
- NVIDIA Corporation. CUDA C++ Best Practices Guide.
- NVIDIA CUDA Compiler Driver NVCC.