CS 610: GPU Architecture and CUDA Programming

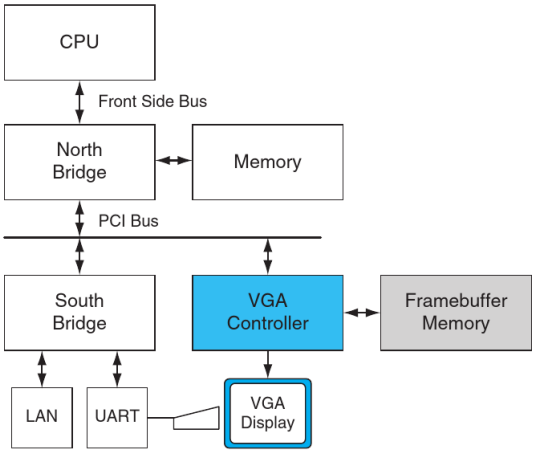
Swarnendu Biswas

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

Sem 2024-25-I

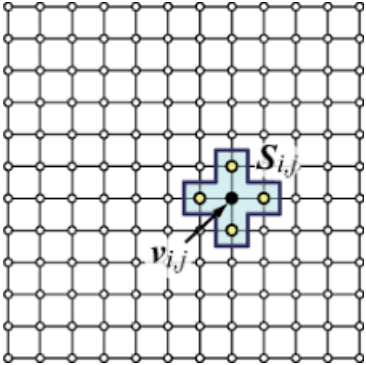
Rise of GPU Computing

- 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

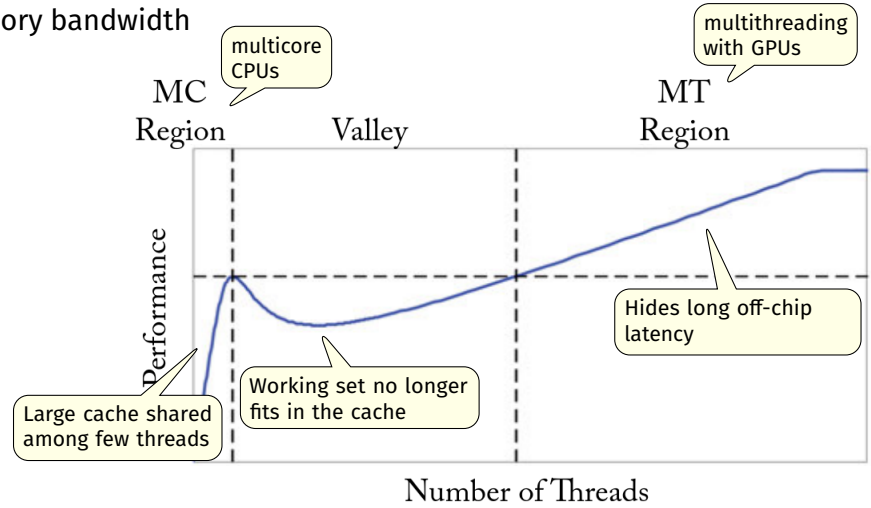
- NVIDIA released GeForce 8800 GTX in 2006 with CUDA architecture
  - + General-purpose ALU and instruction set for general-purpose computation
  - + Allowed arbitrary reads and writes to shared memory
  - + IEEE compliance for single-precision floating-point arithmetic
- Introduced CUDA C and the toolchain 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

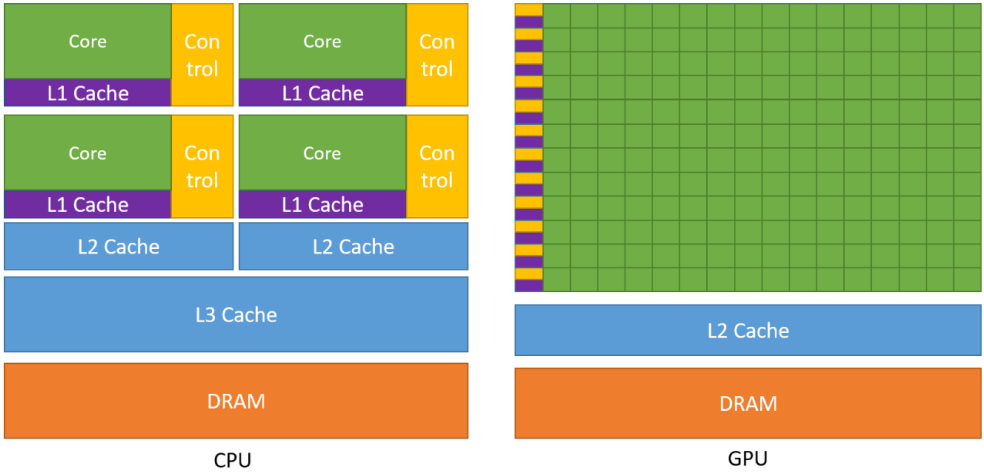


Key Insights in GPU Architecture

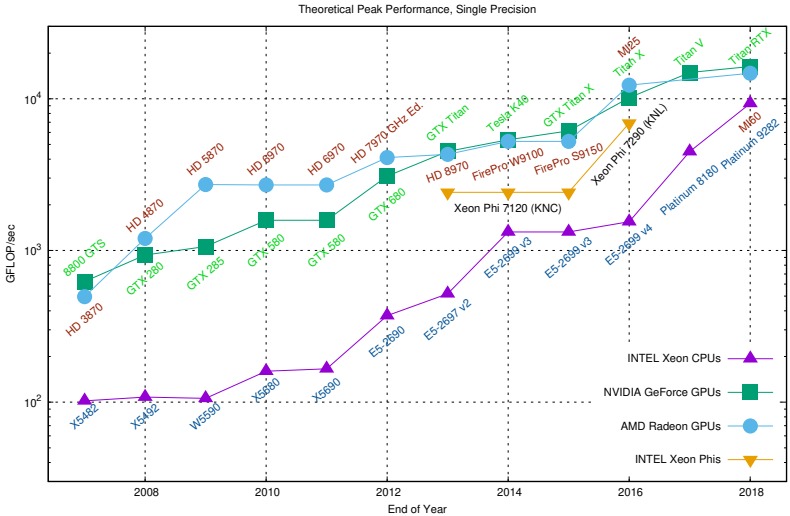
- GPUs are suited for **compute-intensive data-parallel** applications
  - The same program is executed for each data element
  - Less complex control flow
- Many-core chip
  - SIMD execution within a single core (many ALUs performing the same instruction)
  - Multi-threaded execution on a single core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

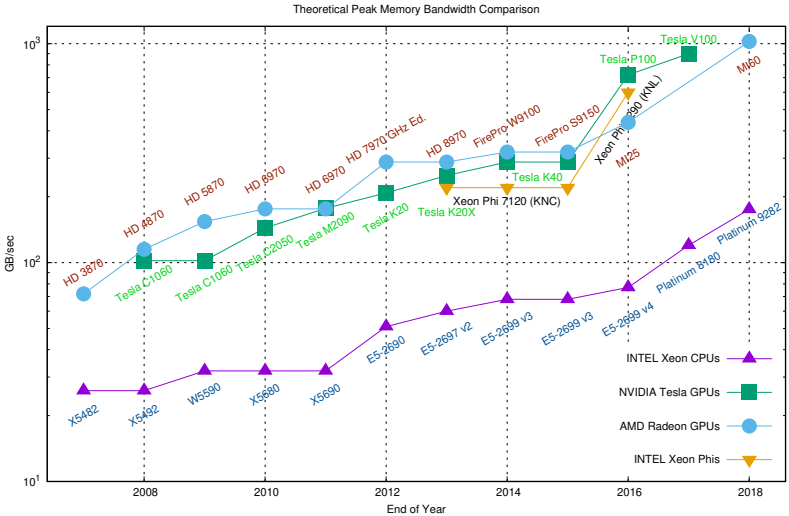


Comparing FLOPS



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

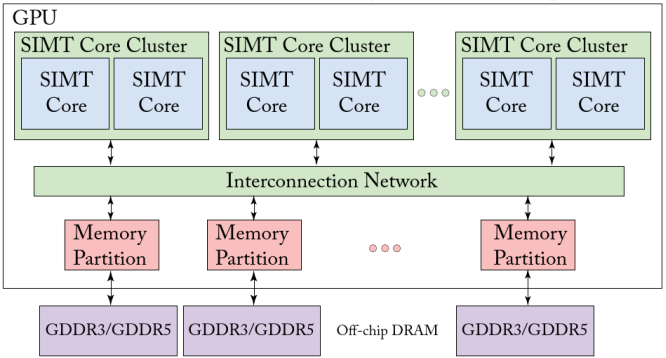
- Designed for running a few potentially complex tasks
  - Tasks may be unconnected
  - Suitable to run system software like the OS and applications
- Small number of registers per core private to a task
  - Context switch between tasks is expensive in terms of time
  - Register set must be saved to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

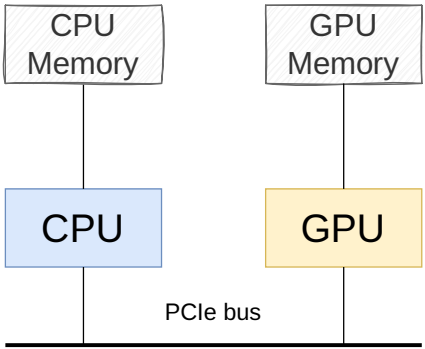
Compute Capability (CC)

- 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 16KB local memory per thread, and 2.x and 3.x devices have 512KB local memory per thread

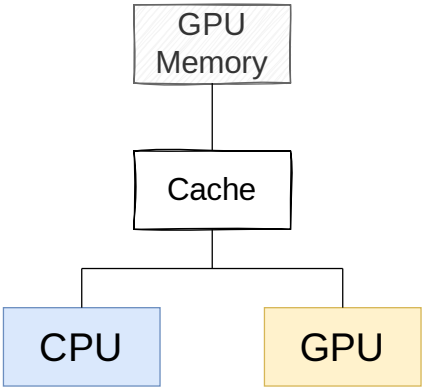
CUDA: Version features and specifications

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.cpp
$ ./a.out

$
```

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.cpp
$ ./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

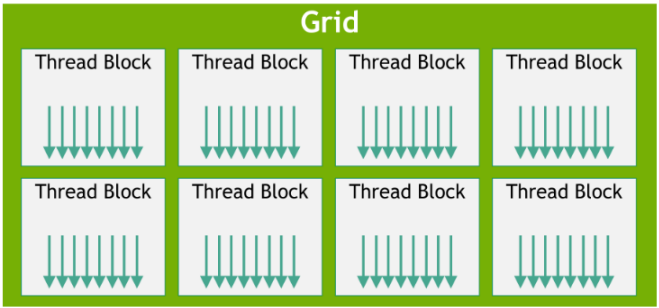
```
KernelName<<<m, n>>>(arg1, arg2, ...)
```

- GPU spawns `m` blocks with `n` threads that run a copy of the same function
  - CPU can continue processing while GPU runs 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 barrier synchronization and shared memory
- A grid is an array of thread blocks that execute the same kernel



Dimension and Index Variables

Dimension

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

Index

Type is `dim3`

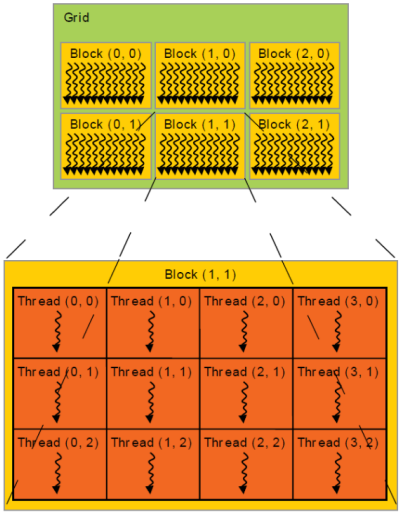
- `blockIdx` gives the index of the block in the grid
- `threadIdx` gives the index of the thread within the block

Thread Hierarchy

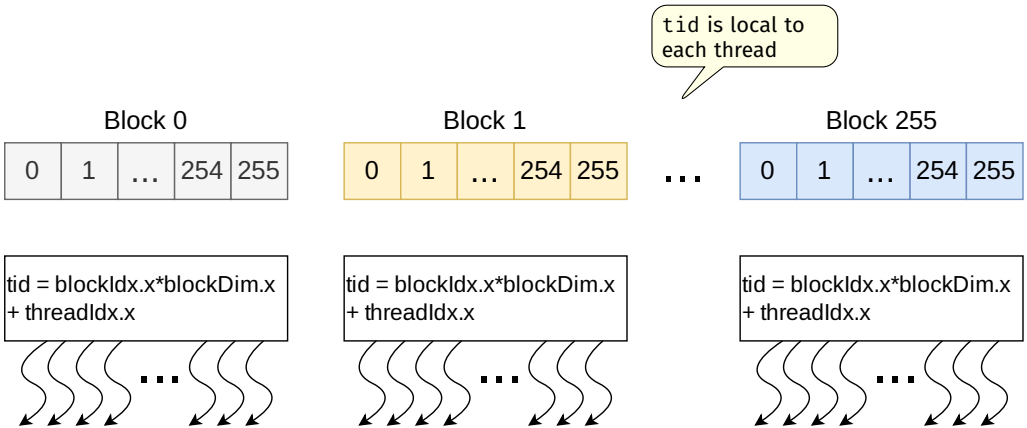
- `threadIdx` is a 3-component vector
  - Thread index can be 1D, 2D, or 3D
  - Thread blocks can be 1D, 2D, or 3D
- How to find out the relation between thread IDs and `threadIdx`?
  - 1D: `tid = threadIdx.x`
  - 2D block of size `(Dx, Dy)`: thread ID of a thread of index `(x, y)` is `(x + yDx)`
  - 3D block of size `(Dx, Dy, Dz)`: thread ID of a thread of index `(x, y, z)` is `(x + yDx + zDxDy)`

Thread 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



Device Management

Query device information 

Compile with

- 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 much greater number of threads
- No fixed rule, needs exploration and experimentation
- Choose number of threads in a block to be some 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);
```

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)`

Dynamic Parallelism

- 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 compute capability 3.5 and CUDA 5.0 or higher

Warp Scheduling

Mapping Blocks and Threads

- A GPU executes one or more kernel grids
- When a CUDA kernel is launched, the thread blocks are distributed to SMs
  - There is a limit on the number of blocks that can be assigned to each SM
  - For example, a CUDA device may allow up to eight blocks to be assigned to each SM
  - Multiple thread blocks can execute concurrently on one SM
- The threads of a block execute concurrently on one SM
  - CUDA cores in the SM execute threads
- 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 multiprocessors
- Blocks are mostly **not** supposed to synchronize with each other
  - Allows for simple hardware support for data parallelism

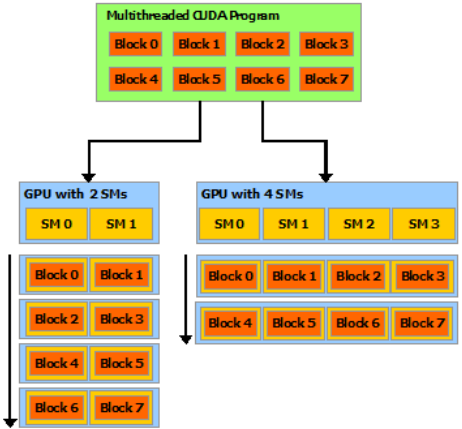
CUDA runtime can execute blocks in any order

Scheduling Blocks

- Number of threads that can be simultaneously tracked and scheduled is bounded
  - Requires resources for an SM to maintain block and thread indices and their execution status
- 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

Block Scalability

- A GPU kernel is partitioned into thread blocks that execute independently from each other
- Hardware can assign blocks to SMs in any order
  - A kernel with enough blocks scales across GPUs
  - Not all blocks may be resident at the same time
- A GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors

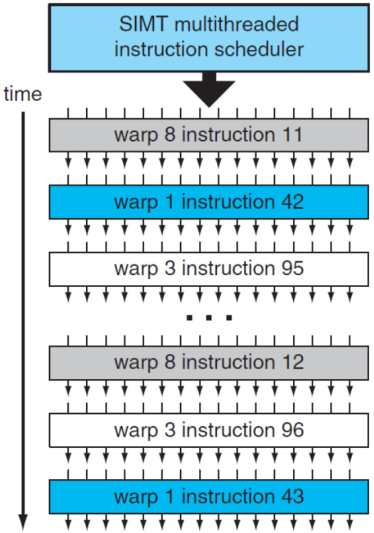


Thread Warps

- Conceptually, threads in a block can execute in any order
- Sharing a control unit among compute units reduce hardware complexity, cost, and power consumption
- A set of consecutive threads (currently 32) that execute in SIMD fashion is called a warp
  - These are called wavefront (with 64 threads) on AMD
- Warps are scheduling units in an SM
  - Part of the implementation in NVIDIA, not the programming model

Thread Warps

- 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 if branch and other threads take the else branch, they cannot operate in lockstep
  - Some threads must wait for the others to execute, renders code at that point to be serial rather than parallel
- The programming model does not prevent thread divergence
- Divergence occurs only within a warp, so it is a performance problem at the warp level

Example of Thread Divergence

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

- If the memory system were “ideal” and responded to memory requests within some fixed latency it would, in theory, be possible to design the core to support enough warps to hide this latency using fine-grained multithreading. In this case it can be argued that we can reduce the area of the chip for a given throughput by scheduling warps in “round robin” order. In round robin the warps are given some fixed ordering, for example ordered by increasing thread identifiers, and warps are selected by the scheduler in this order. One property of this scheduling order is that it allows roughly equal time to each issued instruction to complete execution. 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. So, increasing the number of warps up to this point can in principle increase throughput per core. However, there is an important trade off: to enable a different warp to issue an instruction each cycle it is necessary that each thread have its own registers (this avoids the need to copy and restore register state between registers and memory). Thus, increasing the number of warps per core increases the fraction of chip area devoted to register file

Warp Scheduling

- Section 3.1.3

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

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

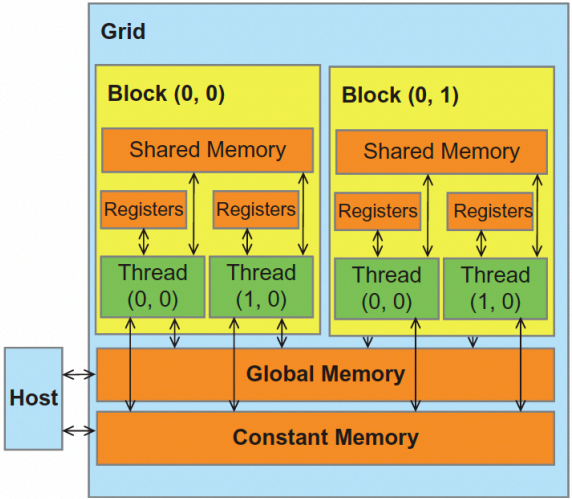
Memory Hierarchy in CUDA

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



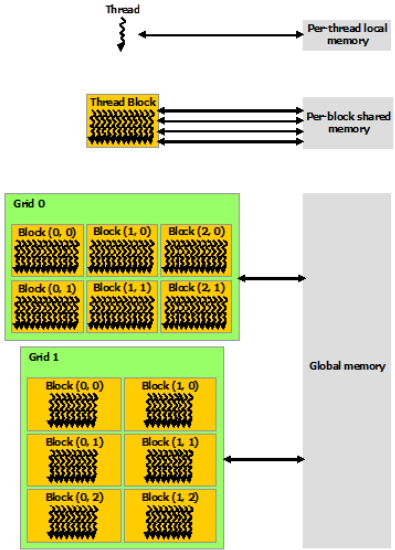
Variable Type Qualifiers in CUDA

	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__`
- Automatic variables without any qualifier reside in a register
  - 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 per SM
  - So 256KB 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 per SM
  - So 256KB 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 aims to bridge the gap in memory speed and access
  - Also called scratchpad memory
  - Usually 16–64KB of storage that can be accessed efficiently by all threads in a block
- Primary mechanism in CUDA for efficiently supporting thread cooperation
- Each SM contains a single shared memory structure
  - Resides adjacent to an SM, on-chip
  - The space is shared among all blocks running on that SM

Shared Memory

- Variable in shared memory is allocated using the `__shared__` specifier
  - Faster than global memory
  - Can be accessed only by threads within a block
- 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
  - Constant memory is 64KB
- Constant memory is 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

Constant Variables

- Constant variables cannot be modified by kernels
  - Reside in constant memory
  - Accessible from all threads within a grid
- They are defined with global scope within the kernel using the prefix `__constant__`
- Host code can access via `cudaMemcpyToSymbol()` and `cudaMemcpyFromSymbol()`

Local Memory

- Local memory is off-chip memory
  - More like thread-local global memory, so it requires memory transactions and consumes bandwidth
- Automatic variables are placed in local memory
  - Arrays for which it is not known whether indices are constant quantities
  - Large structures or arrays that consume too much register space
  - 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

Matrix Multiplication Example

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

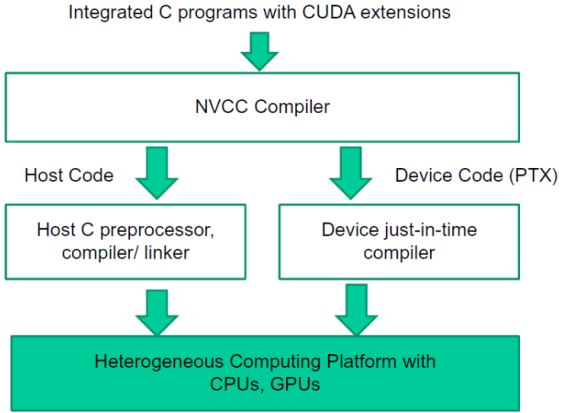
- Data movement must be orchestrated by programmer
  - Association between threads and storage is exposed to programming model
- L1 cache is not coherent, L2 cache is coherent

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

Description of Selected NVCC Options

Options	Description
`-std {c++03|c++11|c++14}`	Select a particular C++ dialect
`-m {32|64}`	Specify the architecture
`-arch ARCH`	Specify the class of the virtual GPU architecture
`-code CODE`	Specify the name of the GPU 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

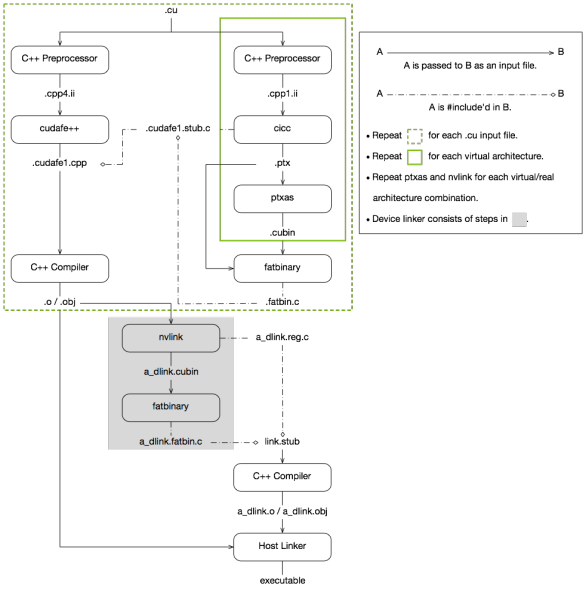
(iii) Input program is processed for compilation of the host code

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

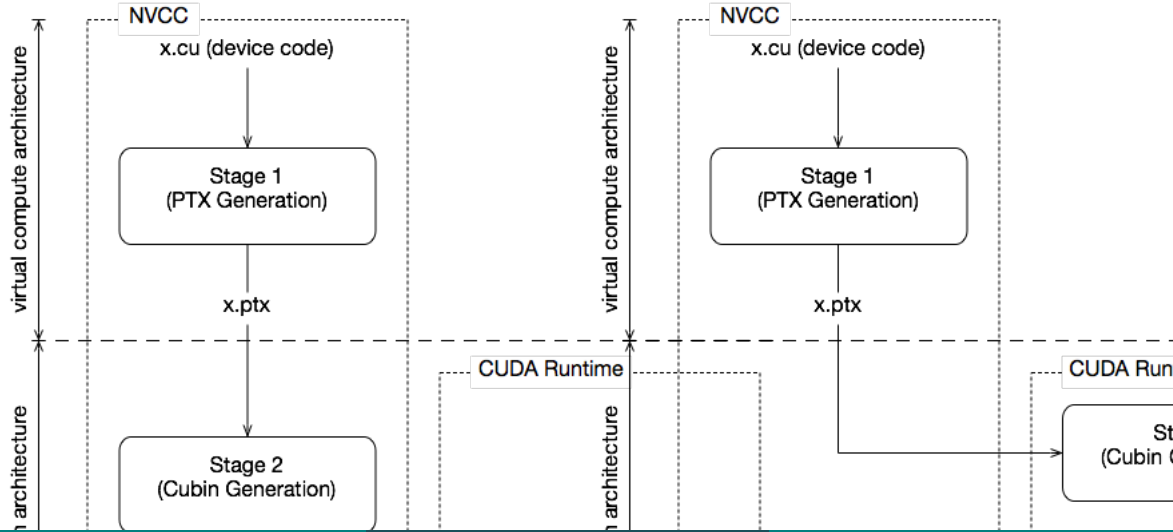
(v) Synthesized host code and the embedded fatbinary are linked together to generate the executable

- 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



Two-Staged Compilation with NVCC



PTX

SASS

NVCC Examples

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

Generate virtual PTX assembly assuming CC 3.0 while 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

Synchronization Constructs in CUDA

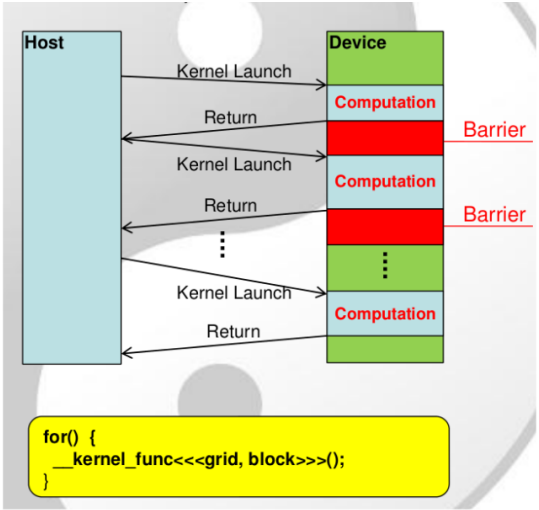
- `__syncThreads()` synchronizes threads within a block
- `cudaDeviceSynchronize()` synchronizes all threads in a grid
  - There are other variants
- Atomic operations prevent conflicts associated with multiple threads concurrently accessing a global or shared memory variable
  - For example, `float atomicAdd(float* addr, float amount)`

__syncthreads()

- 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

Synchronization among Grids

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



Atomic Operations

- Perform read-modify-write (RMW) atomic operations on data residing in global or shared memory
  - For example, `atomicAdd()`, `atomicSub()`, `atomicMin()`, `atomicMax()`, `atomicInc()`, `atomicDec()`, `atomicExch()`, `atomicCAS()`
- Predictable result when simultaneous access to memory is required

Scoped Synchronization

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 kernel1<<<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 is asynchronous
- Data transfer on line 3 cannot begin due to the device-side ordering (i.e., until the kernel completes)

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 kernel1<<<1,N>>>(d_a);
3 cudaMemcpy(h_res, d_a, numBytes, cudaMemcpyDeviceToHost);
```

```
1 cudaMemcpy(d_a, h_a, numBytes, cudaMemcpyHostToDevice);
2 kernel1<<<1,N>>>(d_a);
3 // Host gets work done
4 h_func(h_b);
5 cudaMemcpy(h_res, d_a, numBytes, cudaMemcpyDeviceToHost);
```

Goal is to Utilize GPU Hardware

- Overlap kernel execution with memory copy between host and device
- Overlap execution of multiple kernels if there are enough resources
- Depends on whether the GPU architecture supports overlapped execution

CUDA Streams

- 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

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 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,
5 cudaMemcpyHostToDevice, stream1);
6 increment<<<1,N,0,stream1>>>(d_a);
7 // Block the host thread
8 cudaStreamSynchronize(stream1);
9 res = cudaStreamDestroy(&stream1);
```

Why Use CUDA Streams?

- Memory copy and kernel execution can be overlapped if they occur in different, non-default streams
- Recent GPUs are capable of “concurrent copy and execution”, can be queried from the `deviceOverlap/asyncEngineCount` field of the `cudaDeviceProp` struct
- Individual kernels can overlap if there are enough resources on the GPU

Overlapping Kernel Execution and Data Transfers

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

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,
12   cudaMemcpyDeviceToHost, stream[i]);
13 }
```

Concurrent Host Execution

- Asynchronous functions are nonblocking
  - kernel launches
  - memory copies from host to device of a memory block of 64 KB or less
  - memory copies performed by functions that are suffixed with `Async`

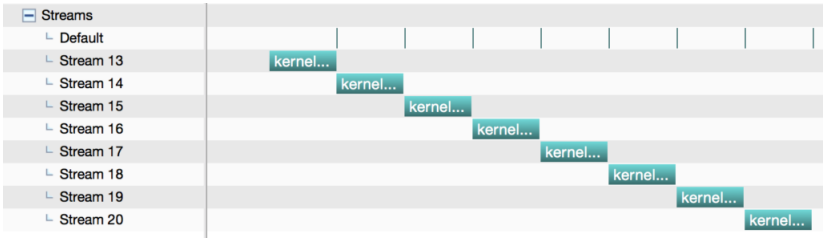
Streams and Concurrency in CUDA 7+

- Prior to CUDA 7, all host threads shared the default stream
  - Implied synchronization
- 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

GPU Pro Tip: CUDA 7 Streams Simplify Concurrency

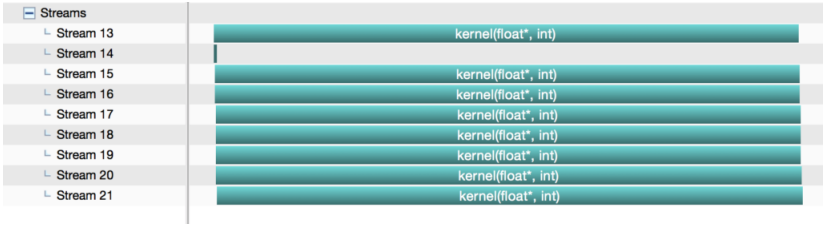
Multi-Stream Example: Legacy Behavior

```
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: Multi-Stream Example: Legacy Behavior

```
nvcc –default-stream per-thread <file.cu>
```



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

Desired Application Characteristics for Device Execution

- Large data-parallel computation
- Complex computation kernel to justify the data movement costs
  - Think of matrix addition versus matrix multiplication
  - Keep data on the device to avoid repeated transfers

Key Ideas for Extracting Performance

- 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)
- Memory access optimization
  - 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) {}
```

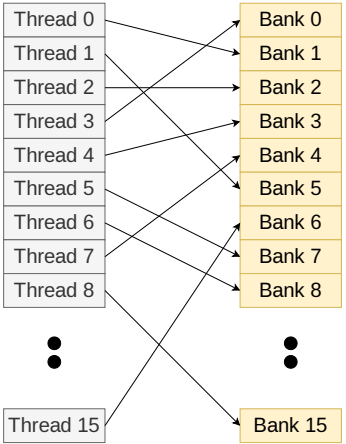
- Conditions evaluating to different truth-values for threads in a warp is bad
  - Creates two different control paths for threads in a block; branch granularity < warp size; threads 0 and 1 follow different path than the rest of the threads in the first warp

Parallel Memory Architecture

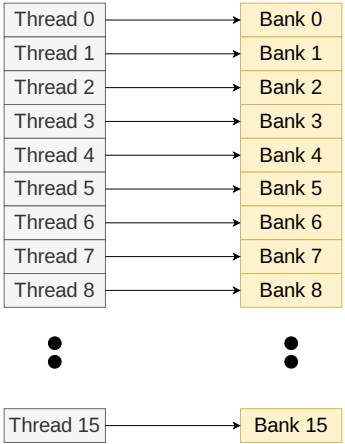
- In a parallel architecture, many threads access memory
- Memory is divided into banks to achieve high bandwidth
  - Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict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

Example of Bank Addressing

Random access

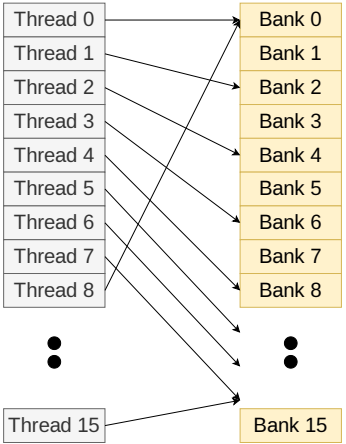


Linear access, stride=1

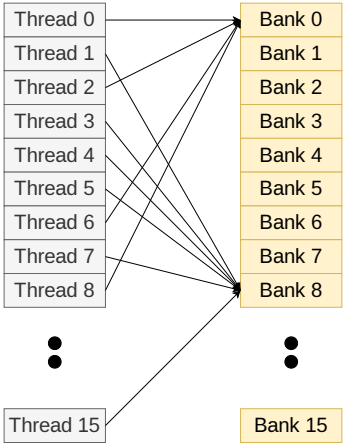


Example of Bank Addressing

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 the 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

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 access
  - A warp of threads accesses adjacent data in a cache line
  - In the best case, this results in one memory transaction (best bandwidth)
- Uncoalesced memory access
  - A warp of threads accesses scattered data in different cache lines leading to memory divergence
  - This may result in 32 different memory transactions (poor bandwidth)

Matrix Transpose

```
__global__ void copy(float *odata, const float *idata) {
  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)
    odata[(y+j)*width + x] = idata[(y+j)*width + x];
}
```

```
__global__ void transposeNaive(float *odata, const float *idata) {
  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)
    odata[x*width + (y+j)] = idata[(y+j)*width + x];
}
```

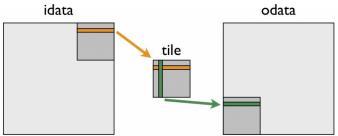
reads from `idata` are coalesced, but writes to `odata` have a stride of 1024

An Efficient Matrix Transpose in CUDA C/C++

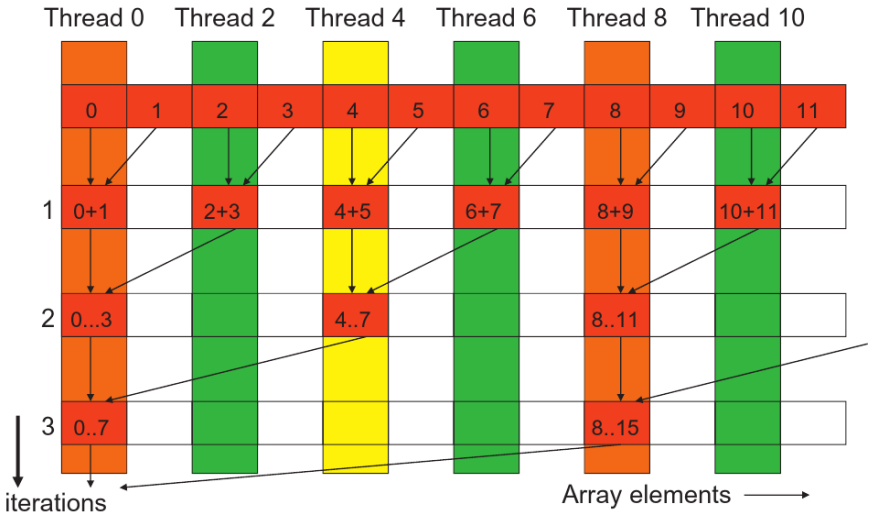
Optimizing Matrix Transpose

```
__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



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

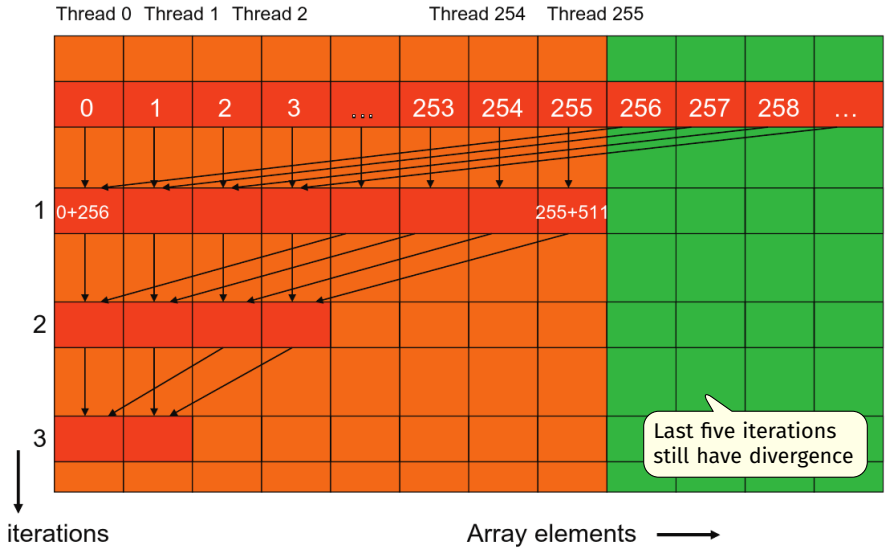
(vi) ...

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

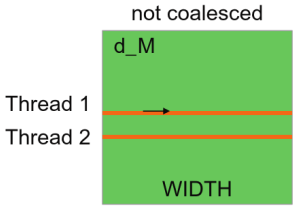


Avoid Divergence in the Last Few Iterations

```
for (unsigned int stride = blockDim.x/2; stride > 32; stride /= 2) {
  if (t < stride)
    partialSum[t] += partialSum[t+stride];
  __syncthreads();
}
if (tid < 32) {
  partialSum[tid] += partialSum[tid+32];
  partialSum[tid] += partialSum[tid+16];
  partialSum[tid] += partialSum[tid+8];
  partialSum[tid] += partialSum[tid+4];
  partialSum[tid] += partialSum[tid+2];
  partialSum[tid] += partialSum[tid+1];
}
```

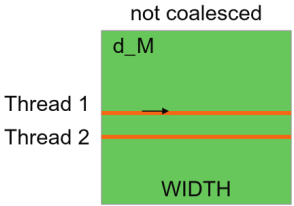
Matrix Multiplication Example

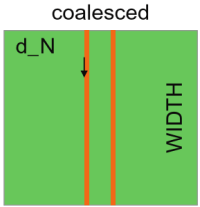
```
1  __global__ void matmulKernel(float* A, float* B, float*
       C) {
2    int row = blockIdx.y * blockDim.y + threadIdx.y;
3    int col = blockIdx.x * blockDim.x + threadIdx.x;
4    float tmp = 0;
5    if (row < N && col < N) {
6      // Each thread computes one element of the matrix
7      for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





Memory Divergence

- A warp can experience memory divergence if there is a significant difference between the times taken to service memory requests from the constituent threads

R. Ausavarungnirun et a. Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance. PACT'15.

Profiling CUDA Kernels with NVIDIA Nsight

Efficient Data Management

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.