EE382N (20): Computer Architecture - Parallelism and Locality
Fall 2011

Lecture 21 – GPUs (VI) + CUDA (I)

Mattan Erez

The University of Texas at Austin

Control

- Each SM has its own warp scheduler
- Schedules warps OoO based on hazards and resources
- Warps can be issued in any order within and across blocks
- Within a warp, all threads always have the same position
  - Current implementation has warps of 32 threads
  - Can change with no notice from NVIDIA

Conditionals within a Thread

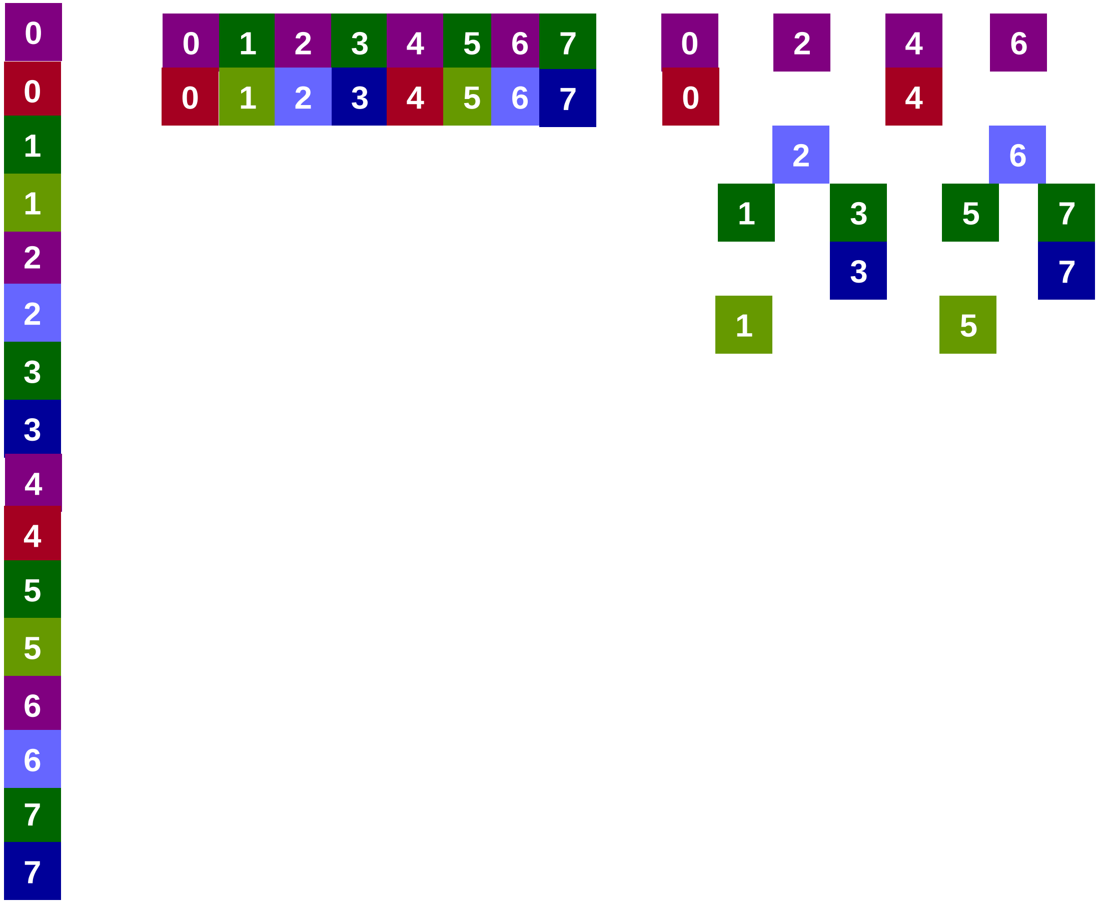
- What happens if there is a conditional statement within a thread?
- No problem if all threads in a warp follow same path
- ***Divergence***: threads in a warp follow different paths
  - HW will ensure correct behavior by (partially) serializing execution
  - Compiler can add predication to eliminate divergence
- Try to avoid divergence
  - If (TID > 2) {…} → If(TID / warp_size > 2) {…}

Control Flow

- Recap:
  - 32 threads in a warm are executed in SIMD (share one instruction sequencer)
  - Threads within a warp can be disabled (masked)
    - For example, handling bank conflicts
  - Threads contain arbitrary code including conditional branches
- How do we handle different conditions in different threads?
  - No problem if the threads are in different warps
  - Control ***divergence***
  - ***Predication***

Control Flow Divergence

```
if (TID % 2 == 0) {
 f2();
 if (TID % 4 == 0) {
   f4();
 }
 else {
   f2'();
 }
}
else {
 f(1);
 if (TID % 4 == 3) {
   f3();
 }
 else {
   f1'();
 }
}
```



Mask Stack Enables Divergence

IP

```
→ 1: if (TID % 2 == 0) {
  2:  f2();
  3:  if (TID % 4 == 0) {
  4:   f4();
  5:  }
  6:  else {
  7:   f2'();
  8:  }
  9: }
 10: else {
 11:  f(1);
 12:  if (TID % 4 == 3) {
 13:   f3();
 14:  }
 15:  else {
 16:   f1'();
 17:  }
 18: }
```

enable mask

1	1	1	1	1	1	1	1

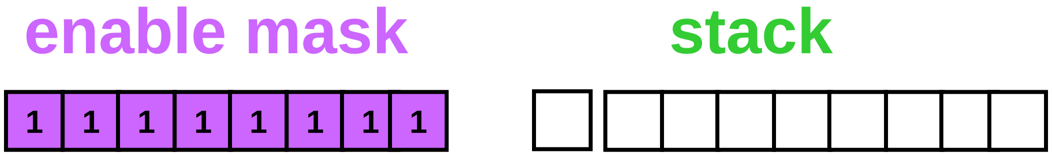
stack

IP	1	1	1	1	1	1	1	1

Mask Stack Enables Divergence

IP

```
→1: if (TID % 2 == 0) {
 2:  f2();
 3:  if (TID % 4 == 0) {
 4:   f4();
 5:  }
 6:  else {
 7:   f2'();
 8:  }
 9: }
10: else {
11:  f(1);
12:  if (TID % 4 == 3) {
13:   f3();
14:  }
15:  else {
16:   f1'();
17:  }
18: }
```



enable mask

1	1	1	1	1	1	1	1

stack

Mask Stack Enables Divergence

IP

```
→ 1: if (TID % 2 == 0) {
  2:  f2();
  3:  if (TID % 4 == 0) {
  4:   f4();
  5:  }
  6:  else {
  7:   f2'();
  8:  }
  9: }
 10: else {
 11:  f(1);
 12:  if (TID % 4 == 3) {
 13:   f3();
 14:  }
 15:  else {
 16:   f1'();
 17:  }
 18: }
```

enable mask

1	1	1	1	1	1	1	1

stack

9	1	1	1	1	1	1	1	1

Mask Stack Enables Divergence

IP

```
   1: if (TID % 2 == 0) {
 → 2:  f2();
   3:  if (TID % 4 == 0) {
   4:   f4();
   5:  }
   6:  else {
   7:   f2'();
   8:  }
   9: }
  10: else {
  11:  f(1);
  12:  if (TID % 4 == 3) {
  13:   f3();
  14:  }
  15:  else {
  16:   f1'();
  17:  }
  18: }
```

enable mask

1	0	1	0	1	0	1	0

stack

9	1	1	1	1	1	1	1	1

Mask Stack Enables Divergence

IP

```
   1: if (TID % 2 == 0) {
   2:  f2();
➡ 3:  if (TID % 4 == 0) {
   4:   f4();
   5:  }
   6:  else {
   7:   f2'();
   8:  }
   9: }
  10: else {
  11:  f(1);
  12:  if (TID % 4 == 3) {
  13:   f3();
  14:  }
  15:  else {
  16:   f1'();
  17:  }
  18: }
```

enable mask

1	0	1	0	1	0	1	0

stack

9	1	1	1	1	1	1	1	1

Mask Stack Enables Divergence

IP

```
   1: if (TID % 2 == 0) {
   2:  f2();
→  3:  if (TID % 4 == 0) {
   4:   f4();
   5:  }
   6:  else {
   7:   f2'();
   8:  }
   9: }
  10: else {
  11:  f(1);
  12:  if (TID % 4 == 3) {
  13:   f3();
  14:  }
  15:  else {
  16:   f1'();
  17:  }
  18: }
```

enable mask

1	0	1	0	1	0	1	0

stack

5	1	0	1	0	1	0	1	0
9	1	1	1	1	1	1	1	1

Mask Stack Enables Divergence

IP

```
  1: if (TID % 2 == 0) {
  2:  f2();
  3:  if (TID % 4 == 0) {
→ 4:   f4();
  5:  }
  6:  else {
  7:   f2'();
  8:  }
  9: }
 10: else {
 11:  f(1);
 12:  if (TID % 4 == 3) {
 13:   f3();
 14:  }
 15:  else {
 16:   f1'();
 17:  }
 18: }
```

enable mask

1	0	0	0	1	0	0	0

stack

5	1	0	1	0	1	0	1	0
9	1	1	1	1	1	1	1	1

Mask Stack Enables Divergence

IP

```
   1: if (TID % 2 == 0) {
   2:  f2();
   3:  if (TID % 4 == 0) {
   4:   f4();
→  5:  }
   6:  else {
   7:   f2'();
   8:  }
   9: }
  10: else {
  11:  f(1);
  12:  if (TID % 4 == 3) {
  13:   f3();
  14:  }
  15:  else {
  16:   f1'();
  17:  }
  18: }
```

enable mask

1	0	0	0	1	0	0	0

stack

5	1	0	1	0	1	0	1	0
9	1	1	1	1	1	1	1	1

Mask Stack Enables Divergence

IP

```
   1: if (TID % 2 == 0) {
   2:  f2();
   3:  if (TID % 4 == 0) {
   4:   f4();
→  5:  }
   6:  else {
   7:   f2'();
   8:  }
   9: }
  10: else {
  11:  f(1);
  12:  if (TID % 4 == 3) {
  13:   f3();
  14:  }
  15:  else {
  16:   f1'();
  17:  }
  18: }
```

enable mask

6	1	0	1	0	1	0	1	0

stack

9	1	1	1	1	1	1	1	1

Mask Stack Enables Divergence

IP

```
   1: if (TID % 2 == 0) {
   2:  f2();
   3:  if (TID % 4 == 0) {
   4:   f4();
   5:  }
→  6:  else {
   7:   f2'();
   8:  }
   9: }
  10: else {
  11:  f(1);
  12:  if (TID % 4 == 3) {
  13:   f3();
  14:  }
  15:  else {
  16:   f1'();
  17:  }
  18: }
```

enable mask

1	0	1	0	1	0	1	0

stack

8	1	0	1	0	1	0	1	0
9	1	1	1	1	1	1	1	1

Mask Stack Enables Divergence

IP

```
   1: if (TID % 2 == 0) {
   2:  f2();
   3:  if (TID % 4 == 0) {
   4:   f4();
   5:  }
   6:  else {
→  7:   f2'();
   8:  }
   9: }
  10: else {
  11:  f(1);
  12:  if (TID % 4 == 3) {
  13:   f3();
  14:  }
  15:  else {
  16:   f1'();
  17:  }
  18: }
```

enable mask

0	0	1	0	0	0	1	0

stack

8	1	0	1	0	1	0	1	0
9	1	1	1	1	1	1	1	1

Mask Stack Enables Divergence

IP

enable mask

stack

```
 1: if (TID % 2 == 0) {
 2:  f2();
 3:  if (TID % 4 == 0) {
 4:   f4();
 5:  }
 6:  else {
 7:   f2'();
 8:  }
 9: }
10: else {
11:  f(1);
12:  if (TID % 4 == 3) {
13:   f3();
14:  }
15:  else {
16:   f1'();
17:  }
18: }
```

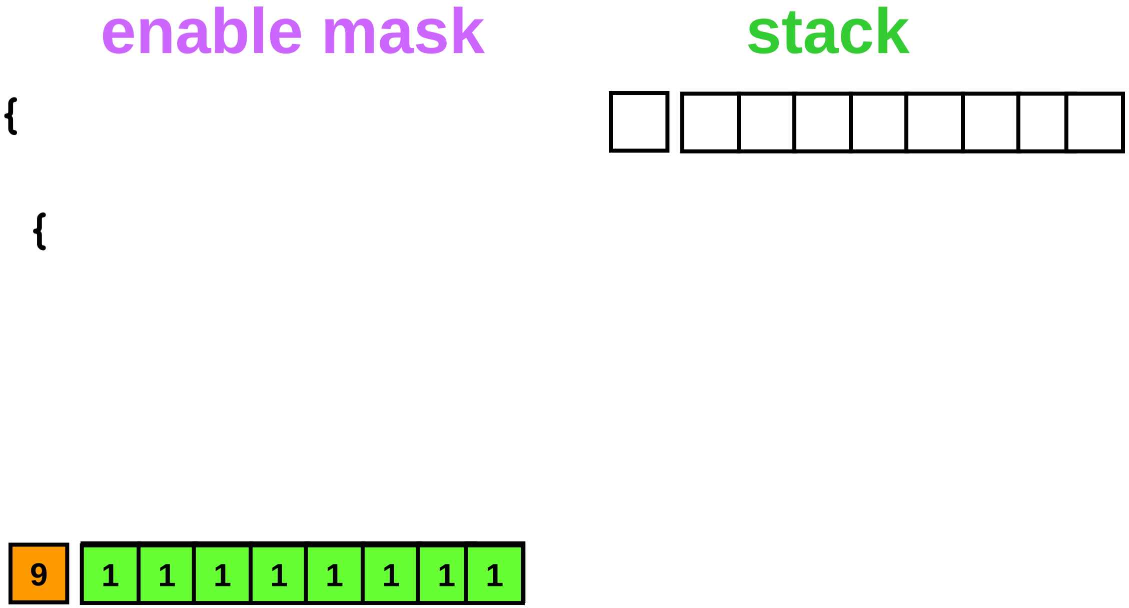
9 | 1 1 1 1 1 1 1 1

8 | 1 0 1 0 1 0 1 0

Mask Stack Enables Divergence

IP

```
 1: if (TID % 2 == 0) {
 2:  f2();
 3:  if (TID % 4 == 0) {
 4:   f4();
 5:  }
 6:  else {
 7:   f2'();
 8:  }
 9: }
10: else {
11:  f(1);
12:  if (TID % 4 == 3) {
13:   f3();
14:  }
15:  else {
16:   f1'();
17:  }
18: }
```



DirectX 10 specifies 4-deep stack

No restrictions today (need to track many PCs)

Predication Eliminates Branches (and Divergence)

```
if (TID % 2 == 0) {
 f2();
 if (TID % 4 == 0) {
   f4();
 }
 else {
   f2’();
 }
}
else {
 f(1);
 if (TID % 4 == 3) {
   f3();
 }
 else {
   f1’();
 }
}
```

Predication Eliminates Branches (and Divergence)

```
   p1 = (TID % 2 == 0)
p1 f2();
```

```
if (TID % 2 == 0) {
 f2();
 if (TID % 4 == 0) {
   f4();
 }
 else {
   f2’();
 }
}
else {
 f(1);
 if (TID % 4 == 3) {
   f3();
 }
 else {
   f1’();
 }
}
```

Predication Eliminates Branches (and Divergence)

```
   p1 = (TID % 2 == 0)
p1 f2();
p1 p2 = (TID % 4 == 0)
p2 f4();
```

```
if (TID % 2 == 0) {
 f2();
 if (TID % 4 == 0) {
   f4();
 }
 else {
   f2’();
 }
}
else {
 f(1);
 if (TID % 4 == 3) {
   f3();
 }
 else {
   f1’();
 }
}
```

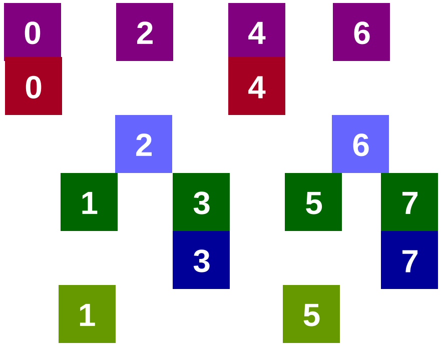
Predication Eliminates Branches (and Divergence)

```
    p1 = (TID % 2 == 0)
p1  f2();
p1  p2 = (TID % 4 == 0)
p2  f4();

p1  p3 = !p2
p3  f2’();



    p4 = !p1
p4  f(1);
p4  p5 = (TID % 3 == 0)
p5  f3();

p4  p6 = !p5
p6  f1’();
```

```
if (TID % 2 == 0) {
 f2();
 if (TID % 4 == 0) {
   f4();
 }
 else {
   f2’();
 }
}
else {
 f(1);
 if (TID % 4 == 3) {
   f3();
 }
 else {
   f1’();
 }
}
```

Equivalence of Divergence and Predication

```
   p1 = (TID % 2 == 0)
p1 f2();
p1 p2 = (TID % 4 == 0)
p2 f4();

p1 p3 = !p2
p3 f2'();


   p4 = !p1
p4 f(1);
p4 p5 = (TID % 4 == 3)
p5 f3();

p4 p6 = !p5
p6 f1'();
```



```
if (TID % 2 == 0) {
 f2();
 if (TID % 4 == 0) {
   f4();
 }
 else {
   f2'();
 }
}
else {
 f(1);
 if (TID % 4 == 3) {
   f3();
 }
 else {
   f1'();
 }
}
```

When to Predicate and When to Diverge?

- Divergence
  - No performance penalty if all warp branches the same way
  - Some extra HW cost
  - Static partitioning of stack resources (to warps)
- Predication
  - **Always execute all paths**
  - Expose more ILP
  - Add predication registers to instruction encoding
- Selects – software predication
  - Simpler HW and just as flexible mode
  - Simple instruction encoding
  - Need to use more registers and insert select instructions

Outline

- CUDA
  - Overview
  - Development process
  - Performance Optimization
  - Syntax

- Most slides courtesy Massimiliano Fatica (NVIDIA)

Compute Unified Device Architecture

- CUDA is a programming system for utilizing NVIDIA GPUs for compute
  - CUDA follows the architecture very closely
- General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor

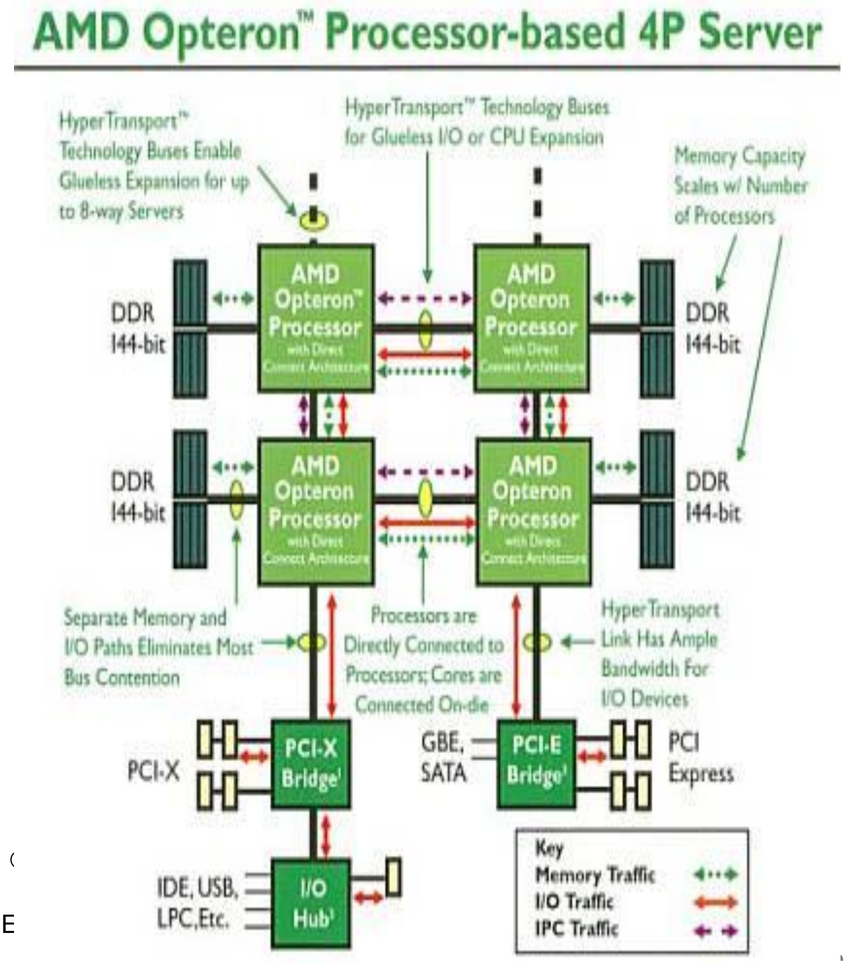
Matches architecture features

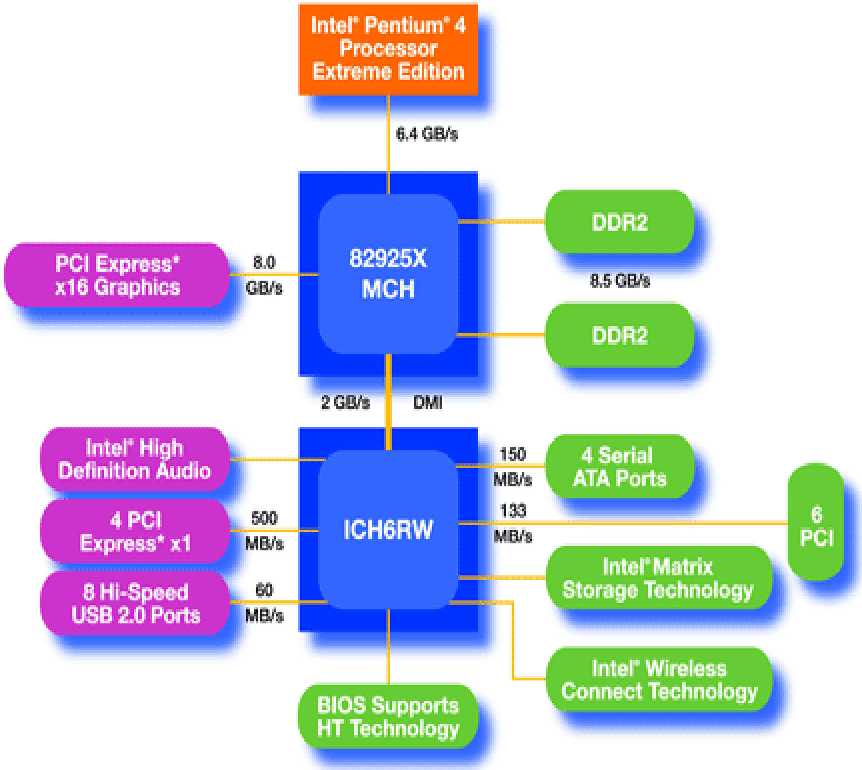
Specific parameters not exposed

© David Kirk/NVIDIA and
Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois,
Urbana-Champaign

The CUDA Platform

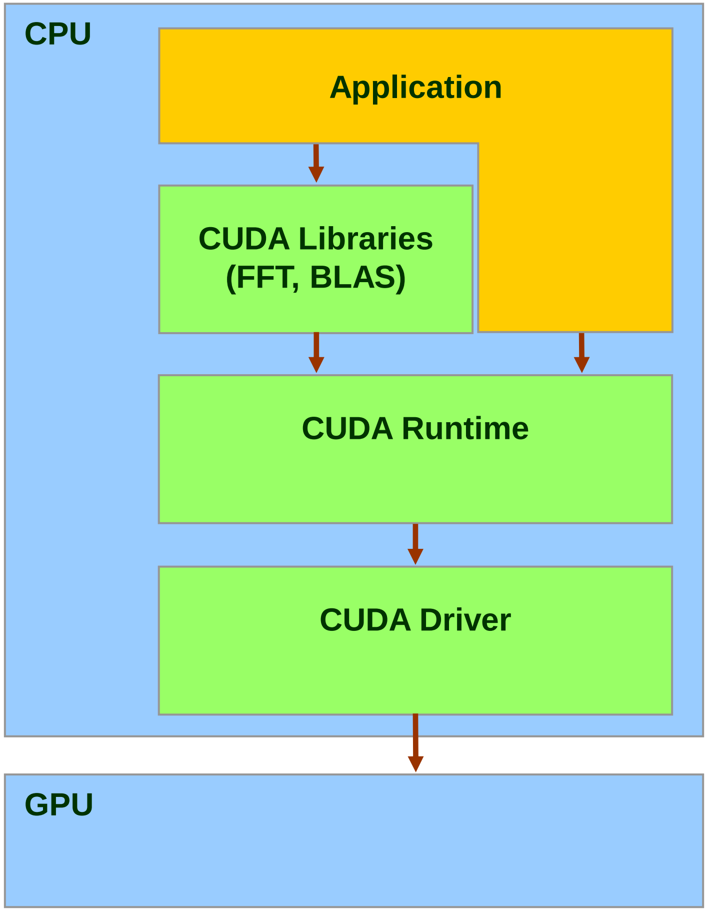
- High-end NVIDIA GPUs not integrated into the CPU
  - GPU connects through a PCI Express bus
  - GPU communicates through OS (drivers)


AMD Opteron™ Processor-based 4P Server
HyperTransport™ Technology Buses Enable Glueless Expansion for up to 8-way Servers
HyperTransport™ Technology Buses for Glueless I/O or CPU Expansion
Memory Capacity Scales w/ Number of Processors
DDR 144-bit
AMD Opteron™ Processor with Direct Connect Architecture
AMD Opteron Processor with Direct Connect Architecture
DDR 144-bit
DDR 144-bit
AMD Opteron Processor with Direct Connect Architecture
AMD Opteron Processor with Direct Connect Architecture
DDR 144-bit
Separate Memory and I/O Paths Eliminates Most Bus Contention
Processors are Directly Connected to Processors; Cores are Connected On-die
HyperTransport Link Has Ample Bandwidth For I/O Devices
PCI-X
PCI-X Bridge¹
GBE, SATA
PCI-E Bridge¹
PCI Express
IDE, USB, LPC,Etc.
I/O Hub¹
Key
Memory Traffic
I/O Traffic
IPC Traffic


Intel® Pentium® 4 Processor Extreme Edition
6.4 GB/s
PCI Express* x16 Graphics
8.0 GB/s
82925X MCH
DDR2
8.5 GB/s
DDR2
2 GB/s
DMI
Intel® High Definition Audio
4 PCI Express* x1
500 MB/s
8 Hi-Speed USB 2.0 Ports
60 MB/s
ICH6RW
150 MB/s
4 Serial ATA Ports
133 MB/s
6 PCI
Intel® Matrix Storage Technology
Intel® Wireless Connect Technology
BIOS Supports HT Technology

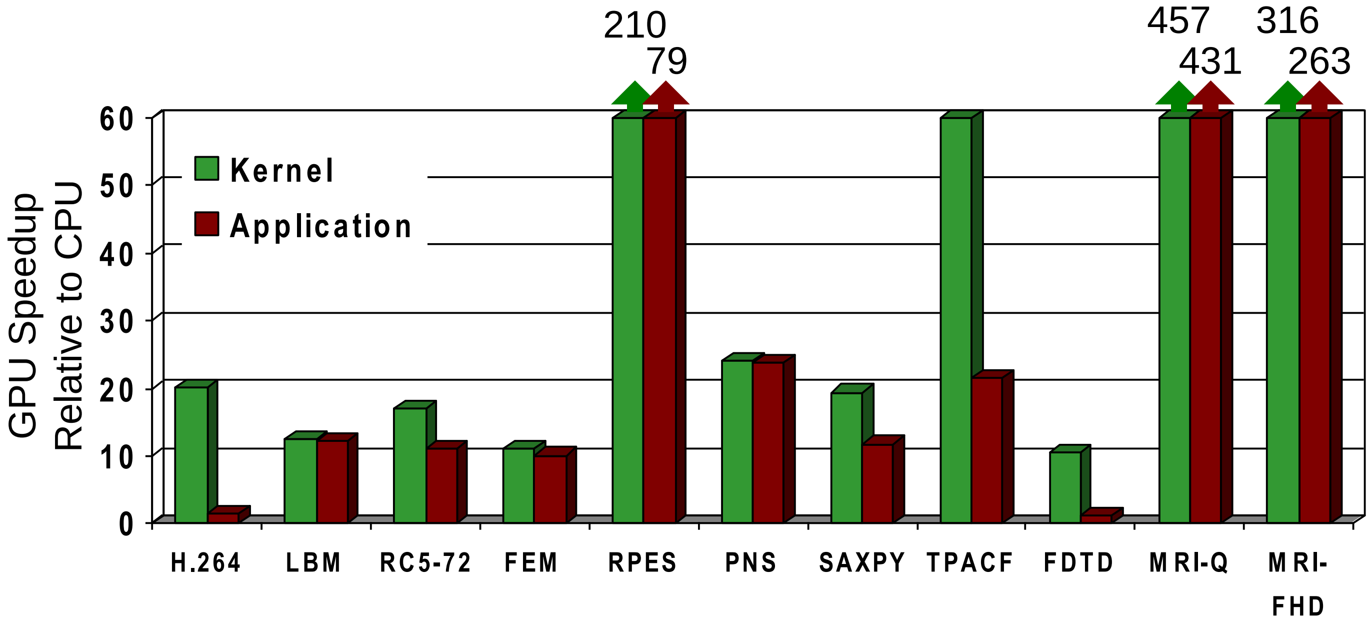
CUDA Programming System

- Targeted software stack
  - Compute oriented drivers, language, and tools
- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute - graphics free API
  - Data sharing with OpenGL buffer objects
  - Guaranteed maximum download & readback speeds
  - Explicit GPU memory management

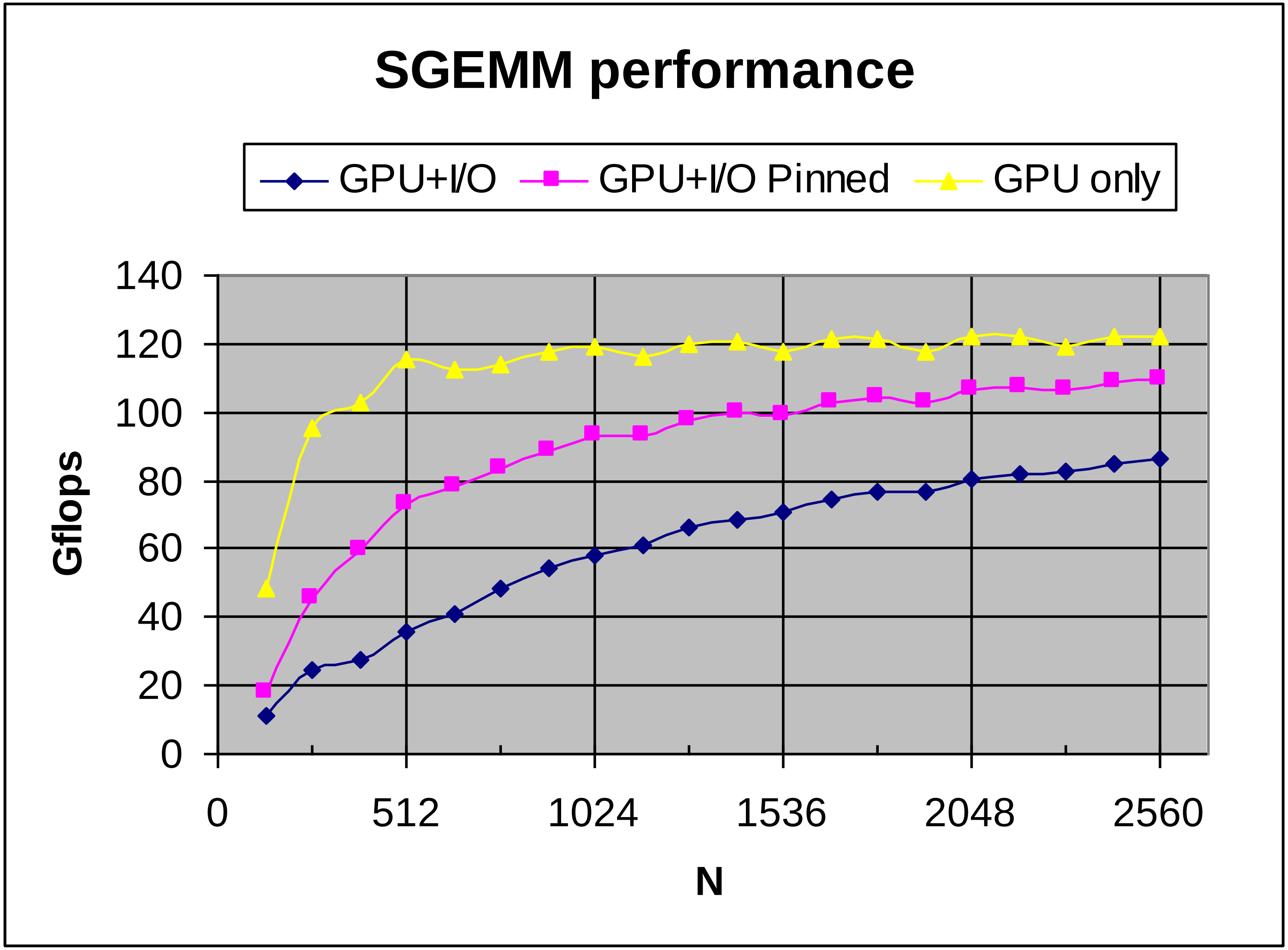


© NVIDIA Corp.

Overall Performance Can be Limited by Interface



Overall Performance Can be Limited by Interface



© NVIDIA Corp.

CUDA API and Language: Easy and Lightweight

- The API is an extension to the ANSI C programming language

  → Low learning curve

- The hardware is designed to enable lightweight runtime and driver

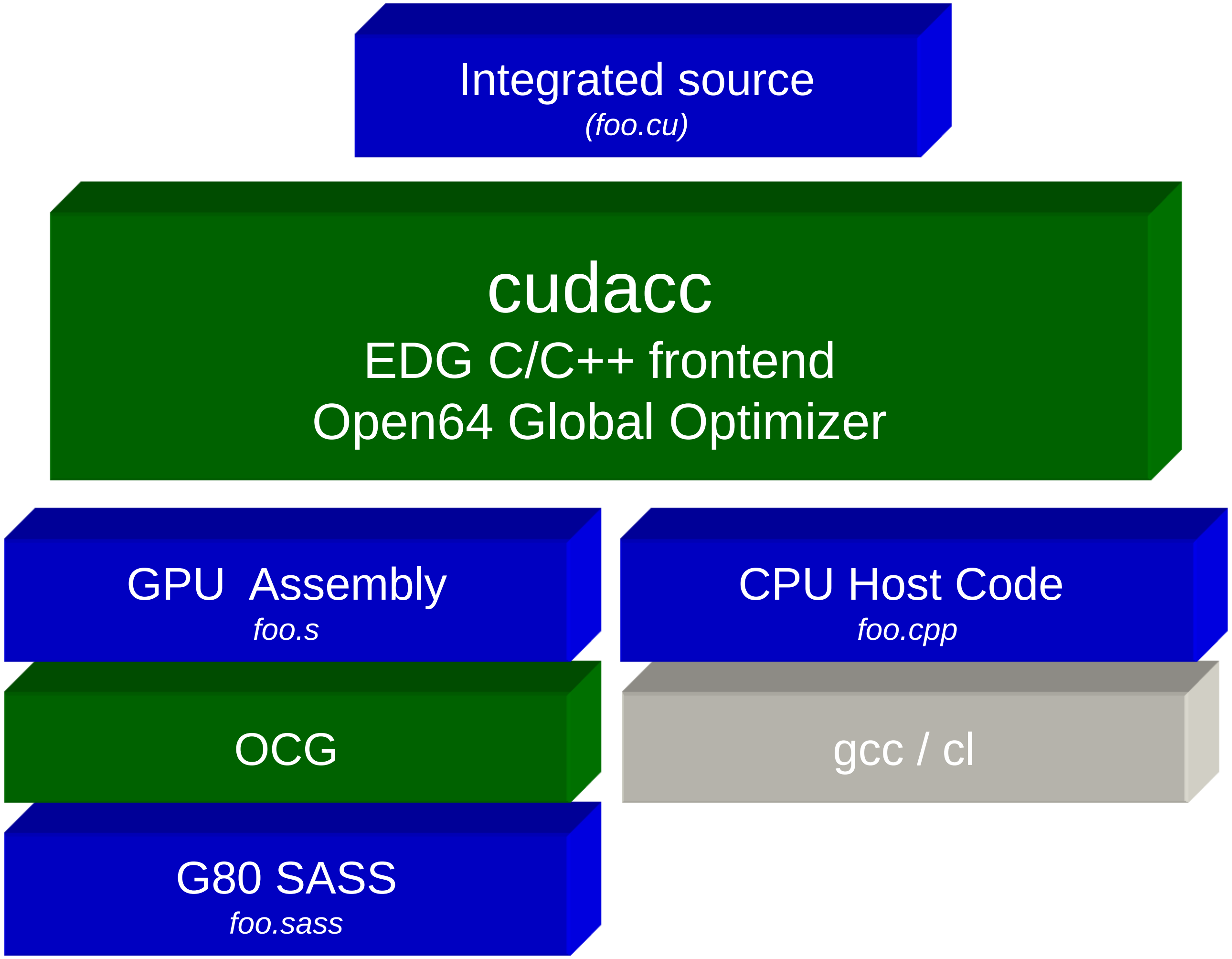
  → High performance

CUDA Programming Model: A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - ti-core CPU needs only a few

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, University of Illinois, Urbana-Champaign

CUDA is an Extension to C++



© David Kirk/NVIDIA and
Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois,
Urbana-Champaign

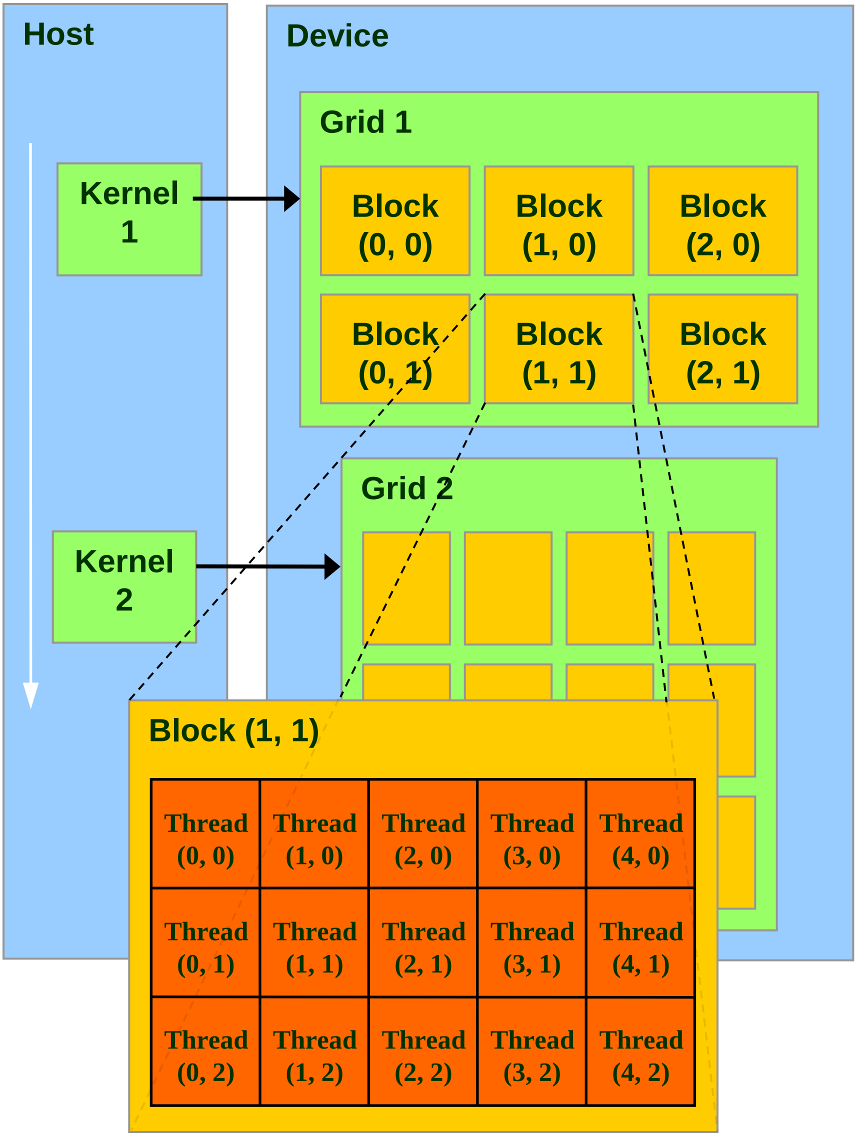
CUDA is an Extension to C++

- Declspecs
  - global, device, shared, local, constant
- Keywords
  - threadIdx, blockIdx
- Intrinsics
  - __syncthreads
  - __theradfence
- Runtime API
  - Memory, symbol, execution management
- Function launch

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

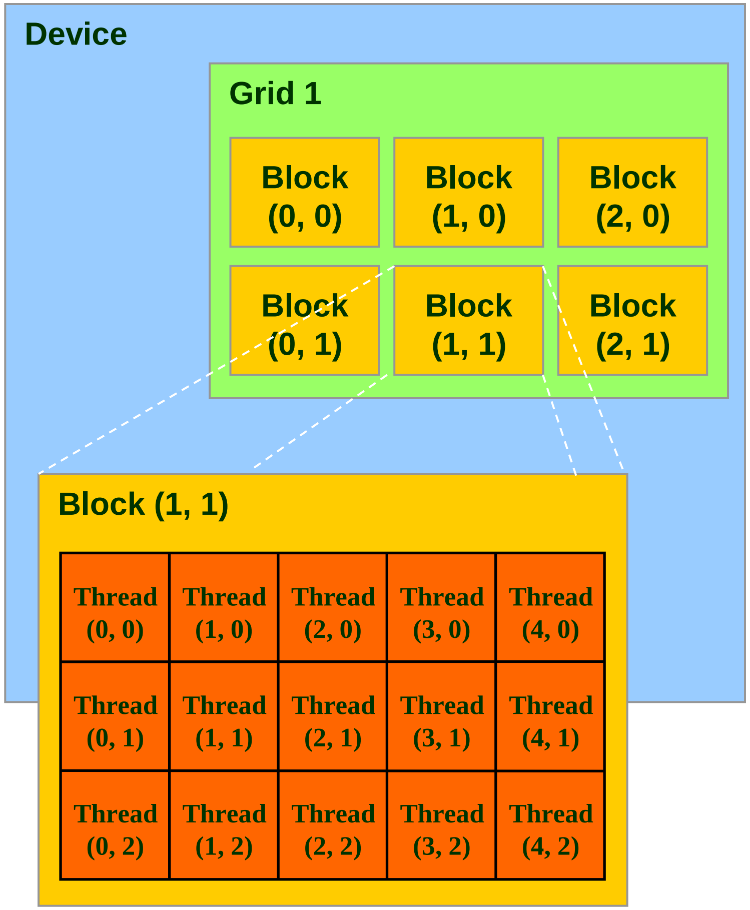
Thread Batching: Grids and Blocks

- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate



Block and Thread IDs

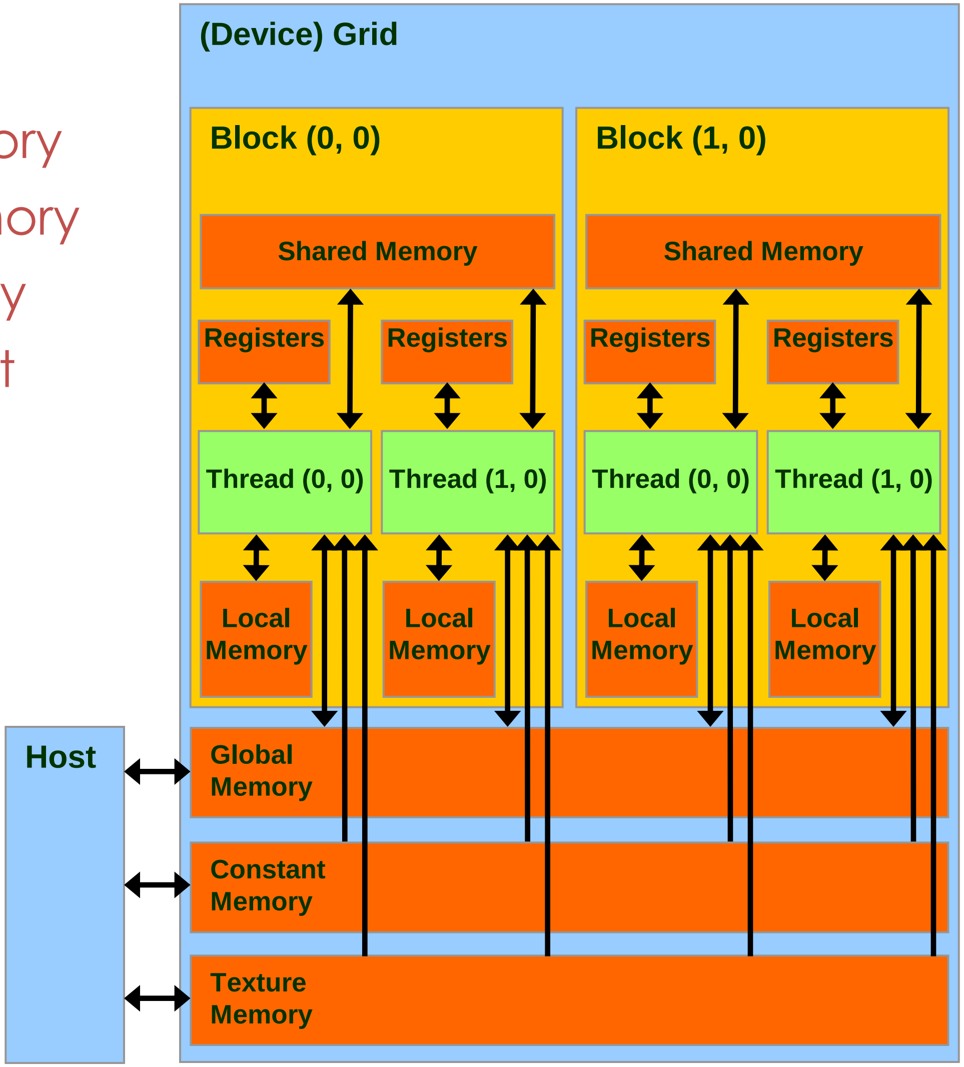
- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



© NVIDIA Corp.

CUDA Device Memory Space Overview

- Each thread 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
  - Read only per-grid texture memory
- The host can R/W global, constant, and texture memories



© NVIDIA Corp.

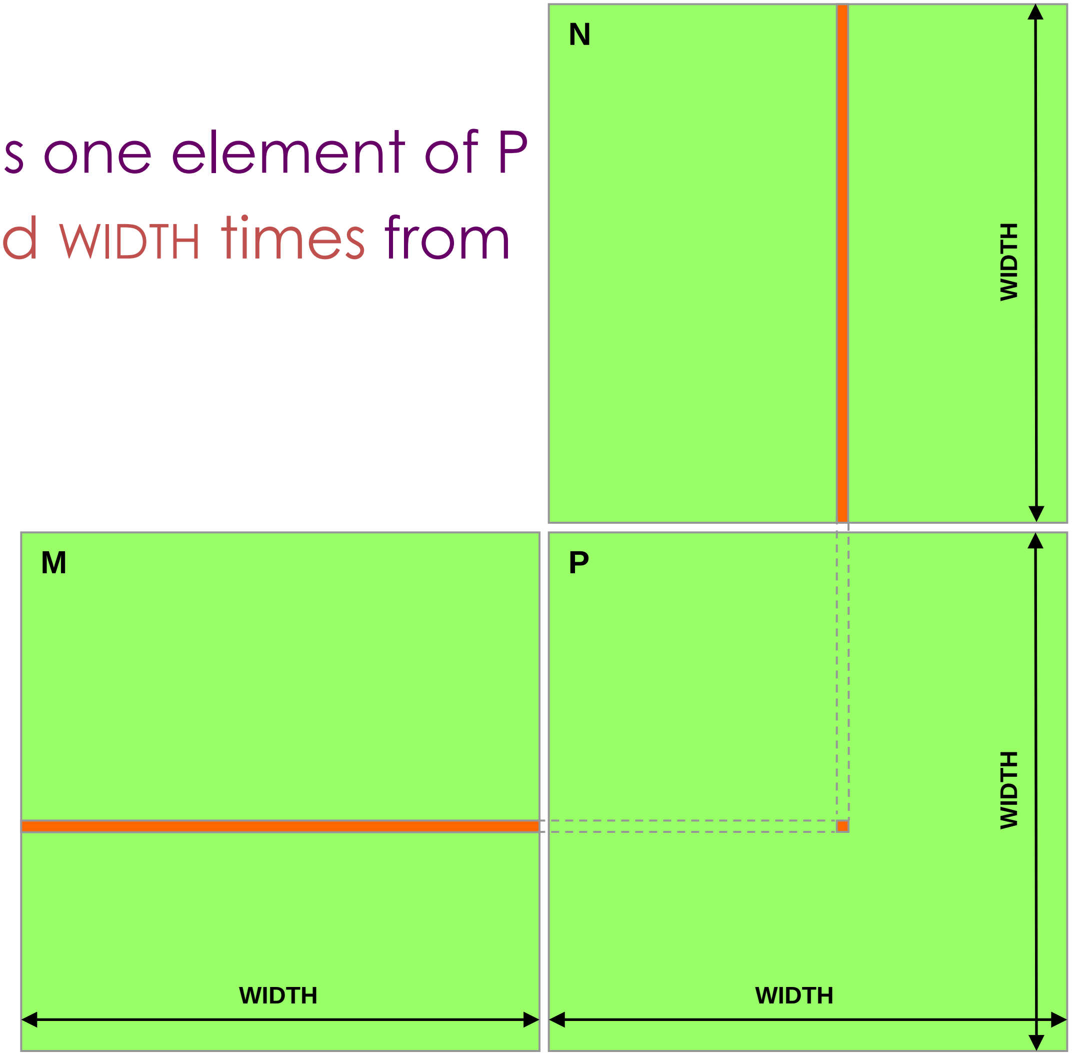
Access Times

- Register – dedicated HW - single cycle
- Shared Memory – dedicated HW - two cycles
  - Hidden by warps
- Local Memory – DRAM, no cache - *slow*
- Global Memory – DRAM, no cache - *slow*
- Constant Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Instruction Memory (invisible) – DRAM, cached

(
Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois,
Urbana-Champaign

Programming Model: Square Matrix Multiplication Example

- P = M * N of size WIDTH x WIDTH
- Without blocking:
  - One **thread** handles one element of P
  - M and N are loaded WIDTH times from global memory



© NVIDIA Corp.

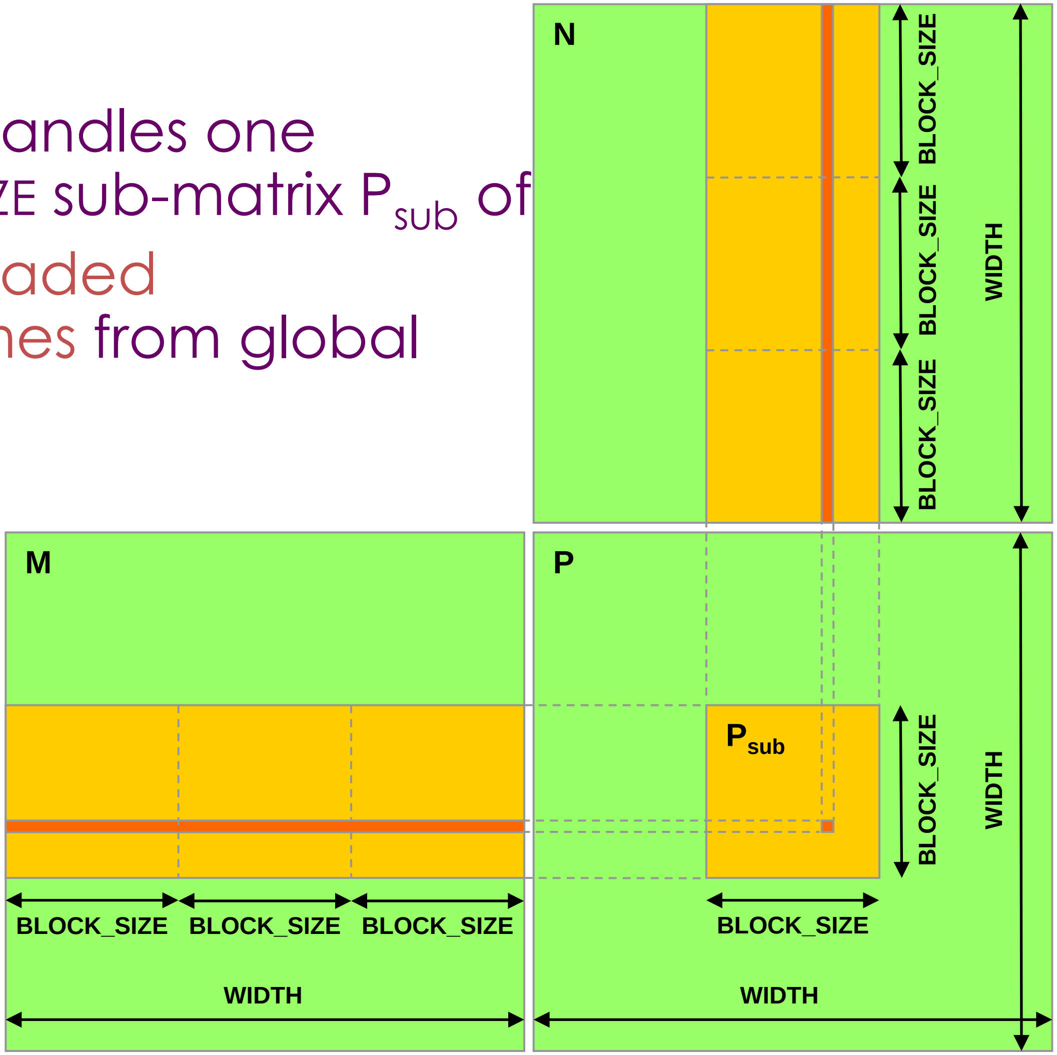
Programming Model: Common Programming Pattern

- Local and global memory reside in device memory (DRAM) - much slower access than shared memory
  - Uncached
- So, a common way of scheduling some computation on the device is to block it up to take advantage of fast shared memory:
  - Partition the data set into data subsets that fit into shared memory
  - Handle each data subset with one thread block by:
    - Loading the subset from global memory to shared memory
    - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory to global memory

© NVIDIA Corp.

Programming Model: Square Matrix Multiplication Example

- P = M * N of size WIDTH x WIDTH
- With blocking:
  - One **thread block** handles one BLOCK_SIZE x BLOCK_SIZE sub-matrix P_{sub} of P
  - M and N are only loaded WIDTH / BLOCK_SIZE times from global memory



- Great saving of memory bandwidth!

A quick review

- device = GPU = set of multiprocessors
- Multiprocessor = set of processors & shared memory
- Kernel = GPU program
- Grid = array of thread blocks that execute a kernel
- Thread block = CTA = group of SIMD threads that execute a kernel and can communicate via shared memory

Memory	Location	Cached	Access	Who
Local	Off-chip	**Yes**(No)	Read/write	One thread
Shared	On-chip	N/A	Read/write	All threads in a CTA
Global	Off-chip	**L1?/L2** (No)	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

- (No) – Not cacheable on G80

CUDA: C on the GPU

- A simple, explicit programming language solution
- Extend only where necessary

```
__global__ void KernelFunc(...);

__shared__ int SharedVar;

KernelFunc<<< 500, 128 >>>(...);
```

- Explicit GPU memory allocation
  - `cudaMalloc(),cudaFree()`
- Memory copy from host to device, etc.
  - `cudaMemcpy(),cudaMemcpy2D(),` ...

© NVIDIA Corp.

Example: Vector Addition Kernel

```
// Pair-wise addition of vector elements
// One thread per addition

__global__ void
vectorAdd(float* iA, float* iB, float* oC)
{
    int idx = threadIdx.x + blockDim.x *
  blockId.x;
    oC[idx] = iA[idx] + iB[idx];
}
```

Example: Vector Addition Host Code

```
float* h_A = (float*) malloc(N * sizeof(float));
float* h_B = (float*) malloc(N * sizeof(float));
// … initalize h_A and h_B

// allocate device memory
float* d_A, d_B, d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float),
   cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float),
   cudaMemcpyHostToDevice) );

// execute the kernel on N/256 blocks of 256 threads each
vectorAdd<<< N/256, 256>>>( d_A, d_B, d_C);
```

Outline

- Bandwidths
- CUDA
  - Overview
  - Development process
  - Performance Optimization
  - Syntax

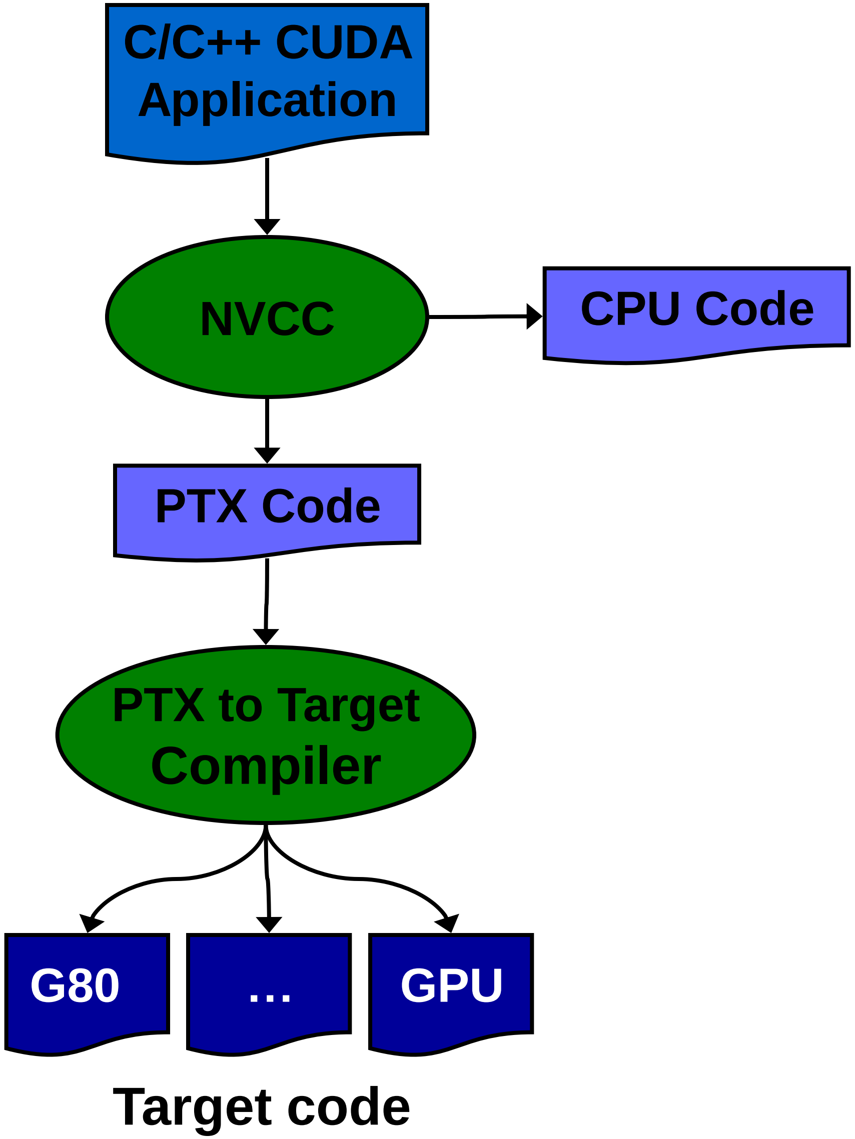
- Most slides courtesy Massimiliano Fatica (NVIDIA)

Compilation

- Any source file containing CUDA language extensions must be compiled with nvcc
- NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- NVCC can output:
  - Either C code (CPU Code)
    - That must then be compiled with the rest of the application using another tool
  - Or PTX object code directly
- Any executable with CUDA code requires two dynamic libraries:
  - The CUDA runtime library (`cudart`)
  - The CUDA core library (`cuda`)

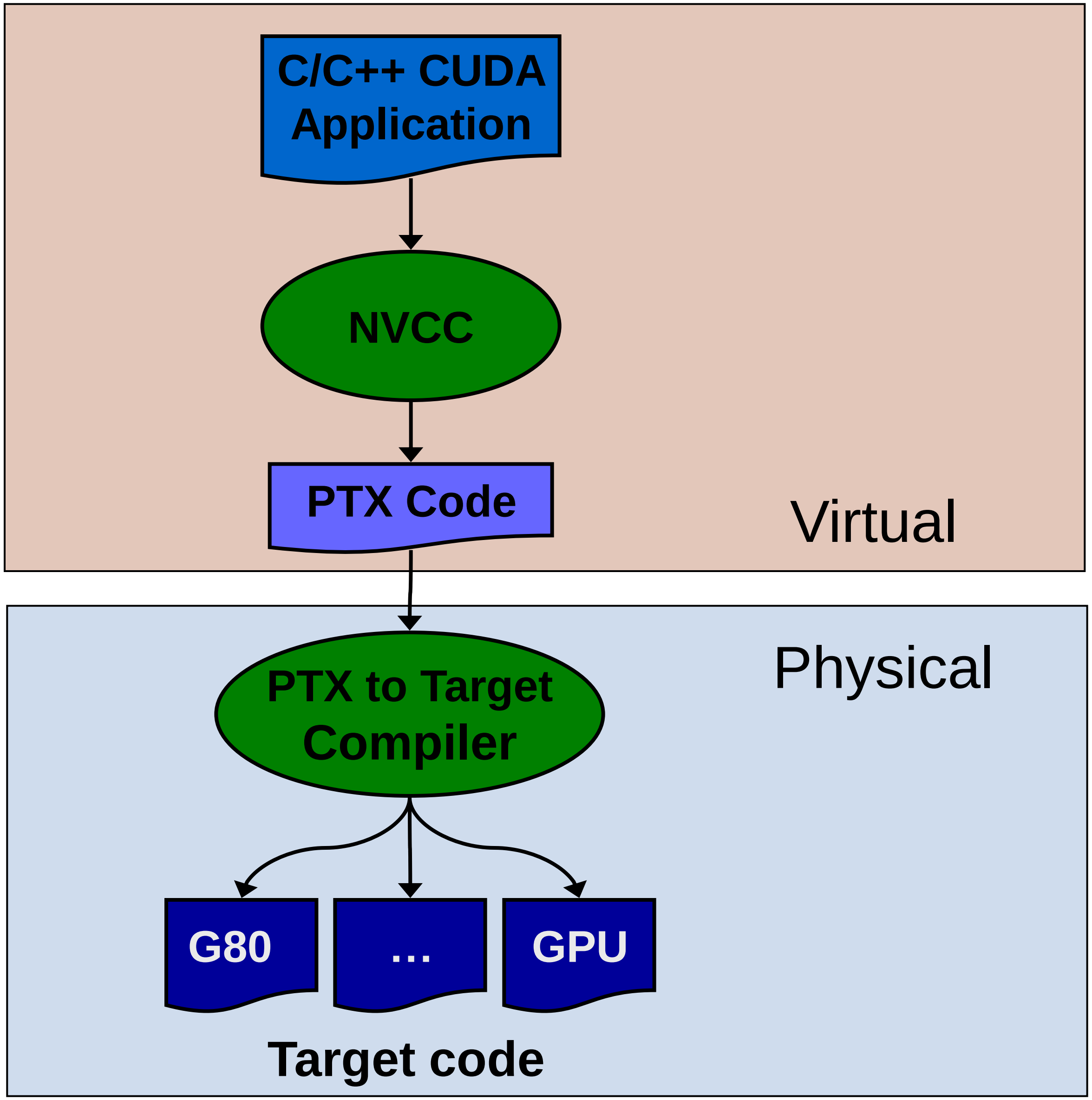
© NVIDIA Corp.

Compiling CUDA



© NVIDIA Corp.

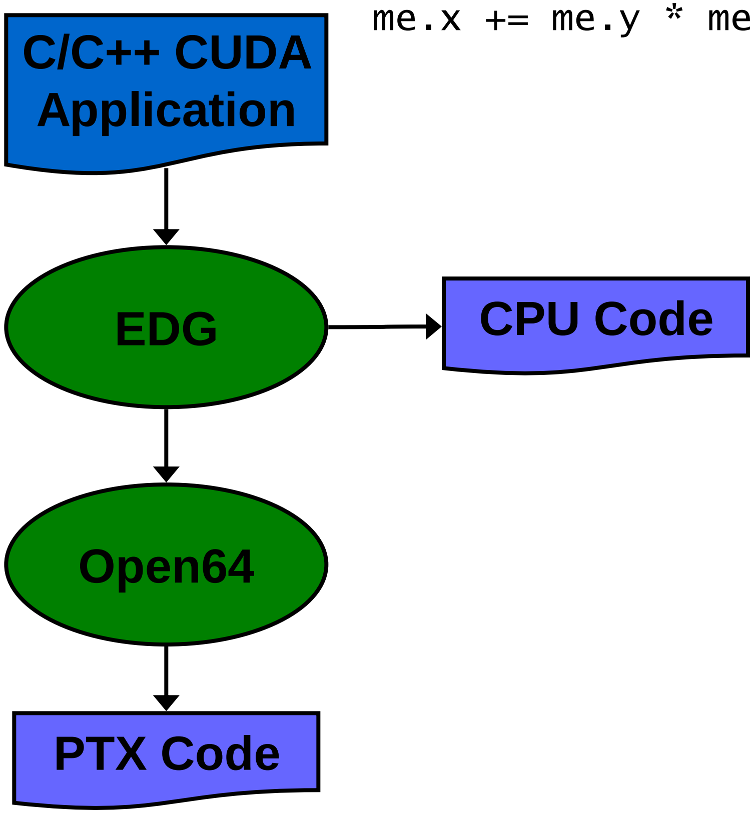
Compiling CUDA



© NVIDIA Corp.

NVCC & PTX Virtual Machine

```
float4 me = gx[gtid];
me.x += me.y * me.z;
```



- EDG
  - Separate GPU vs. CPU code
- Open64
  - Generates GPU PTX assembly
- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Programming model
  - Execution resources and state

```
ld.global.v4.f32  {$f1,$f3,$f5,$f7}, [$r9+0];
mad.f32           $f1, $f5, $f3, $f1;
```

© NVIDIA Corp.

Role of Open64

Open64 compiler gives us

- A complete C/C++ compiler framework. Forward looking. We do not need to add infrastructure framework as our hardware arch advances over time.
- A good collection of high level architecture independent optimizations. All GPU code is in the inner loop.
- Compiler infrastructure that interacts well with other related standardized tools.

Debugging Using the Device Emulation Mode

- An executable compiled in device emulation mode (`nvcc -deviceemu`) runs completely on the host using the CUDA runtime
  - No need of any device and CUDA driver
  - Each device thread is emulated with a host thread

- When running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. `printf`) and vice-versa
  - Detect deadlock situations caused by improper usage of pthreads

Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so simultaneous accesses of the same memory location by multiple threads potentially produce different results
- Dereferencing device pointers on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode
- Results of floating-point computations will slightly differ because of:
  - Different compiler outputs
  - Different instruction sets
  - Use of extended precision for intermediate results
    - There are various options to force strict single precision on the host

Parameterize Your Application

- Parameterization helps adaptation to different GPUs
- GPUs vary in many ways
  - # of multiprocessors
  - Shared memory size
  - Register file size
  - Threads per block
  - Memory bandwidth
- You can even make apps self-tuning (like FFTW)
  - "Experiment" mode discovers and saves optimal config

Outline

- Bandwidths
- CUDA
  - Overview
  - Development process
  - Performance Optimization
  - Syntax

- Most slides courtesy Massimiliano Fatica (NVIDIA)

CUDA Optimization Priorities

- Memory coalescing is #1 priority
  - Highest !/$ optimization
  - Optimize for locality
- Take advantage of shared memory
  - Very high bandwidth
  - Threads can cooperate to save work
- Use parallelism efficiently
  - Keep the GPU busy at all times
  - High arithmetic / bandwidth ratio
  - Many threads & thread blocks
- Leave bank conflicts and divergence for last!
  - 4-way and smaller conflicts are not usually worth avoiding if avoiding them will cost more instructions

© NVIDIA Corp.

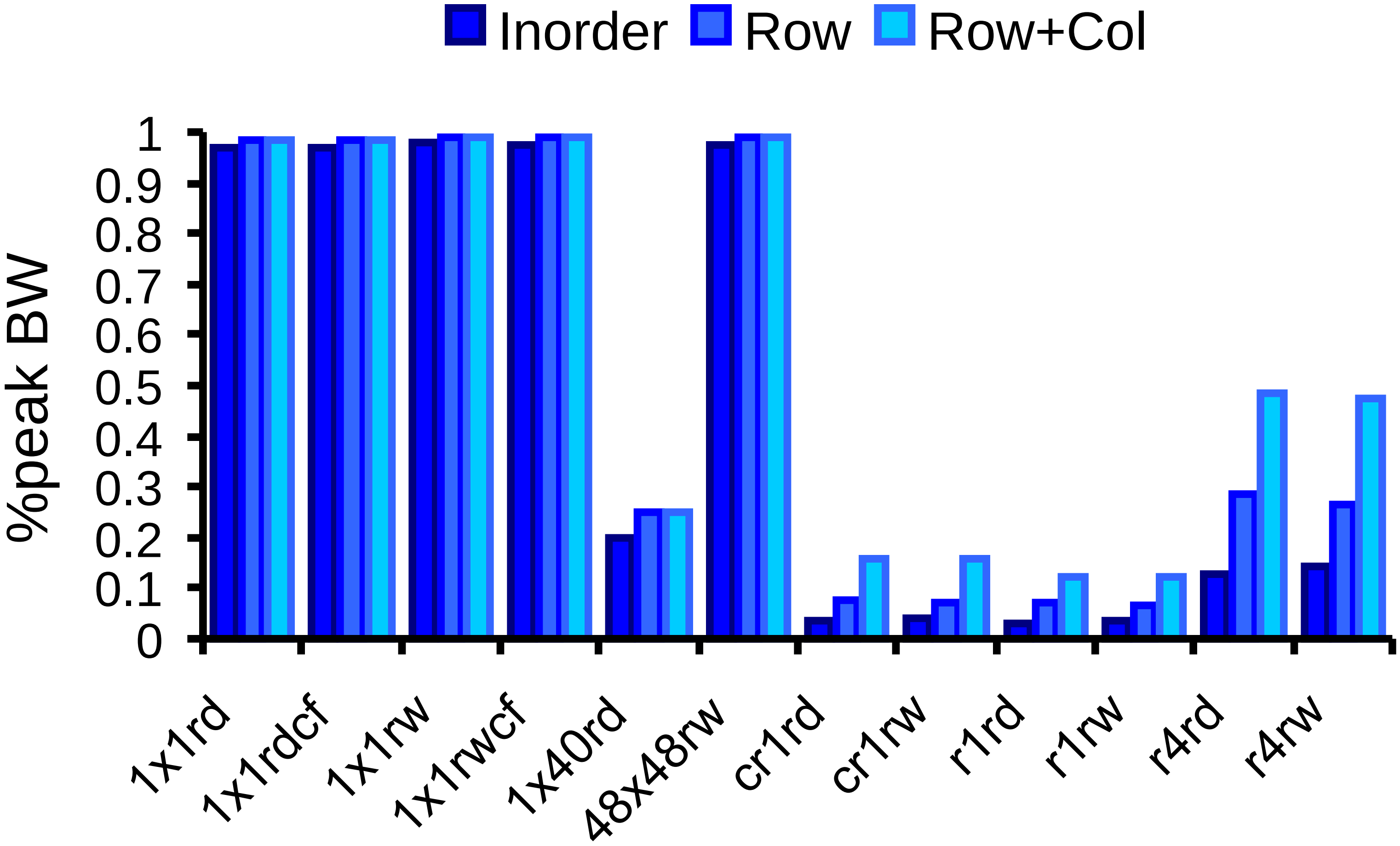
CUDA Optimization Strategies

- Optimize Algorithms for the GPU
- Optimize Memory Access Pattern
- Take Advantage of On-Chip Shared Memory
  - Watch out for bank conflicts – each serialized bank conflict costs 2 cycles!
- Use Parallelism Efficiently
  - Divergence is bad, but not as bad as poor mem usage
- Use appropriate mechanisms

Optimize Algorithms for the GPU

- Maximize independent parallelism
- Maximize arithmetic intensity (math/bandwidth)
- Sometimes it's better to recompute than to cache
  - GPU spends its transistors on ALUs, not memory
- Do more computation on the GPU to avoid costly data transfers
  - Even low parallelism computations can sometimes be faster than transfering back and forth to host

Modern DRAMs are Sensitive to Pattern



Optimize Memory Pattern ("Coherence")

- Coalesced vs. Non-coalesced = order of magnitude
  - Global/Local device memory
  - Sequential access by threads in a half-warp get coalesced
- Fermi's caches help a lot
  - Simplify coalescing and provide more buffering
- Optimize for spatial locality in cached texture memory
- Constant memory provides broadcast within SM
- In shared memory, avoid high-degree bank conflicts

Take Advantage of Shared Memory

- Hundreds of times faster than global memory
- Threads can cooperate via shared memory

- Use one / a few threads to load / compute data shared by all threads

- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing
  - See the transpose SDK sample for an example

Use Parallelism Efficiently

- Partition your computation to keep the GPU multiprocessors equally busy
  - Many threads, many thread blocks
- Keep resource usage low enough to support multiple active thread blocks per multiprocessor
  - Registers, shared memory

Maximizing Instruction Throughput

- Minimize use of low-throughput instructions
- Maximize use of high-bandwidth memory
  - Maximize use of shared memory
  - Maximize coherence of cached accesses
  - Minimize accesses to (uncached) global and local memory
  - Maximize coalescing of global memory accesses
- Optimize performance by overlapping memory accesses with HW computation
  - High arithmetic intensity programs
    - i.e. high ratio of math to memory transactions
  - Many concurrent threads

Data Transfers

- Device memory to host memory bandwidth much lower than device memory to device bandwidth
  - 4GB/s peak (PCI-e x16) vs. 80 GB/s peak (Quadro FX 5600)
- Minimize transfers
  - Intermediate data structures can be allocated, operated on, and deallocated without ever copying them to host memory
- Group transfers
  - One large transfer much better than many small ones

Page-Locked Memory Transfers

- cuMemAllocHost() allows allocation of page-locked host memory
- Enables highest cudaMemcpy performance
  - 3.2 GB/s common on PCI-e x16
  - ~4 GB/s measured on nForce 680i motherboards
- See the “bandwidthTest” CUDA SDK sample
- Use with caution
  - Allocating too much page-locked memory can reduce overall system performance
  - Test your systems and apps to learn their limits
- Memory allocation is also the time to control caching
  - Dynamic allocation possible, but interacts with driver so is slow.

Optimizing threads per block

- Given: total threads in a grid
  - Choose block size and number of blocks to maximize occupancy:

Occupancy: # of warps running concurrently on a multiprocessor divided by maximum # of warps that can run concurrently

(Demonstrate CUDA Occupancy Calculator)

Grid/Block Size Heuristics

- # of blocks / # of multiprocessors > 1
  - So all multiprocessors have at least a block to execute
- Per-block resources at most half of total available
  - Shared memory and registers
  - Multiple blocks can run concurrently in a multiprocessor
  - If multiple blocks coexist that aren't all waiting at a __syncthreads(), machine can stay busy
- # of blocks / # of multiprocessors > 2
  - So multiple blocks run concurrently in a multiprocessor
- # of blocks > 100 to scale to future devices
  - Blocks stream through machine in pipeline fashion
  - 1000 blocks per grid will scale across multiple generations

Occupancy != Performance

- Increasing occupancy does not necessarily increase performance

BUT…

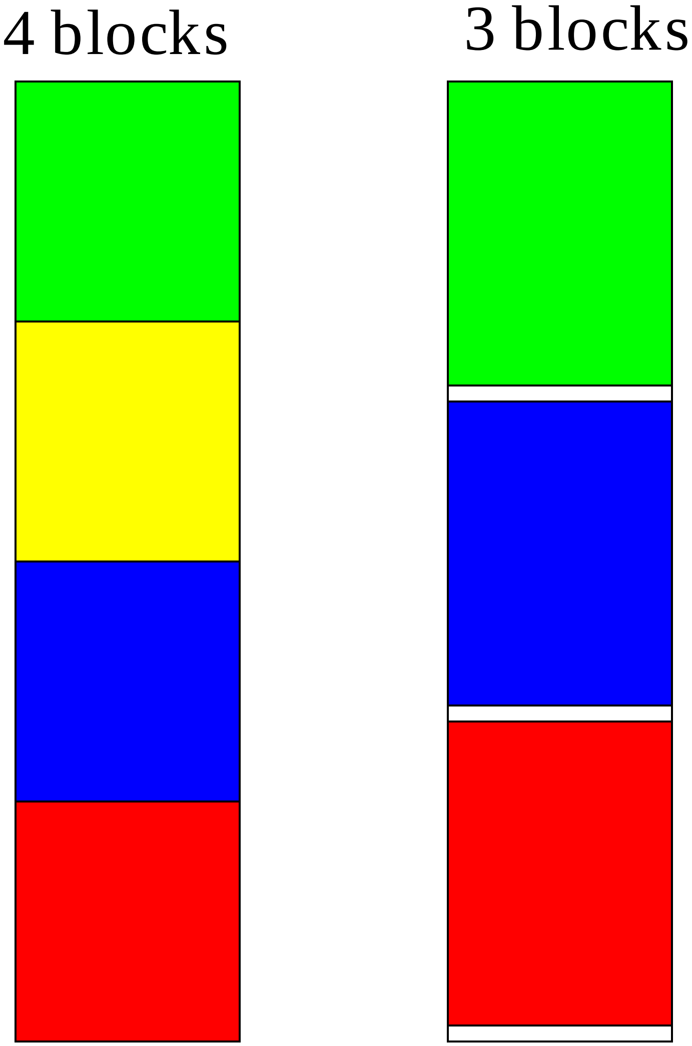
- Low-occupancy multiprocessors cannot adequately hide latency on memory-bound kernels
  - (It all comes down to arithmetic intensity and available parallelism)

Optimizing threads per block

- Choose threads per block as a multiple of warp size
  - Avoid wasting computation on under-populated warps
- More threads per block == better memory latency hiding
- But, more threads per block == fewer regs per thread
  - Kernel invocations can fail if too many registers are used
- Heuristics
  - Minimum: 64 threads per block
    - Only if multiple concurrent blocks
  - 192 or 256 threads a better choice
    - Usually still enough regs to compile and invoke successfully
  - This all depends on your computation!
    - Experiment!

Programmer View of Register File

- There are 8192 registers in each SM in G80
  - This is an implementation decision, not part of CUDA
  - Registers are dynamically partitioned across all Blocks assigned to the SM
  - Once assigned to a Block, the register is NOT accessible by threads in other Blocks
  - Each thread in the same nly access assigned to itself



Communication

- How do threads communicate?
- Remember the execution model:
  - Data parallel streams that represent independent vertices, triangles, fragments, and pixels in the graphics world
  - These ***never*** communicate
- Some communication allowed in compute mode:
  - Shared memory for threads in a thread block
    - No special communication within warp or using registers
  - No communication between thread blocks
  - Kernels communicate through global device memory
- **Mechanisms designed to ensure portability**

In-Kernel Synchronization

- Do threads need to synchronize?
  - Basically no communication allowed
- Threads in a block share memory – need sync
  - Warps scheduled OoO, can't rely on warp order
  - Barrier command for all threads in a block
  - __synchthreads()
- Blocks cannot synchronize
  - Implicit synchronization at end of kernel

Inter-Kernel Synchronization

- Synchronize across device
  - Kernels and memory transfers launched asynchronously!
  - Need to use appropriate synchronization for correctness
- By default, kernels and DMAs are asynchronous
  - Can even run concurrent kernels on Fermi
- Manage synchronization with streams and events
- Stream: an in-order sequence of bulk operations
  - Streams can be arbitrarily interleaved or executed concurrently
  - cudaStreamCreate(cudastream_t* st) / cudaStreamDestroy
  - cudaStreamQuery / cudaStreamSynchronize:
    - Check that / block until all preceding commands in the specified stream complete
  - cudaDeviceSynchronize:
    - Wait until all preceding commands in all streams complete
- Event explicit event that you can explicitly “record” or “wait” for
  - cudaEventCreate(cudaevent_t* e) / cudaEvenDestroy
  - cudaEventRecord / cudaEventSynchronize

Atomic Operations and consistency

- Exception to communication between blocks
- Atomic read-modify-write
  - Shared memory
  - Global memory
- Simple ALU operations
  - Add, subtract, AND, OR, min, max, inc, dec
- Exchange operations
  - Compare-and-swap, exchange

- Extremely relaxed memory consistency with fences
  - __threadfence_block(): global and shared visible to CTA
  - __threadfence(): shared to CTA, global across device
  - __threadfence_system(): also visible to host threads (for locked pages)