

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

Lecture 22 – CUDA

Mattan Erez



The University of Texas at Austin



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

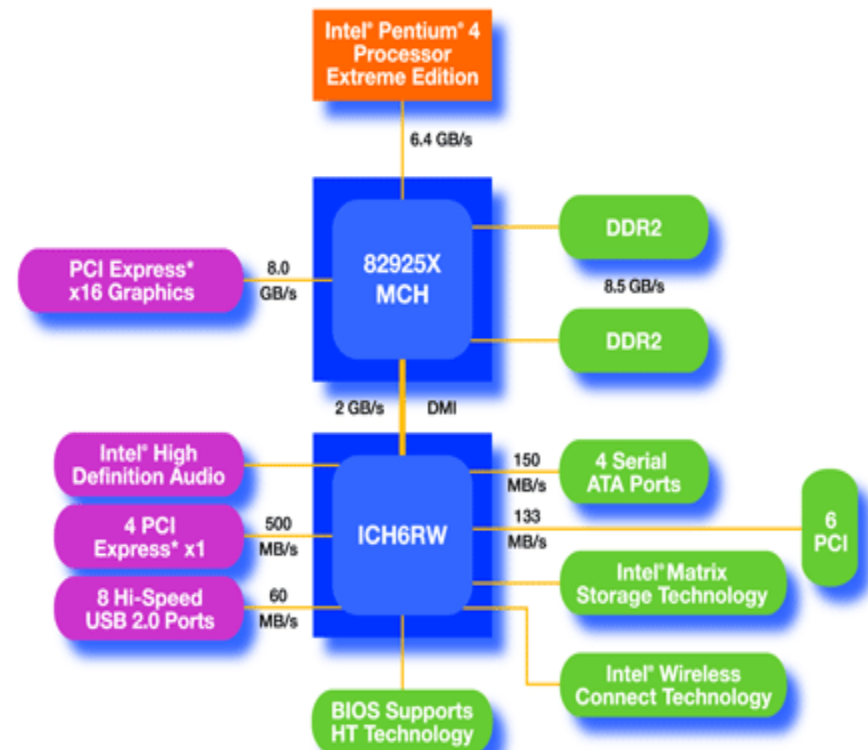
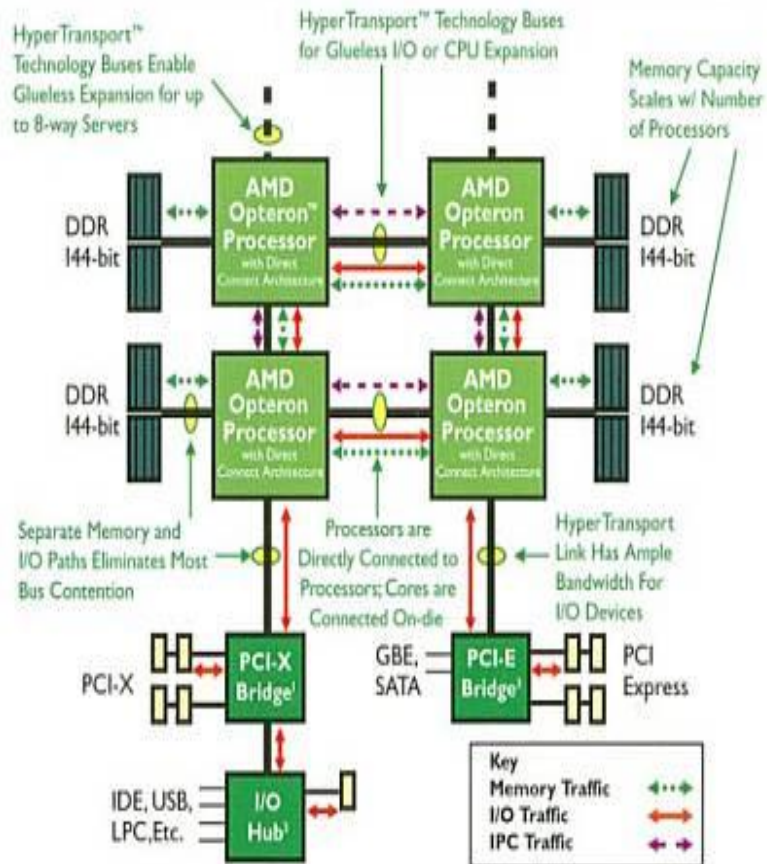
Specific parameters not exposed

The CUDA Platform

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

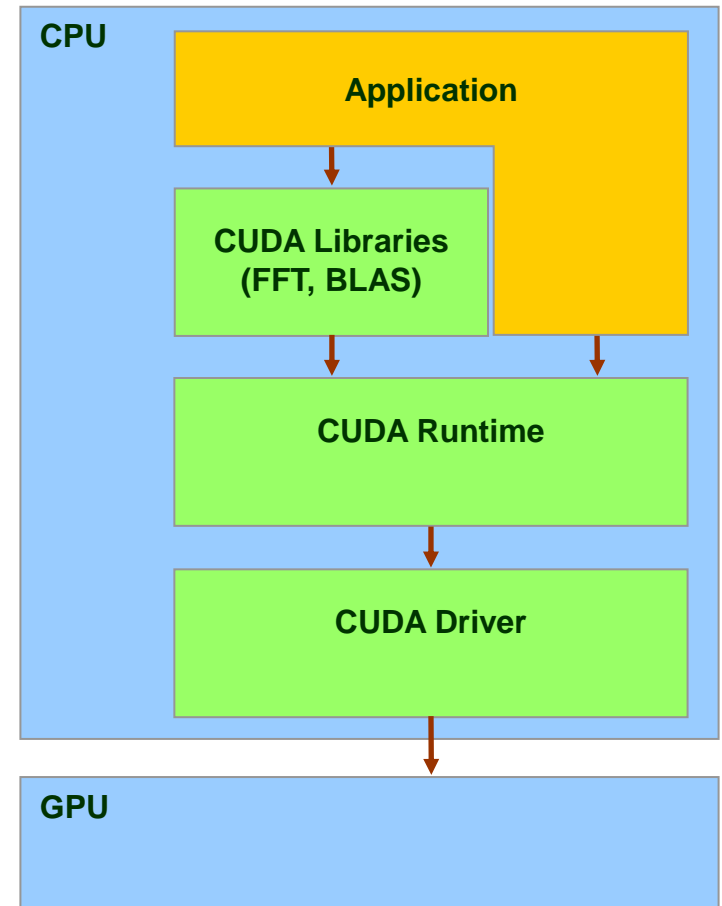


AMD Opteron™ Processor-based 4P Server

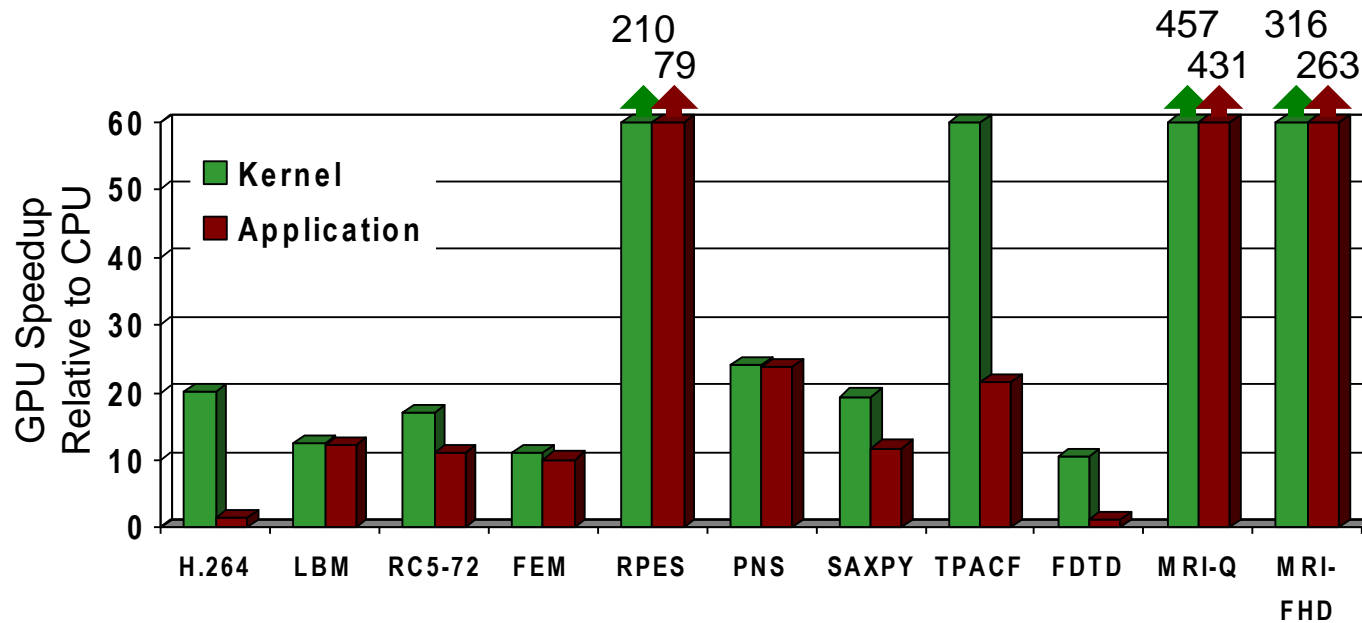


CUDA Programming System

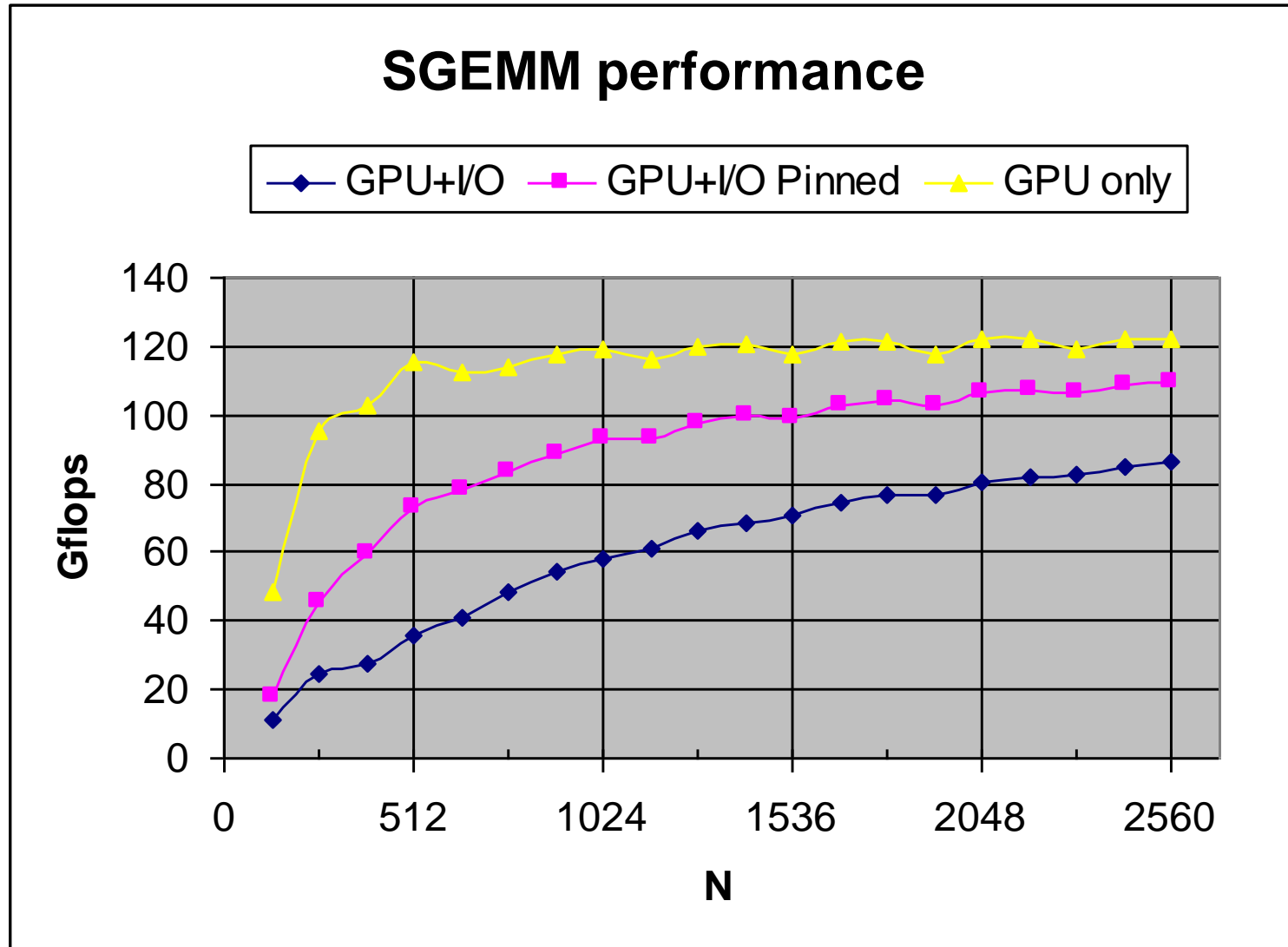
- Targeted software stack
 - Compute oriented drivers, language, and tools
- Driver for loading computation programs into GPU
 - Standalone Driver - Optimized for computation
 - Interface designed for compute - graphics free API
 - Data sharing with OpenGL buffer objects
 - Guaranteed maximum download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



Overall Performance Can be Limited by Interface



Overall Performance Can be Limited by Interface



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

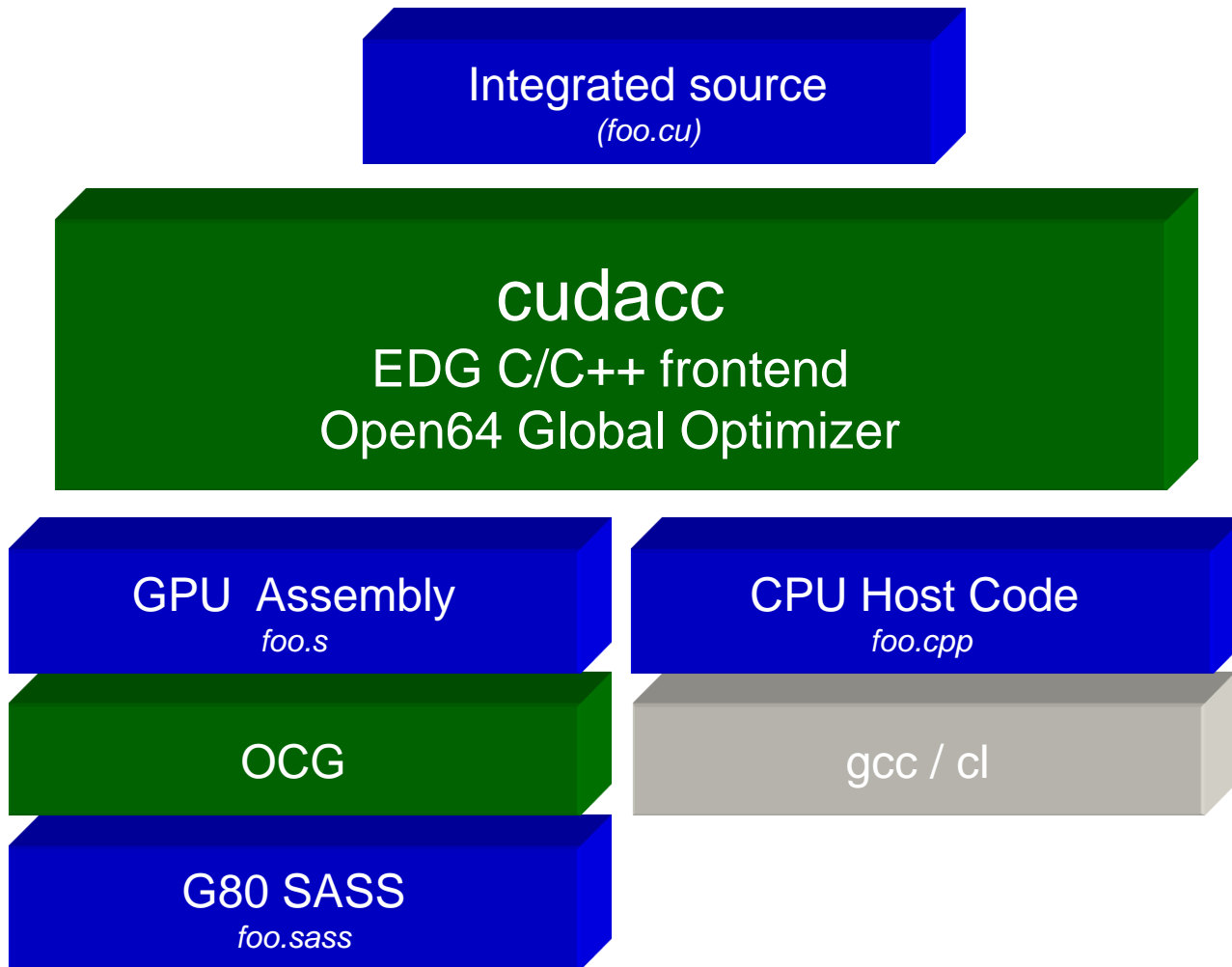


CUDA Programming Model: A Highly Multithreaded Coprocessor

- 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

CUDA is an Extension to C++



CUDA is an Extension to C++

- Declspecs

- global, device, shared, local, constant

```
__device__ float filter[N];
```

- Keywords

- threadIdx, blockIdx

```
__global__ void convolve (float *image) {
```

```
    __shared__ float region[M];
```

```
    ...
```

```
    region[threadIdx] = image[i];
```

- Intrinsic

- __syncthreads
- __threadfence

```
    __syncthreads()
```

```
    ...
```

```
    image[j] = result;
```

```
}
```

- Runtime API

- Memory, symbol, execution management

```
// Allocate GPU memory
```

```
void *myimage = cudaMalloc(bytes)
```

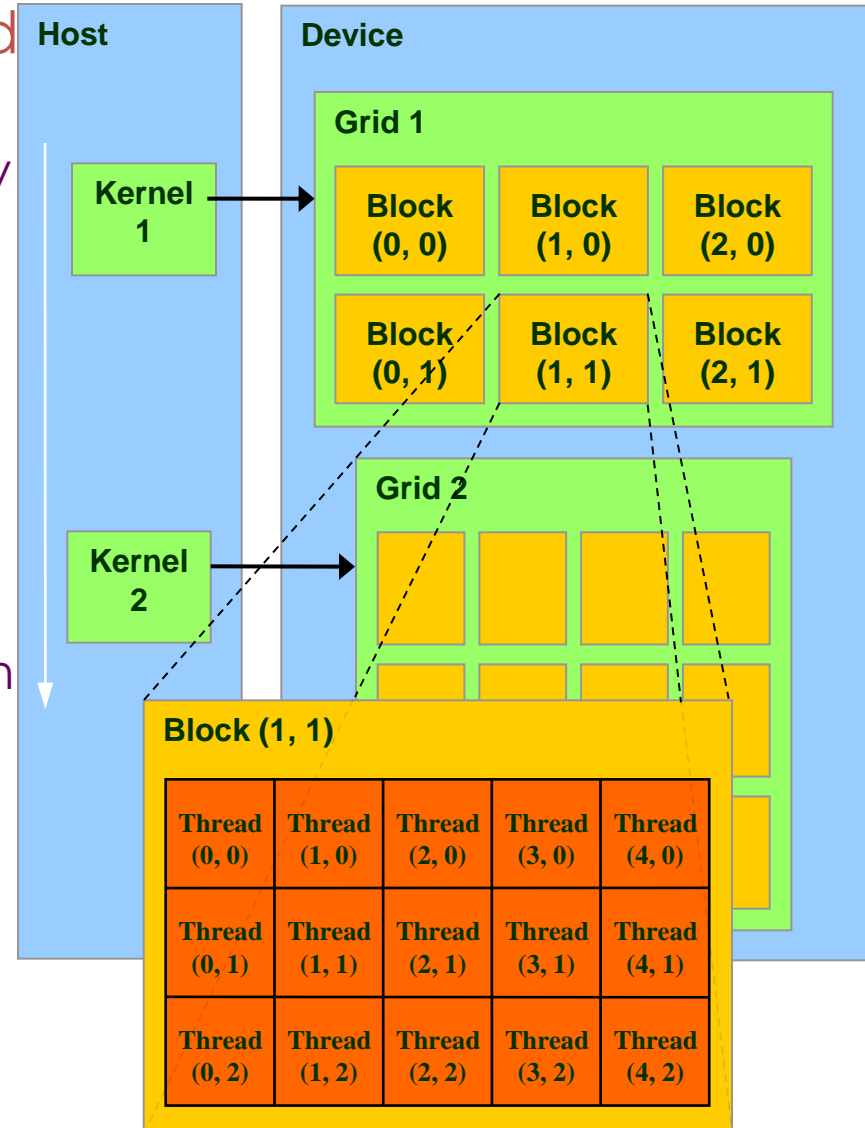
- Function launch

```
// 100 blocks, 10 threads per block
```

```
convolve<<<100, 10>>> (myimage);
```

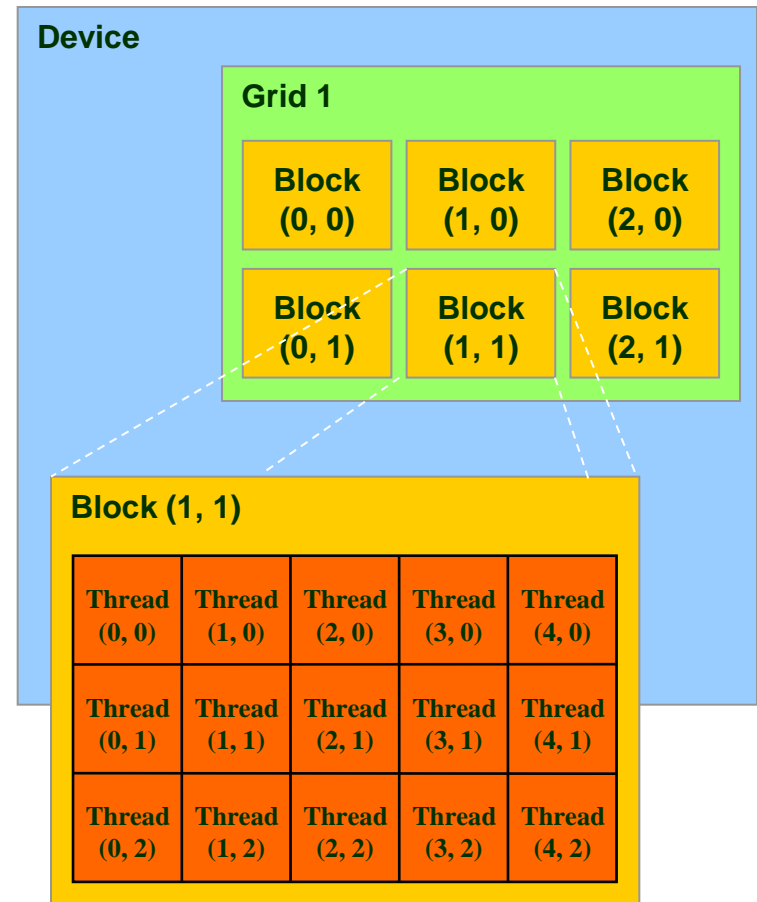
Thread Batching: Grids and Blocks

- 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



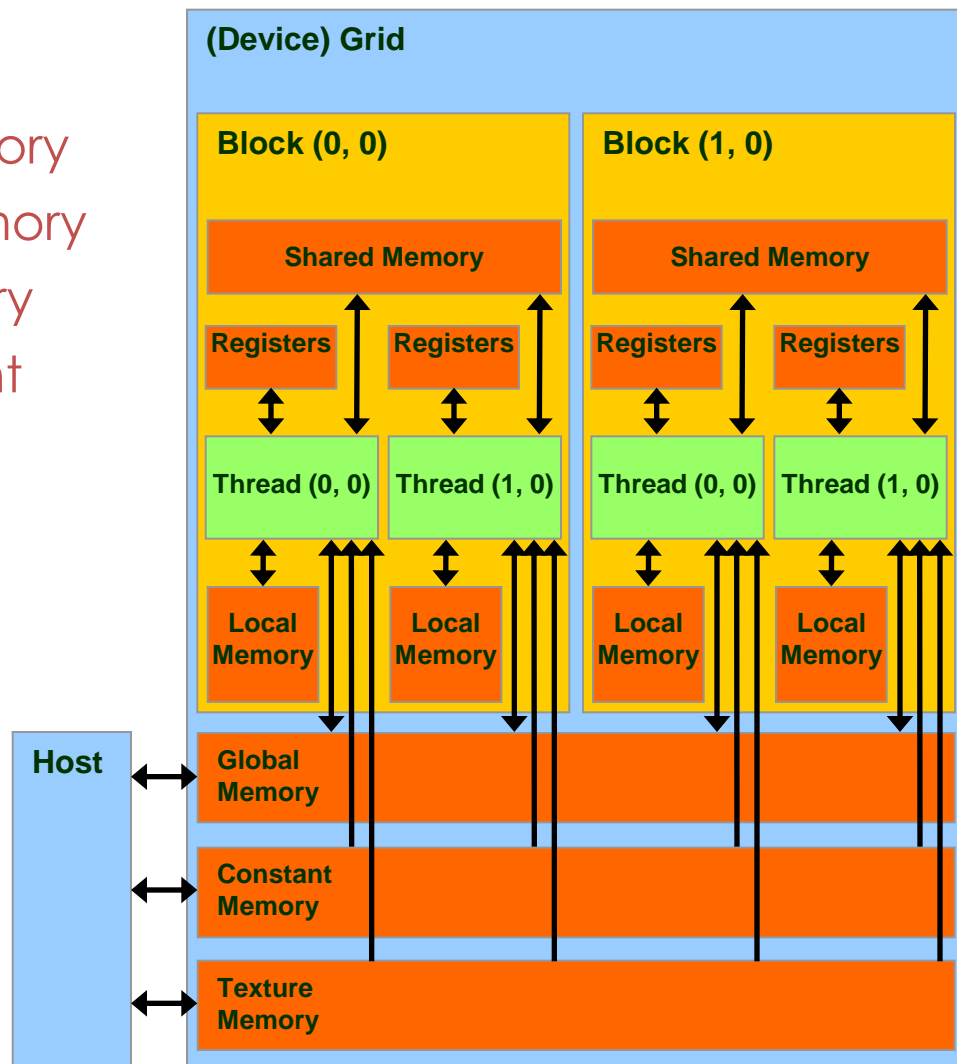
Block and Thread IDs

- 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
 - ...



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



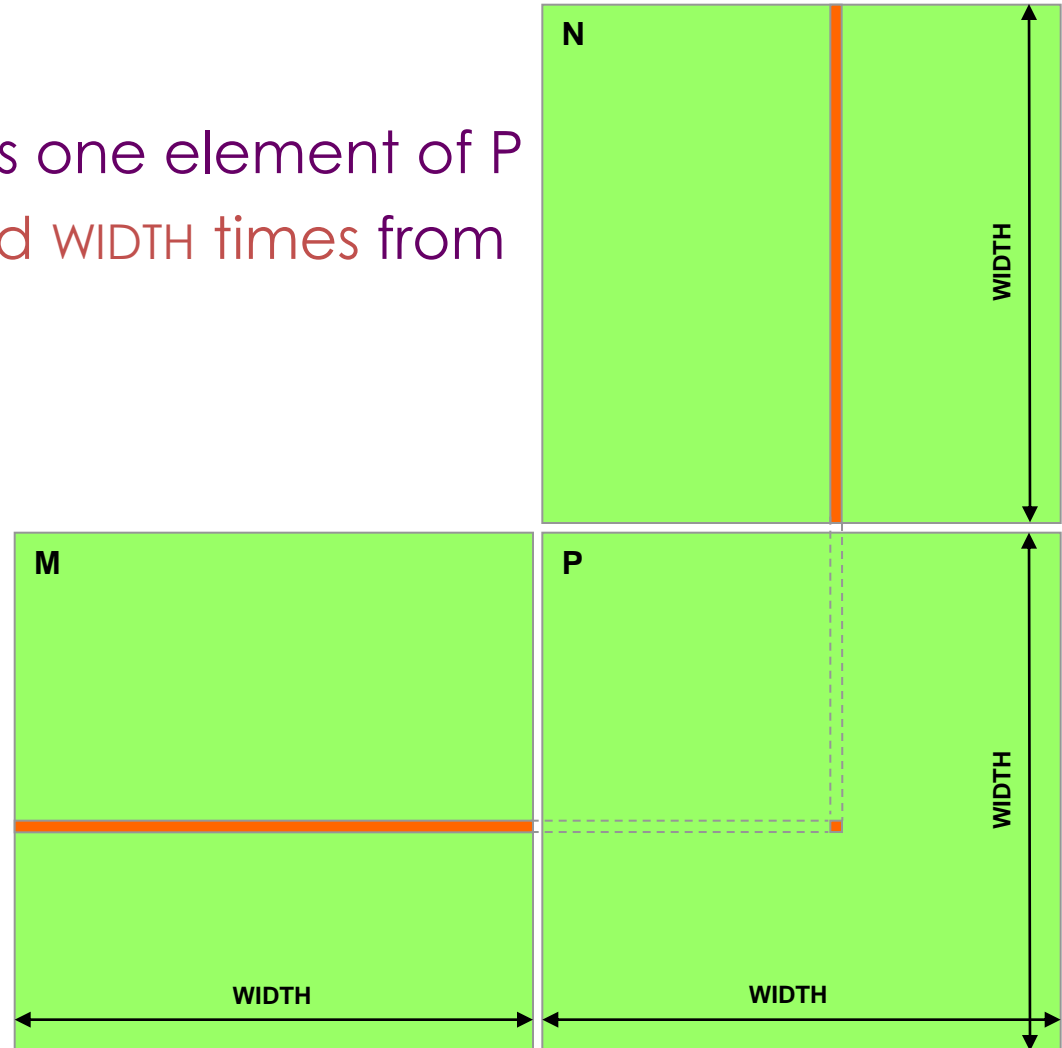
Access Times

- 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

(

Programming Model: Square Matrix Multiplication Example

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



Programming Model:

Common Programming Pattern

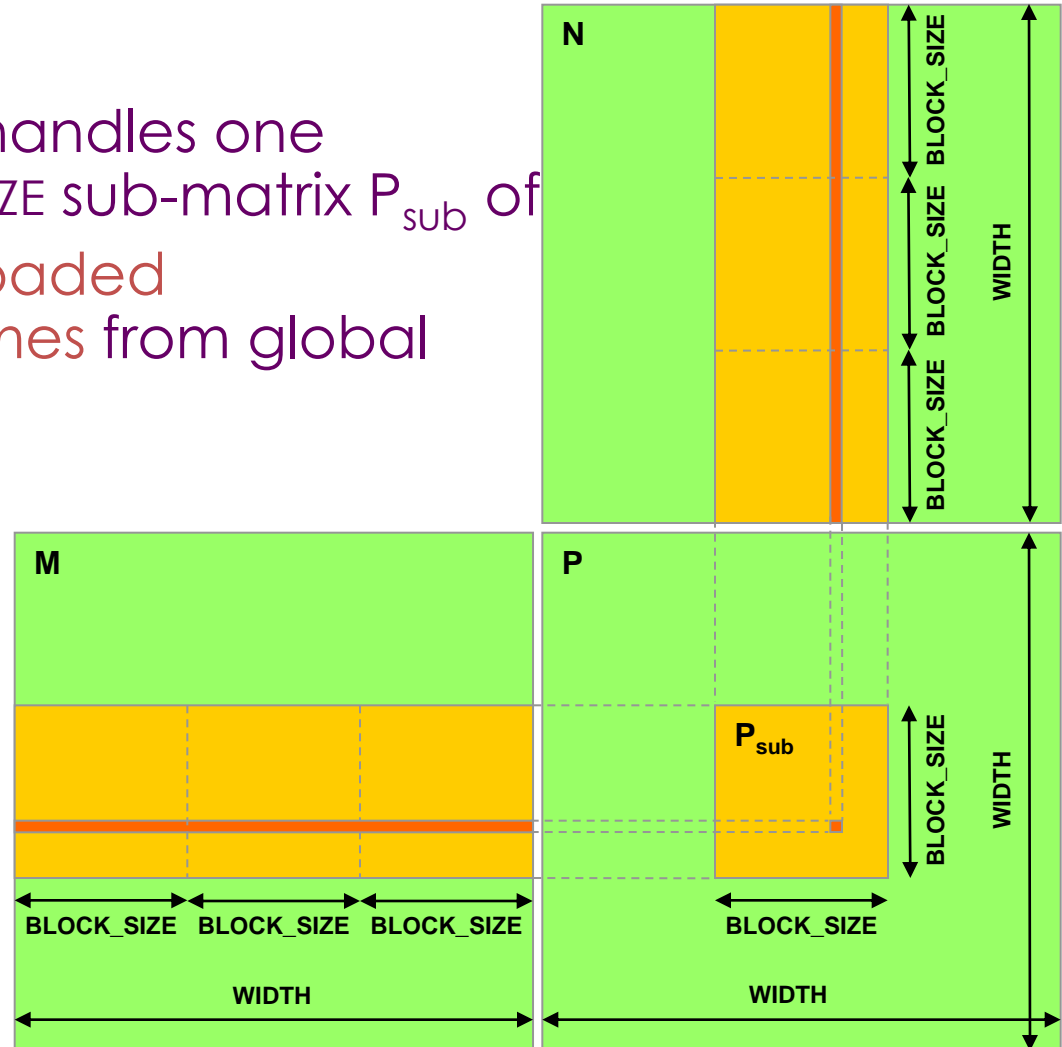
- Local and global memory reside in device memory (DRAM) - much slower access than shared memory
 - Uncached
- So, a common way of scheduling some computation on the device is to **block it up** to take advantage of fast shared memory:
 - Partition the data set into data subsets that fit into shared 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



Programming Model: Square Matrix Multiplication Example

- $P = M * N$ of size $WIDTH \times WIDTH$
- With blocking:
 - One **thread block** handles one $BLOCK_SIZE \times BLOCK_SIZE$ sub-matrix P_{sub} of
 - 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()`, ...



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 *
    blockIdx.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));
// ... initialize 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)

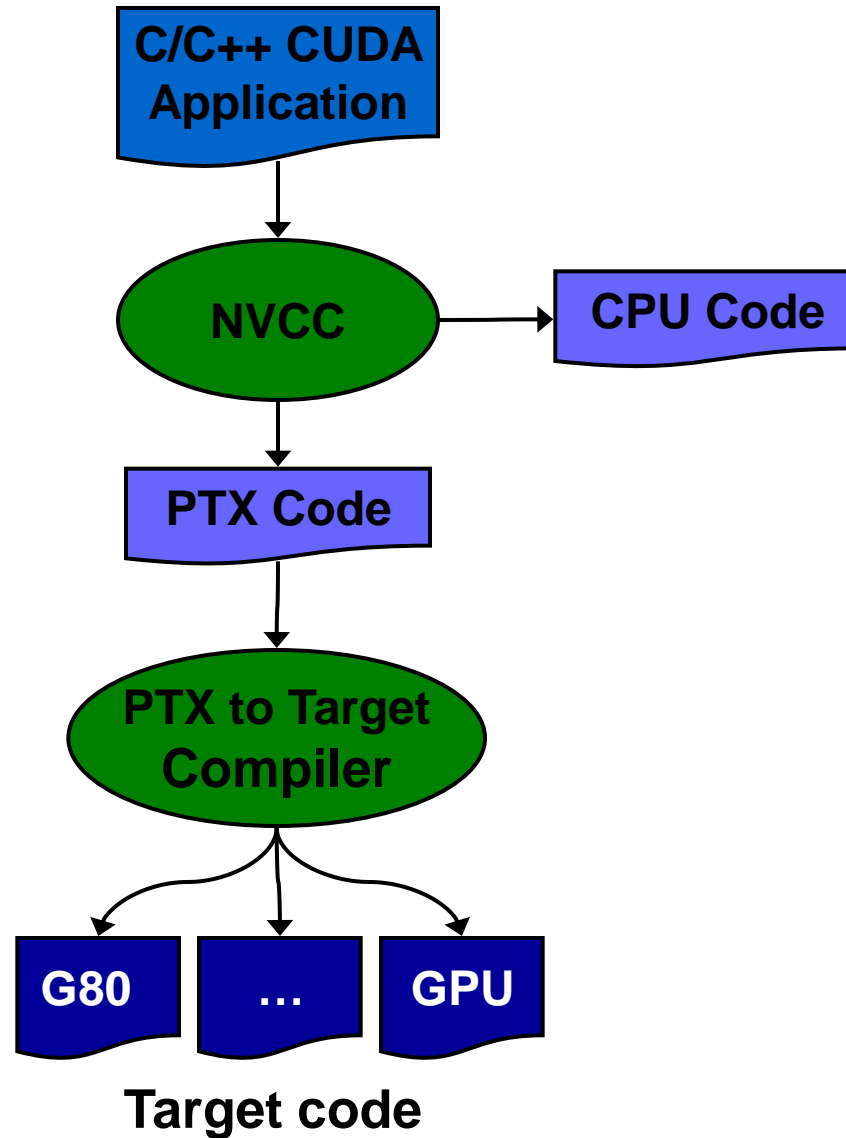


Compilation

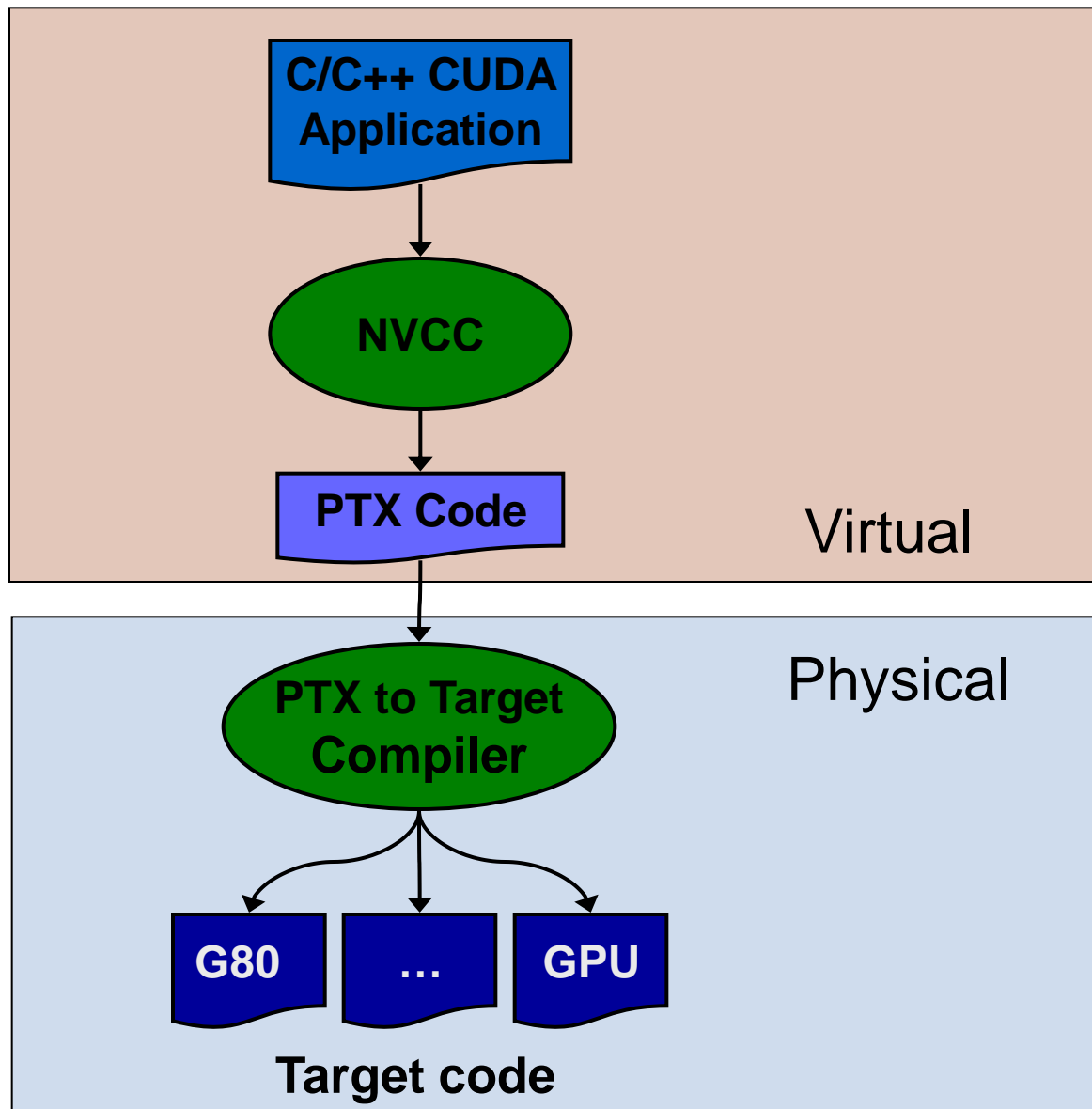
- Any source file containing CUDA language extensions must be compiled with `nvcc`
- NVCC is a `compiler driver`
 - Works by invoking all the necessary tools and compilers like `cudacc`, `g++`, `cl`, ...
- NVCC can output:
 - Either C code (CPU Code)
 - That must then be compiled with the rest of the 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`)



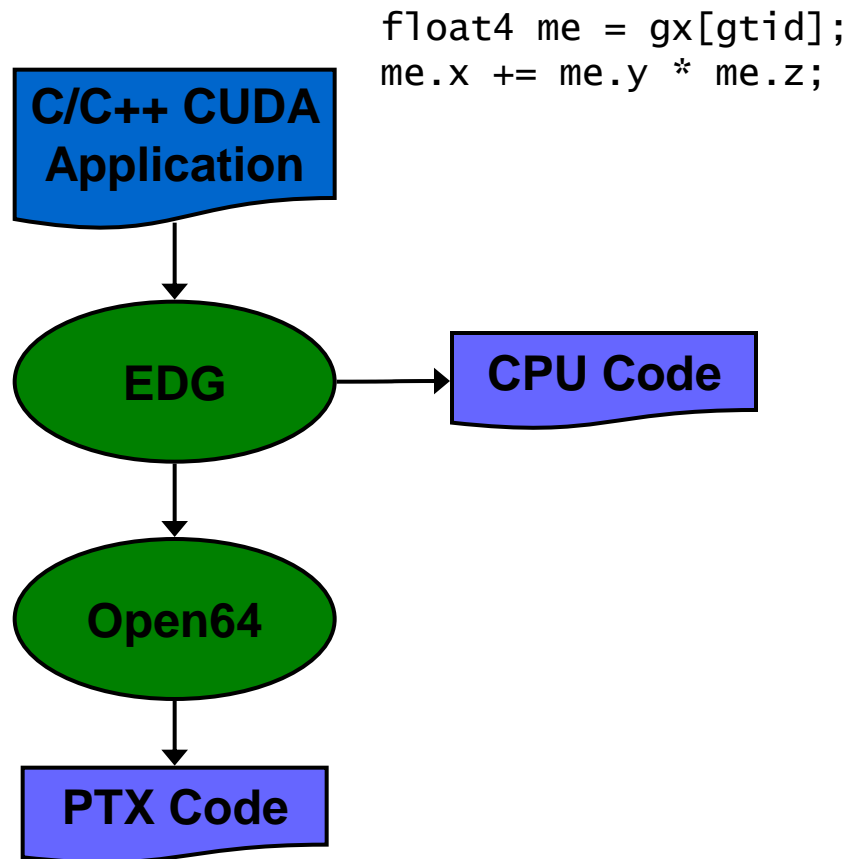
Compiling CUDA



Compiling CUDA



NVCC & PTX Virtual Machine



- 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;
```

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



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



CUDA Programming Tools

- CUDA GDB
 - A real debugger for both CPU and GPU CUDA parts
- CUDA MEMCHECK
 - Invalid memory access detection
- CUDA Visual Profiler
 - Useful performance information
- Integration with Microsoft Visual C++
 - Not all that helpful with our setup



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



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

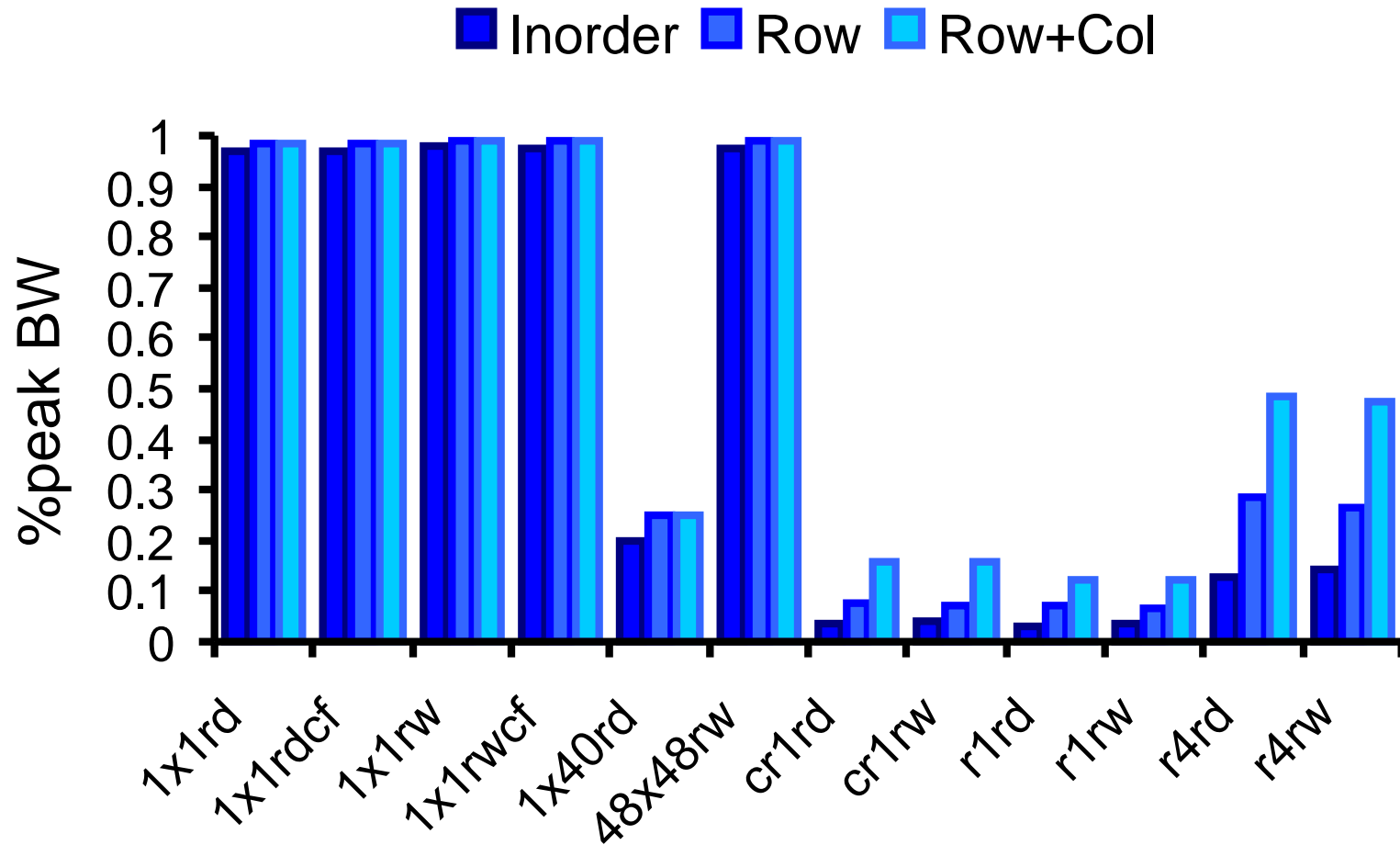


Optimize Algorithms for the GPU

- 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 transferring 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

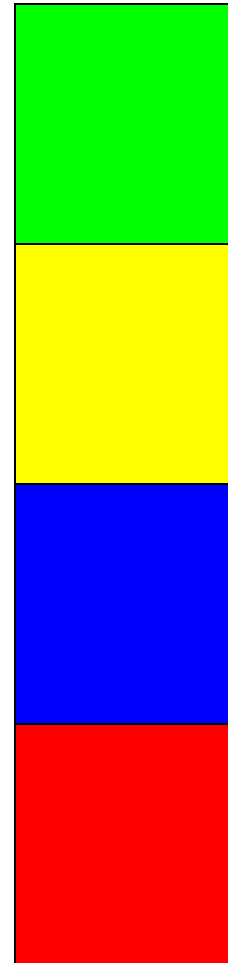
- Choose threads per block as a multiple of warp size
 - Avoid wasting computation on under-populated warps
- More threads per block == better memory latency hiding
- But, more threads per block == fewer regs per thread
 - Kernel invocations can fail if too many registers are used
- Heuristics
 - Minimum: 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!



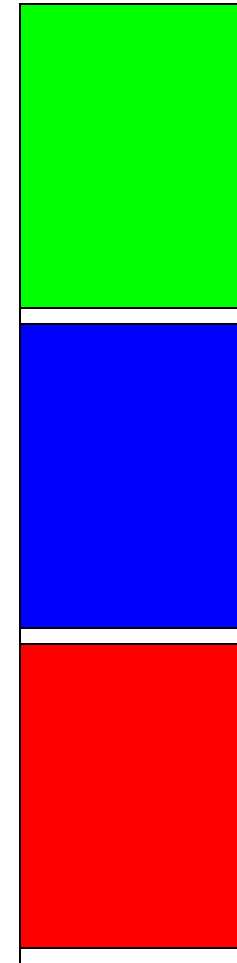
Programmer View of Register File

- 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

4 blocks



3 blocks



ily 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
 - `__syncthreads()`
- 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) / cudaEventDestroy`
 - `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)



Multiple Devices

- CUDA supports multiple devices on same platform
 - Multiple devices per host
- CUDA host thread selects active device
- Sends streams to a device
- Streams and events must be kept consistent
 - Can't record an event of one stream in another
- Fermi-based Tesla cards can access each others' memory
 - Direct device-to-device memory access
 - Pointer dereferencing
 - DMAs



Alternatives to CUDA

- For NVIDIA GPUs
 - Thrust: STL-like interface on top of CUDA (templates)
 - Copperhead: Python-like language that translates to CUDA
 - Phalanx: current research
- Enhancements
 - Cudadma
- OpenCL
 - CUDA's competition
 - More generally designed for heterogeneous systems
 - Adopts GPU (CUDA) execution model
 - Different syntax, names, and design choices

