EE382V: Principles in Computer Architecture
Parallelism and Locality
Fall 2008

Lecture 16 – 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 the G80 processor for compute
  - CUDA follows the architecture very closely

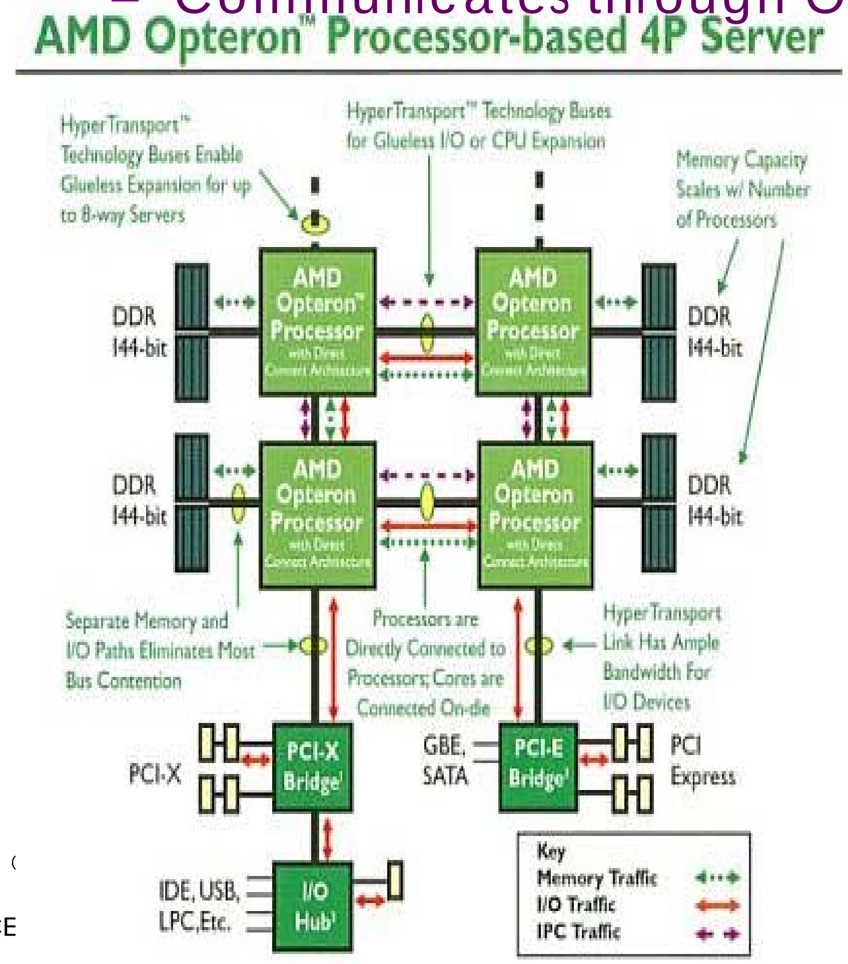
- General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor

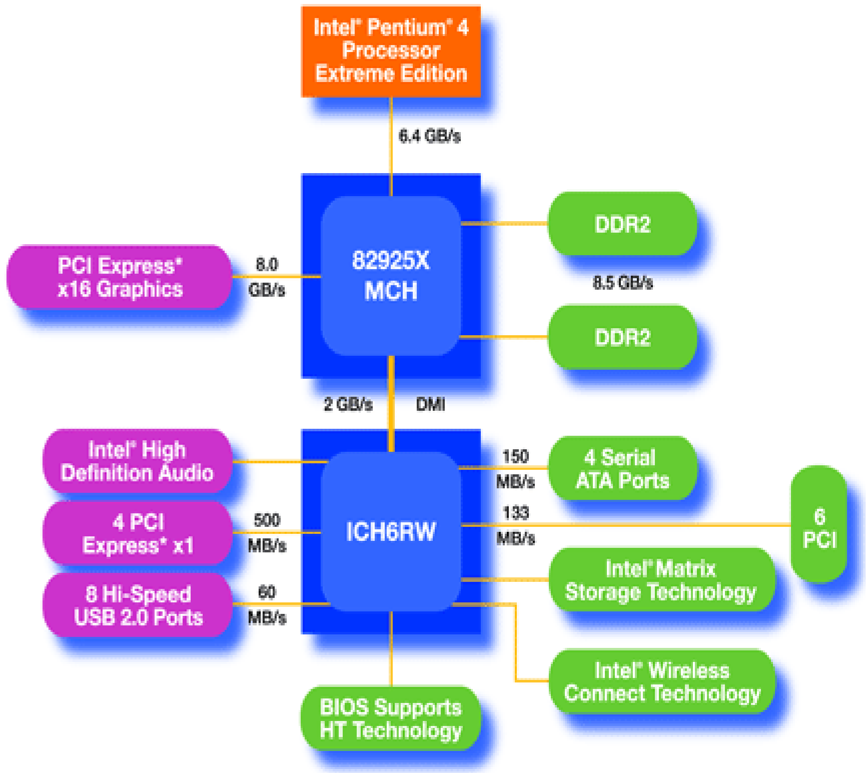
Matches architecture features

Specific parameters not exposed

The CUDA Platform

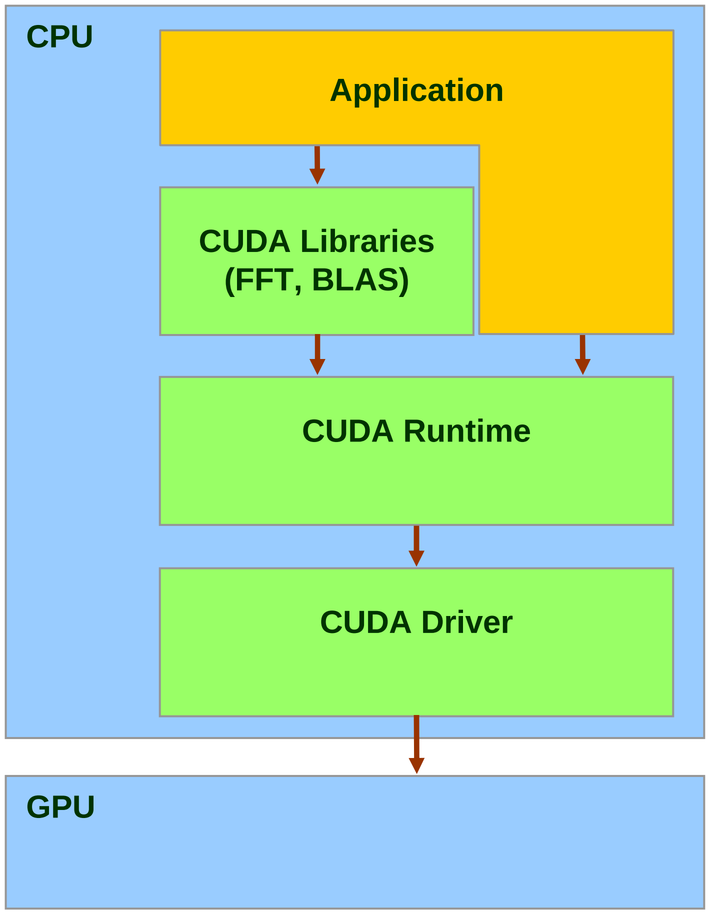
- The G80 GPU is not integrated into the CPU
  - Connects through a PCI Express bus
  - Communicates through OS (drivers)


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

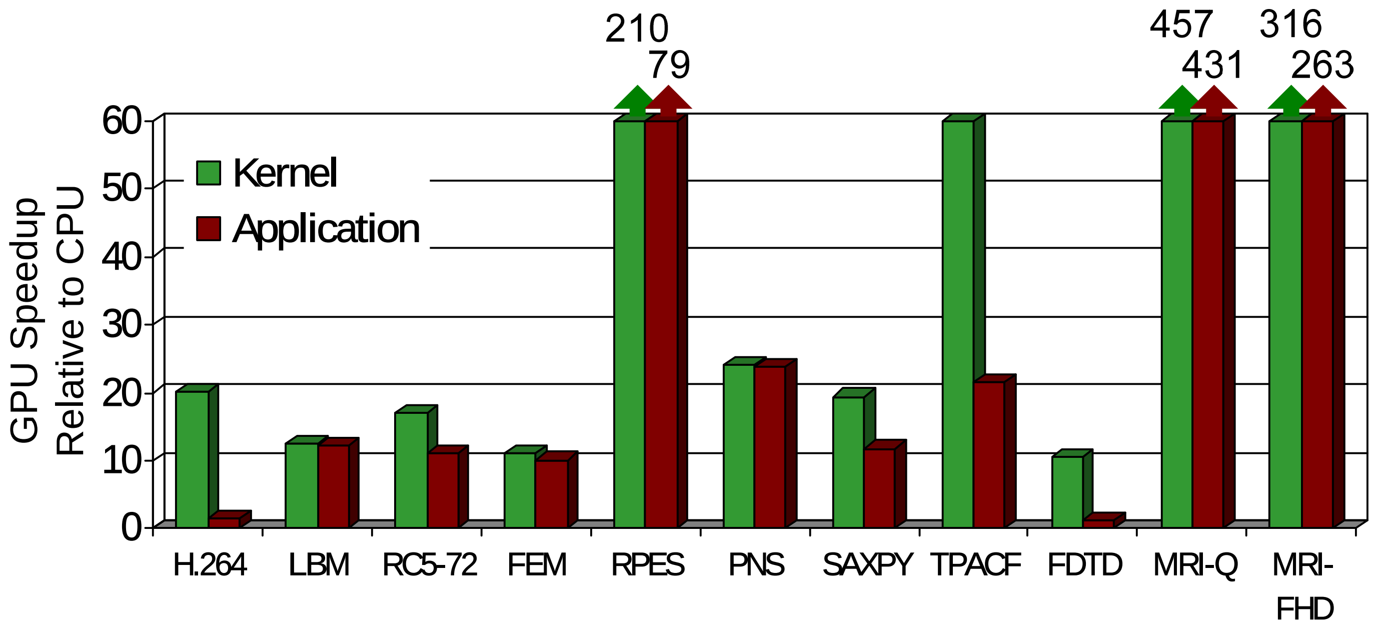
CUDA Programming System

- 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

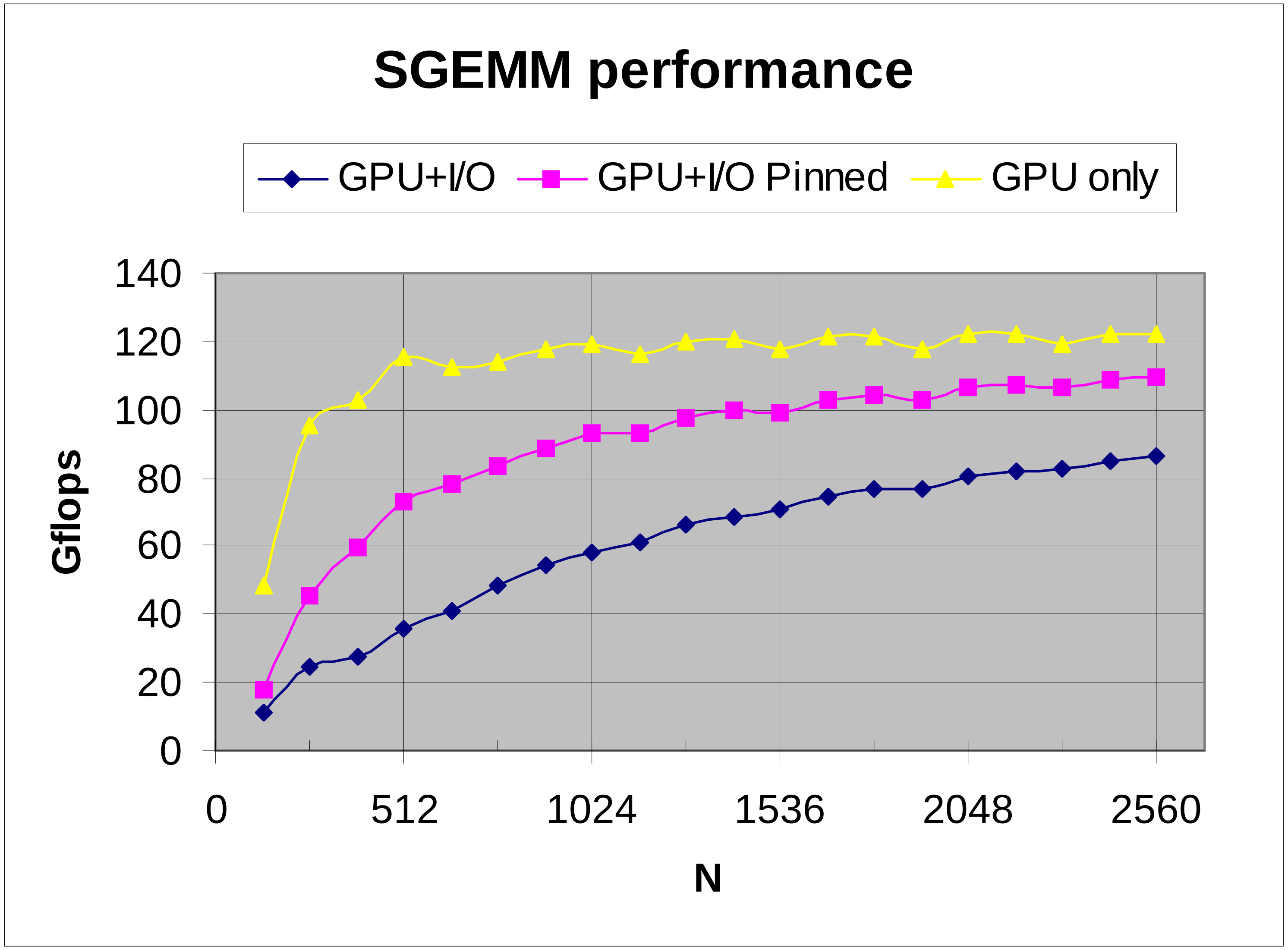


© NVIDIA Corp.

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

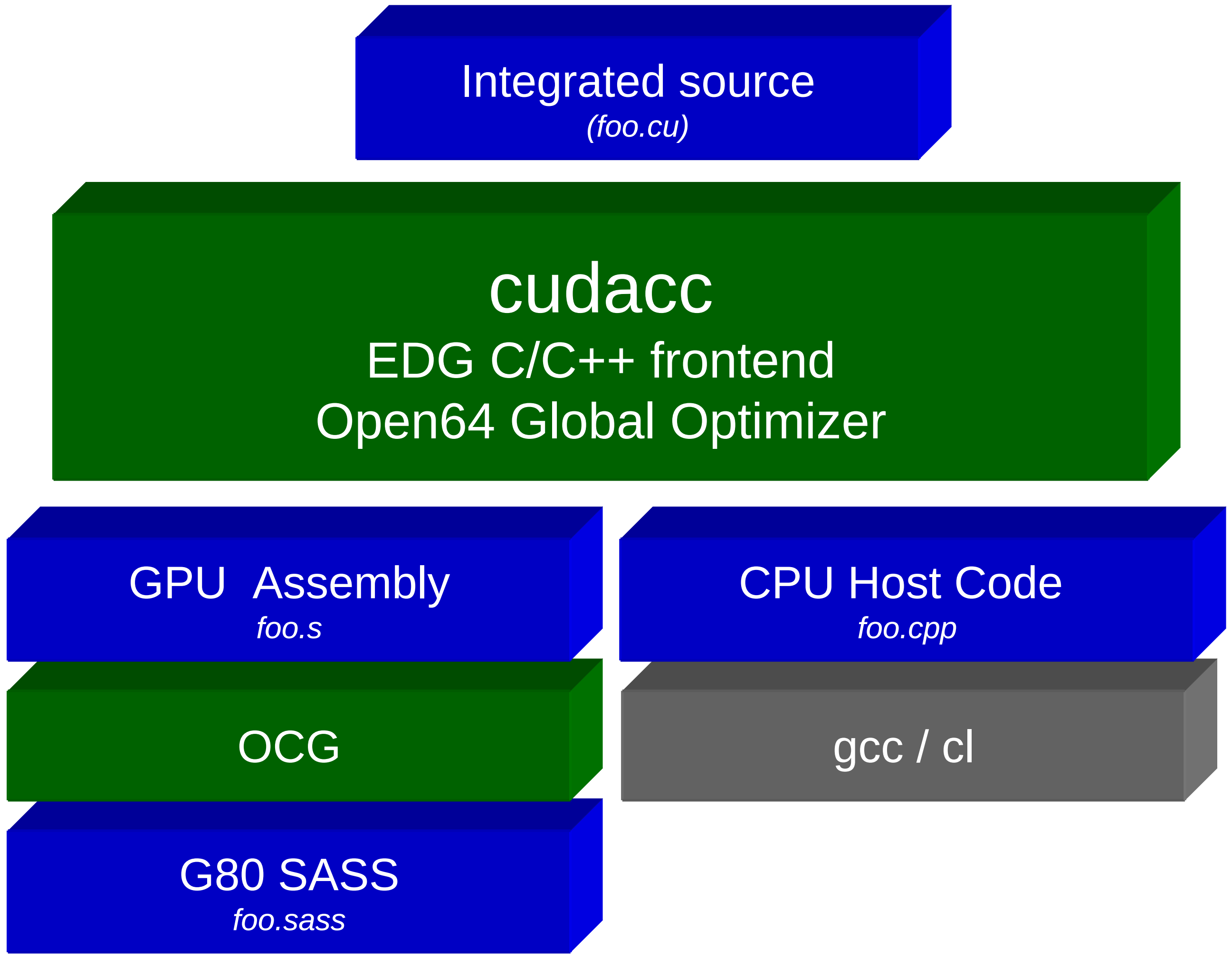
- The hardware is designed to enable lightweight runtime and driver

  → 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
    - Multi-core CPU needs only a few

CUDA is an Extension to C



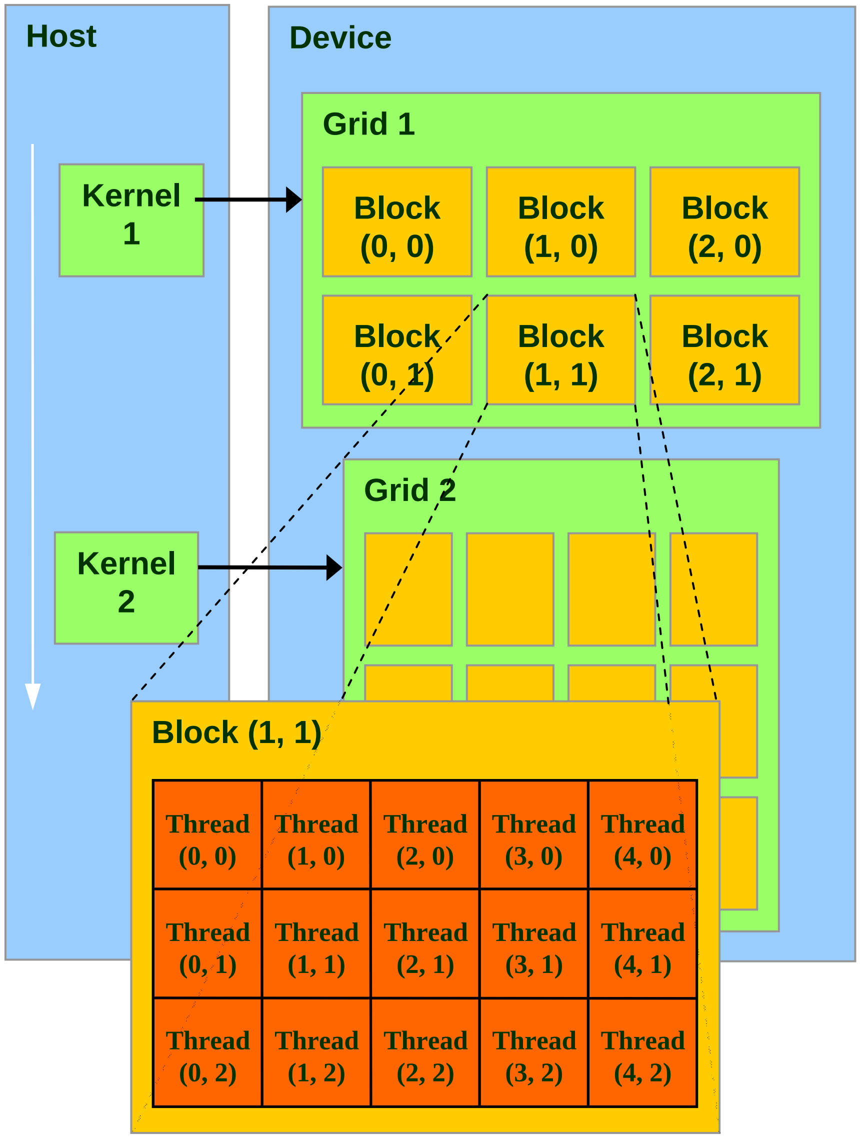
CUDA is an Extension to C

- Declspecs
  - global, device, shared, local, constant
- Keywords
  - threadIdx, blockIdx
- Intrinsics
  - __syncthreads
- 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);
```

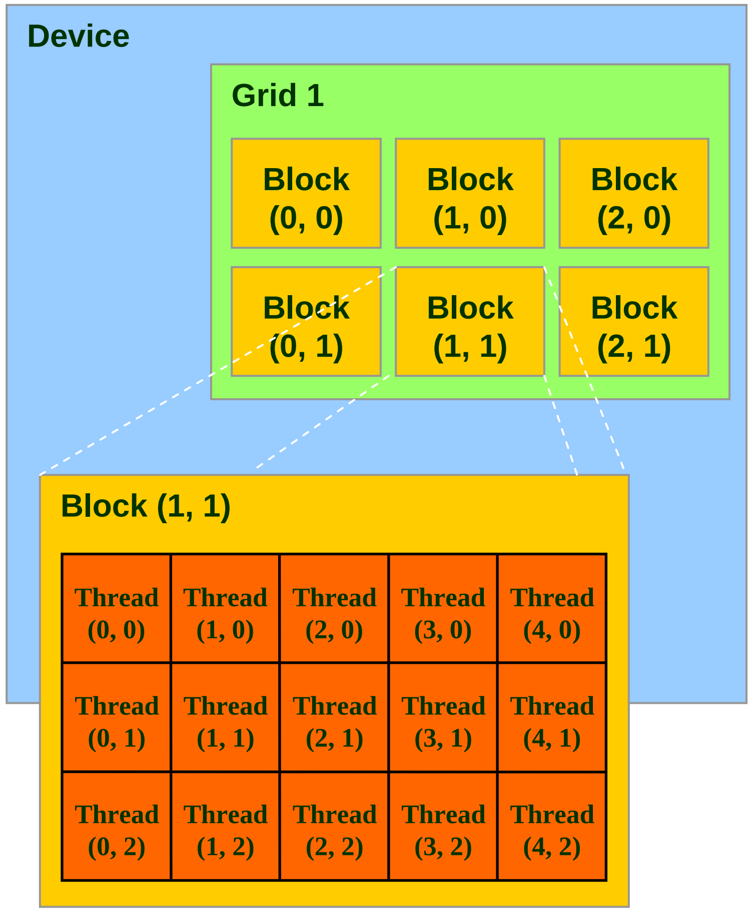
Thread Batching: Grids and Blocks

- 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



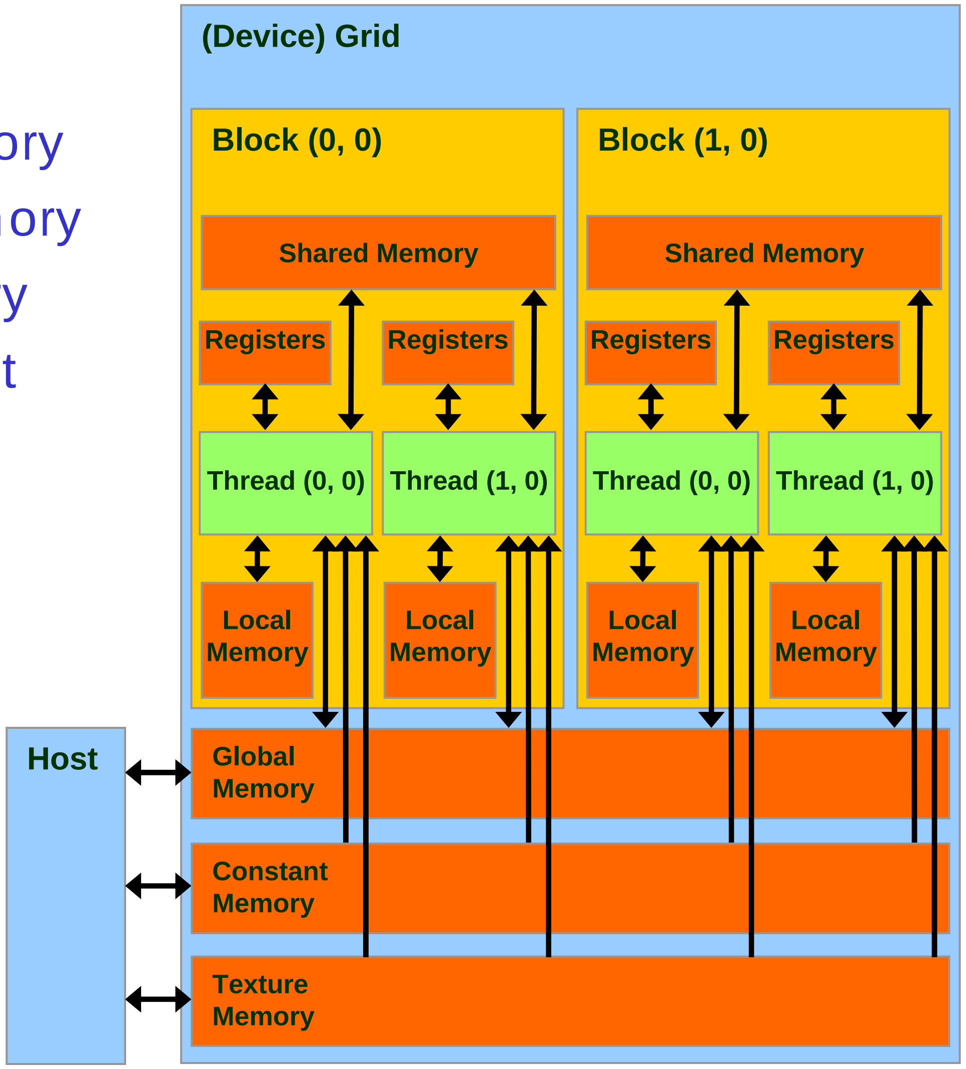
Block and Thread IDs

- 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
  - …



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

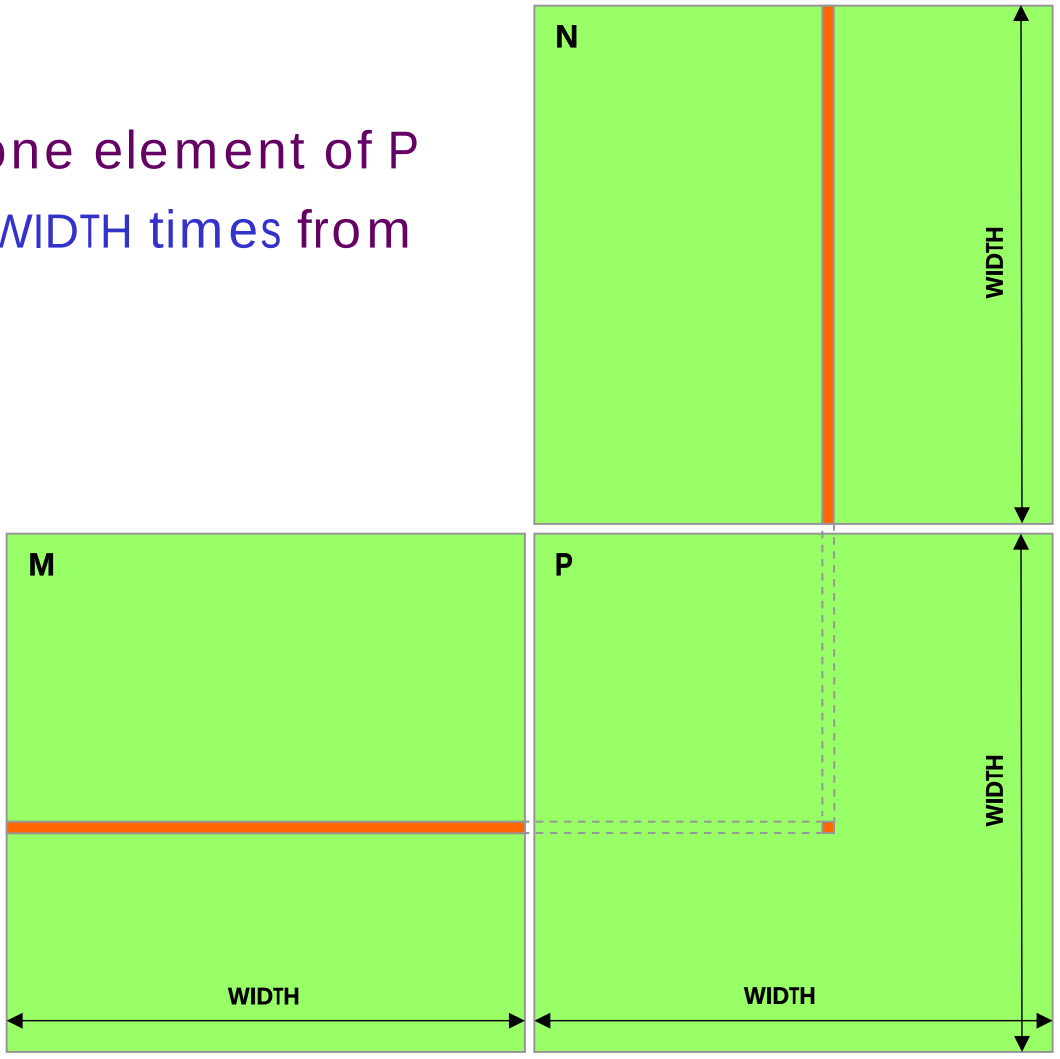


Access Times

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

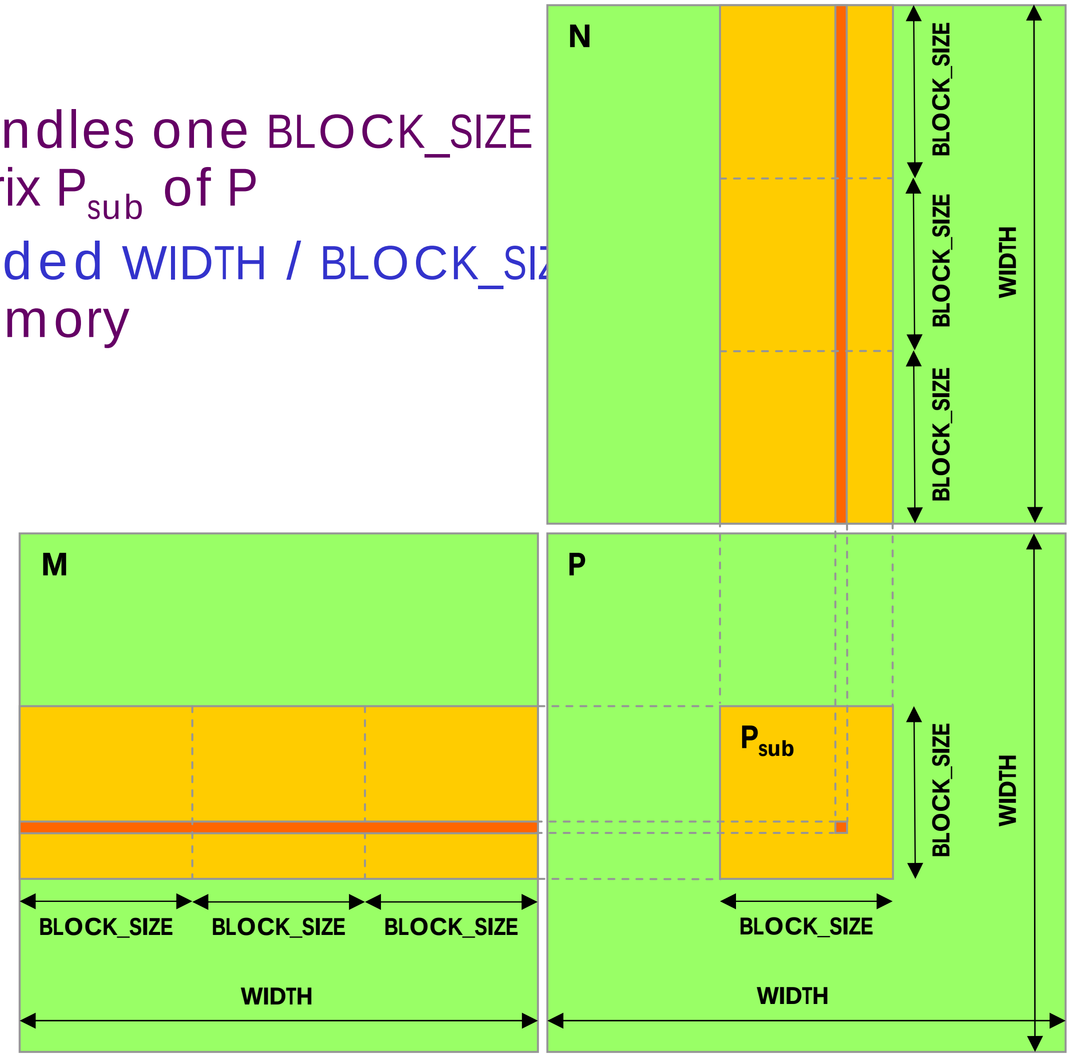


Programming Model: Common Programming Pattern

- 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 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_SIZ 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 = group of SIMD threads that execute a kernel and can communicate via shared memory

Memory	Location	Cached	Access	Who
Local	Off-chip	No	Read/write	One thread
Shared	On-chip	N/A	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

© NVIDIA Corp.

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 * 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

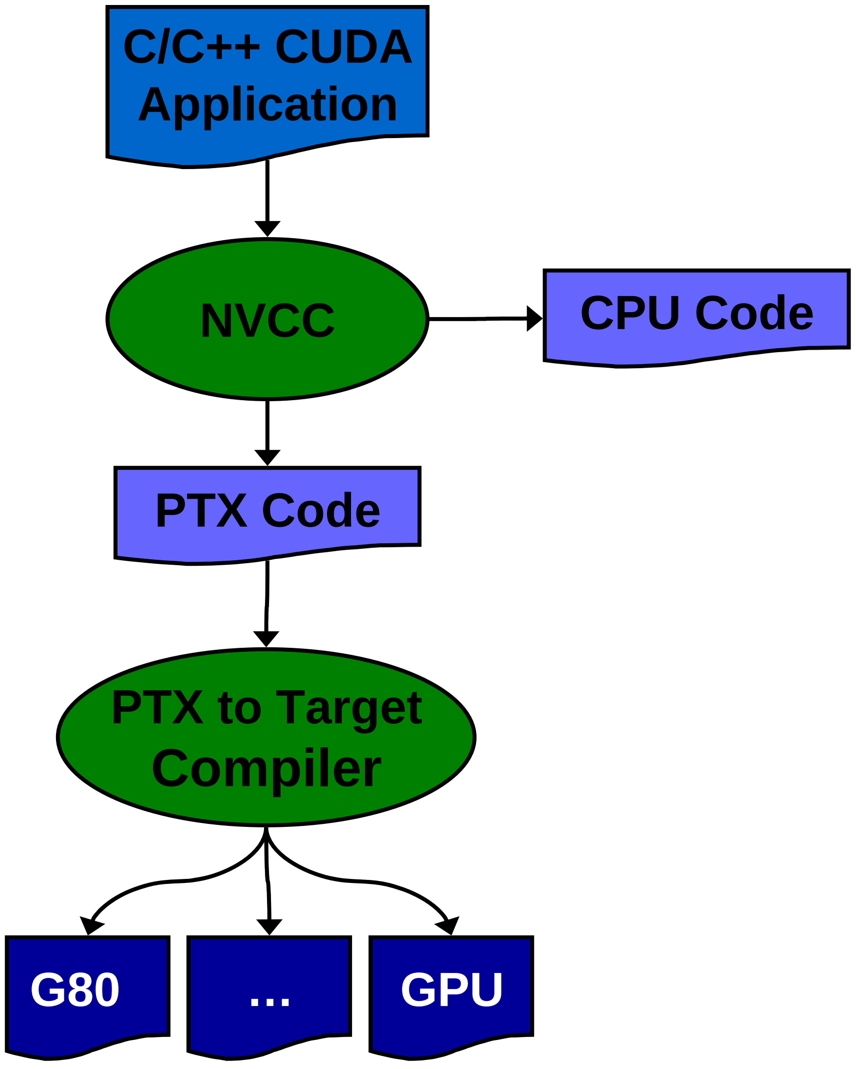
- Bandwidths
- CUDA
  - Overview
  - Development process
  - Performance Optimization
  - Syntax

- Most slides courtesy Massimiliano Fatica (NVIDIA)

Compilation

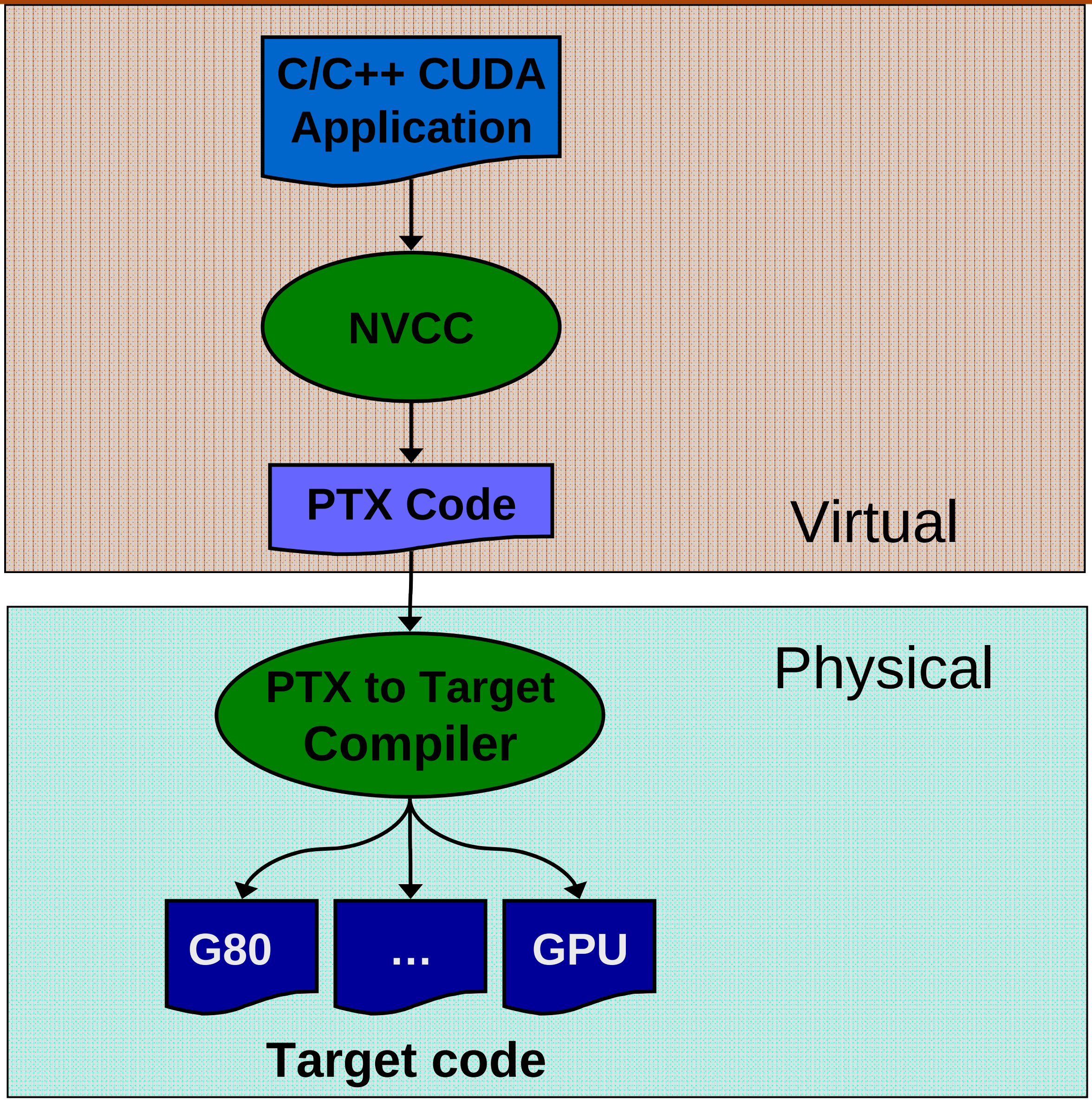
- Any source file containing CUDA language extensions must be compiled with nvcc
- NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- NVCC can output:
  - Either C code (CPU Code)
    - That must then be compiled with the rest of the application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

Compiling CUDA

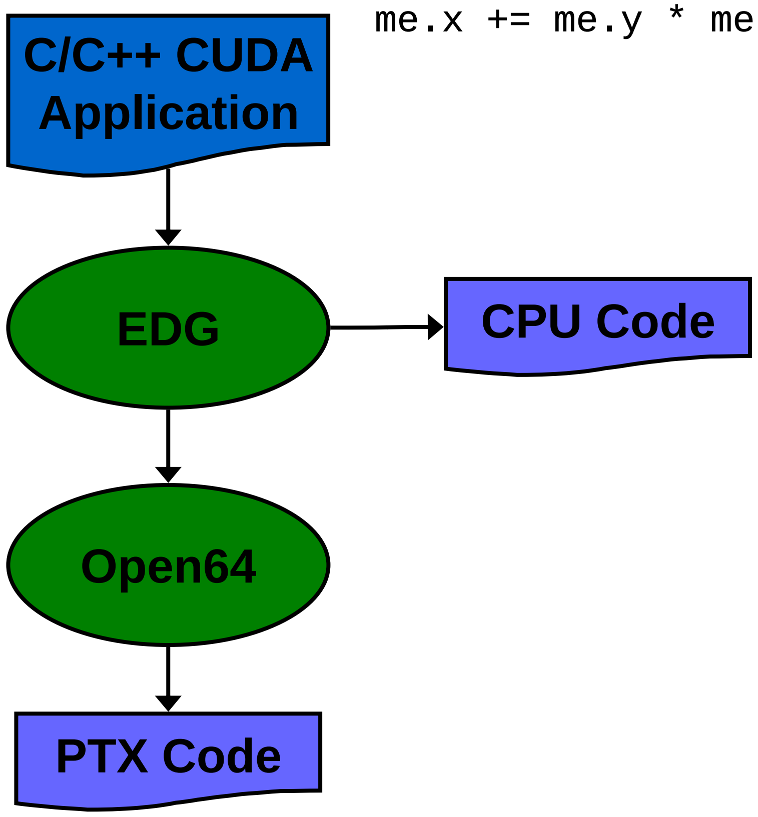


Target code

Compiling CUDA



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;
```

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 `__syncthreads`

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