



High Performance Computing Center
Hanoi University of Science & Technology

CUDA Programming Basic

Duong Nhat Tan (dn.nhattan@gmail.com)

2012

- **CUDA Installation**
- Kernel launches
- Some specifics of GPU code

Design Goals

- Scale to 100's of cores, 1000's of parallel threads
 - G80
 - GT200
 - Tesla
 - Fermi
- + NVIDIA CUDA
- Let programmers focus on parallel algorithms
 - Enable heterogeneous systems (i.e., CPU+GPU)
 - CPU & GPU are separate devices with separate DRAM



GPU Computing with CUDA

- CUDA: Compute Unified Device Architect
- Application Development Environment for NVIDIA GPU
 - Compiler, debugger, profiler, high-level programming languages
 - Libraries (CUBLAS, CUFFT, ..) and Code Samples

CUDA C languages

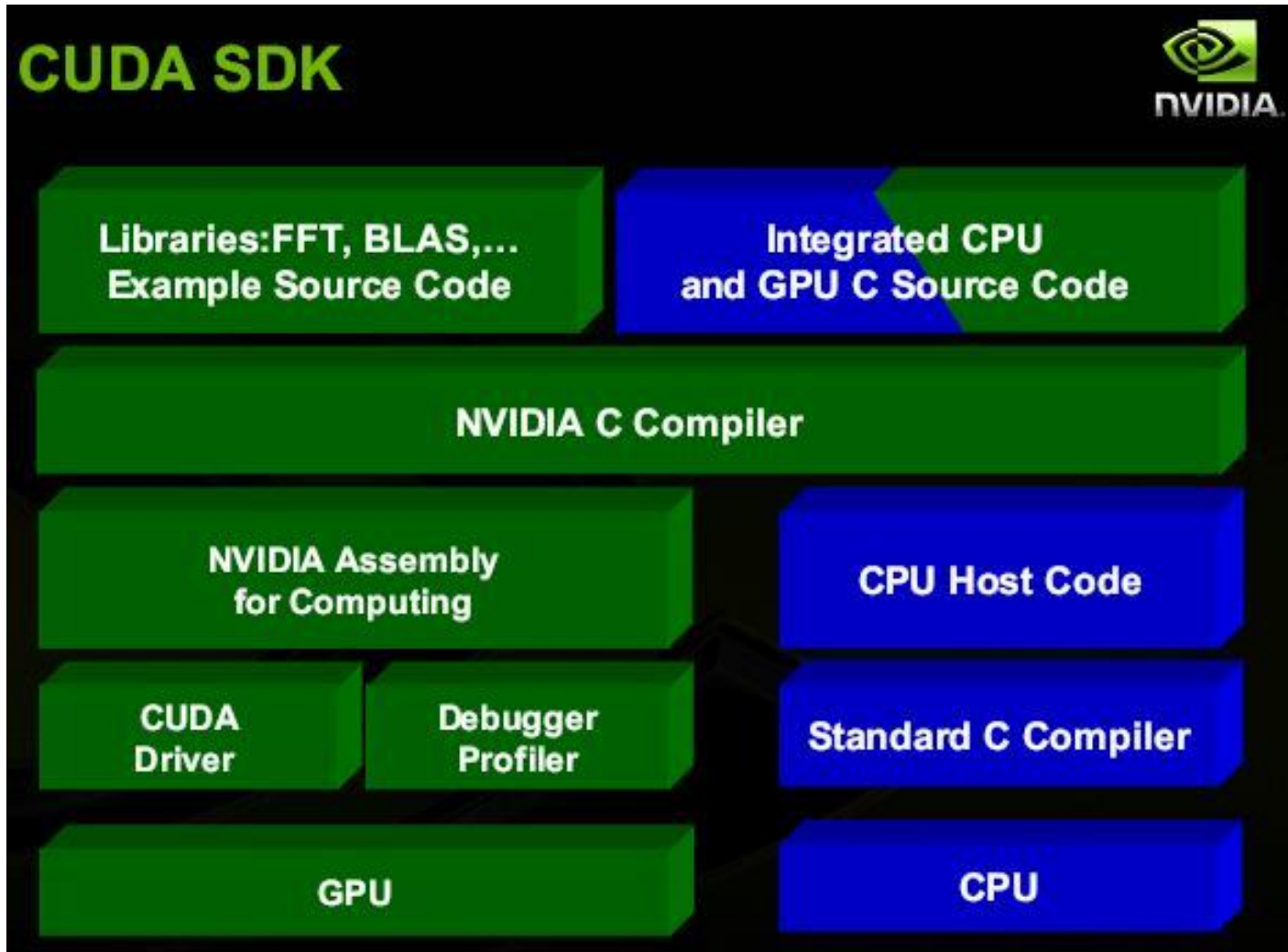
- The extension of C/C++
- Data parallel programming
- Executing a thousands of processes in parallel on GPUs
- Cost of synchronization is not expensive

CUDA Installation

- CUDA development tools consists of three key components:
 - CUDA driver
 - CUDA toolkit
 - Compiler, Assembler
 - Libraries
 - Documentation
 - CUDA SDK








<http://developer.nvidia.com/category/zone/cuda-zone>

CUDA SDK



CUDA files

- Routines that call the device must be in plain C – with extension .cu
- Often 2 files
 - 1) The kernel -- .cu file containing routines that are running on the device.
 - 2) A .cu file that calls the routines from kernel. includes the kernel
- Optional additional .cpp or .c files with other routines can be linked

	..	5/25/2011 4:31:...	rwxr-xr-x
	obj	5/25/2011 4:40:...	rwxr-xr-x
	Makefile	2,121 5/25/2011 4:31:...	rw-r--r--
	matrixMul_gold.cpp	1,532 5/25/2011 4:31:...	rw-r--r--
	matrixMul.cu	11,089 5/25/2011 4:31:...	rw-r--r--
	matrixMul_kernel.cu	3,379 5/25/2011 4:31:...	rw-r--r--
	matrixMul.h	846 5/25/2011 4:31:...	rw-r--r--

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 gcc, g++, cl, ...
- NVCC outputs:
 - C code (host CPU Code)
 - PTX

Compilation

nvcc <filename>.cu [-o <executable>]

- Builds release mode

nvcc -g <filename>.cu

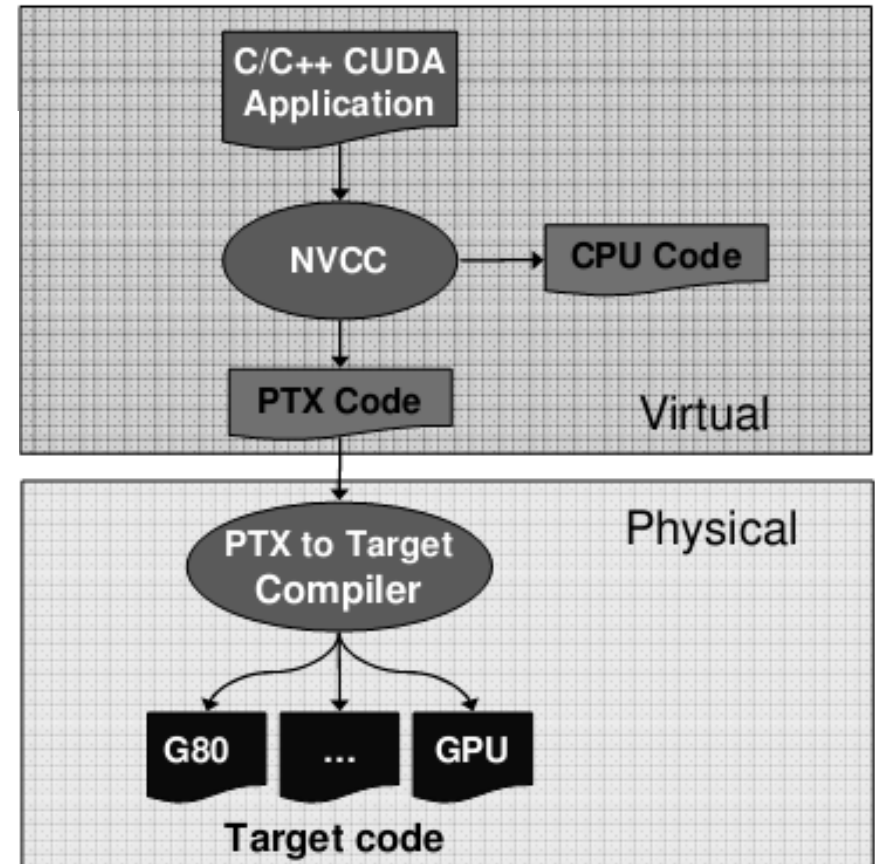
- Builds debug (device) mode
- Can debug host code but not device code (runs on GPU)

nvcc -deviceemu <filename>.cu

- Builds device emulation mode
- All code runs on CPU, but no debug symbols

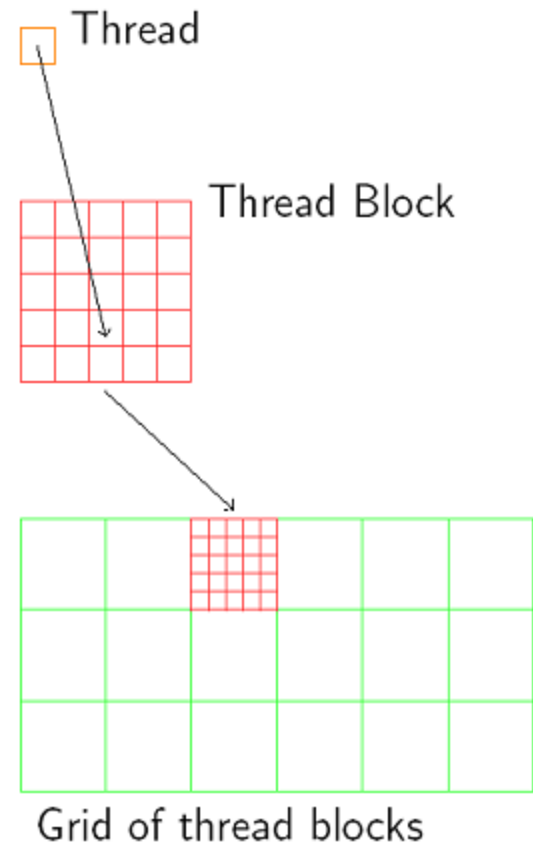
nvcc -deviceemu -g <filename>.cu

- Builds debug device emulation mode
- All code runs on CPU, with debug symbols
- Debug using gdb or other linux debugger



Conceptual Foundations

- Kernels: C functions, when called, are executed by many CUDA threads
- Threads
 - Each thread has a unique thread ID
 - Threads can be 1D, 2D, or 3D
 - Thread id is accessible within the kernel using *threadIdx* variable
- Blocks
 - A group of threads (1D, 2D, 3D)
 - Block id is accessible within the kernel using *blockIdx* variable
- Grids
 - A group of blocks
 - Define the total number of threads (N) can be executed in parallel
 - Threads in different block in the same grid cannot directly communicate with each other



CUDA C kernel

`kernel<<<numBlocks,ThreadsPerBlock>>>(...parameter list ...);`

```
// Kernel definition
__global__ void VecAdd( float * A,
float * B, float * C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

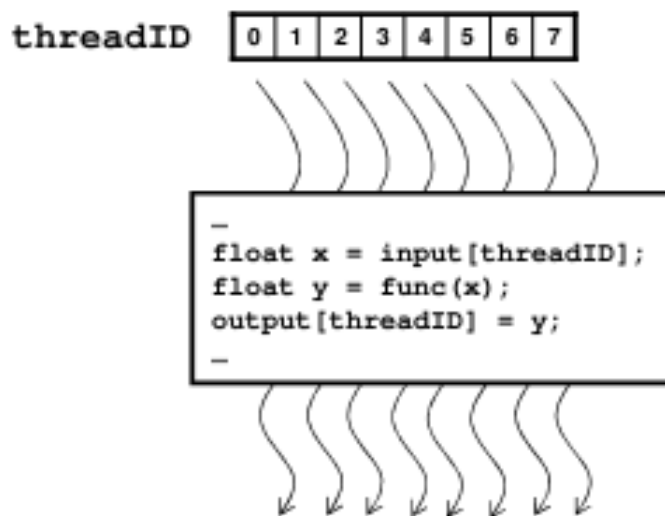
int main()
{
    ...
    // Kernel invocation
    VecAdd<<<1, N>>>(A, B, C);
}
```

```
// function definition
void VecAdd( float * A, float * B, float
* C)
{
    for (int i=0;i<N ; i++)
        C[i] = A[i] + B[i];
}

int main()
{
    ...
    //function invocation
    VecAdd(A, B, C);
}
```

Data Parallelism

- A CUDA kernel is executed by an array of threads
 - All threads run the same code
 - Each thread has an ID that it uses to compute memory addresses and make control decisions



CUDA kernel and thread

- Parallel portions of an application are executed on the device as kernels
 - One kernel is executed at a time
 - Many threads execute each kernel
- Differences between CUDA and CPU threads
 - CUDA threads are extremely lightweight
 - Very little creation overhead
 - Instant switching
 - CUDA uses 1000s of threads to achieve efficiency
 - Multi-core CPUs can use only a few

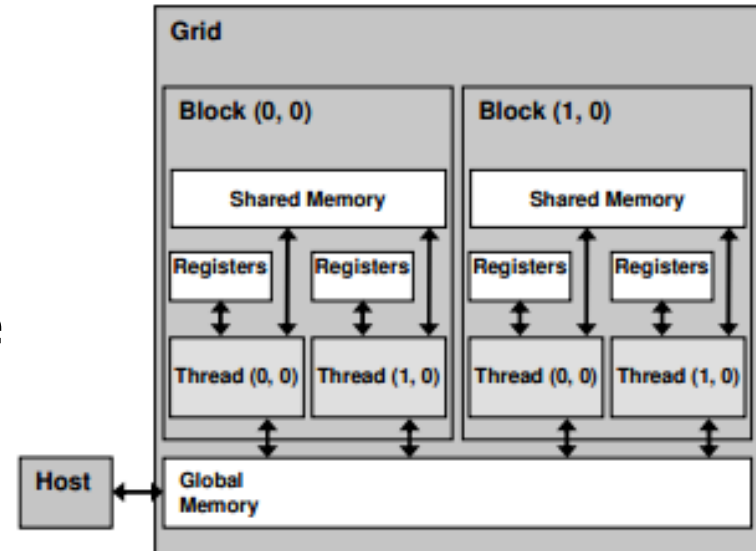
Definitions:

Device = GPU; *Host* = CPU

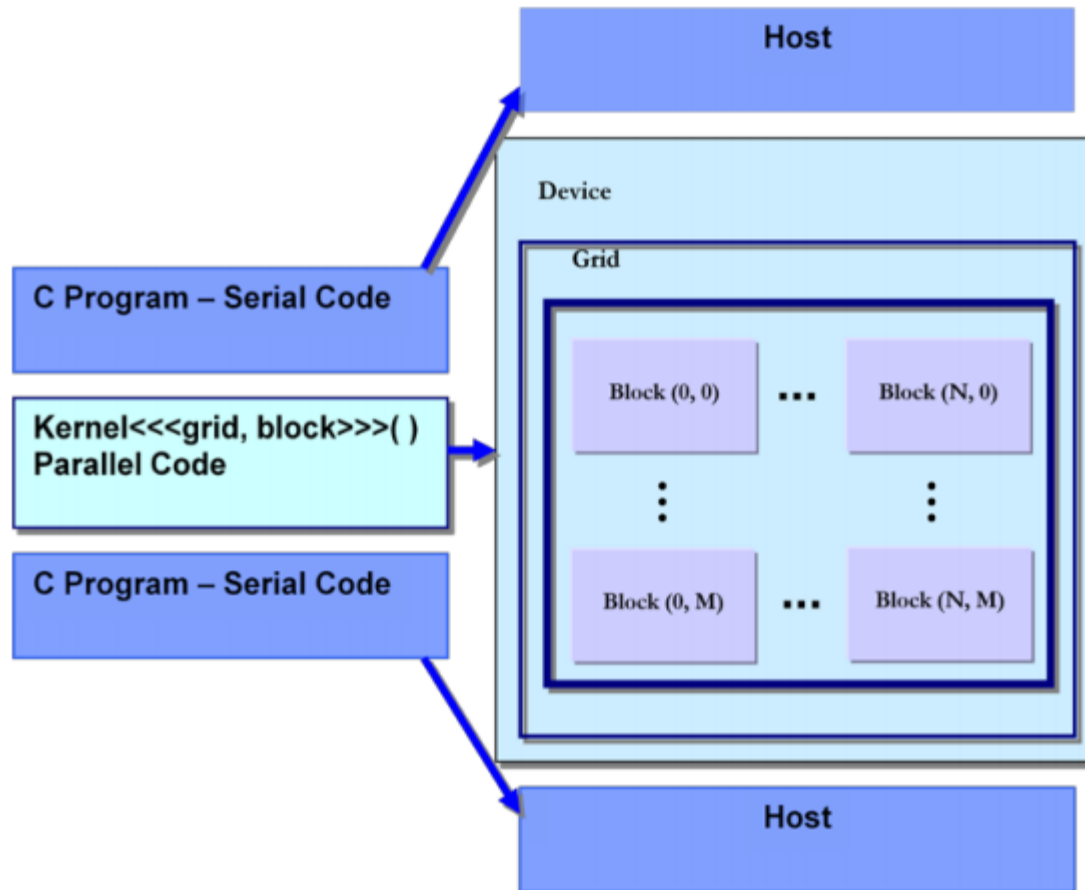
Kernel = function called from the host that runs on the device

Kernel Memory Access

- Registers
- Global Memory
 - Kernel input and output data reside here
 - Off-chip, large
 - Uncached
- Shared Memory
 - Shared among threads in a single block
 - On-chip, small
 - As fast as registers
- The host can read & write global memory but not shared memory



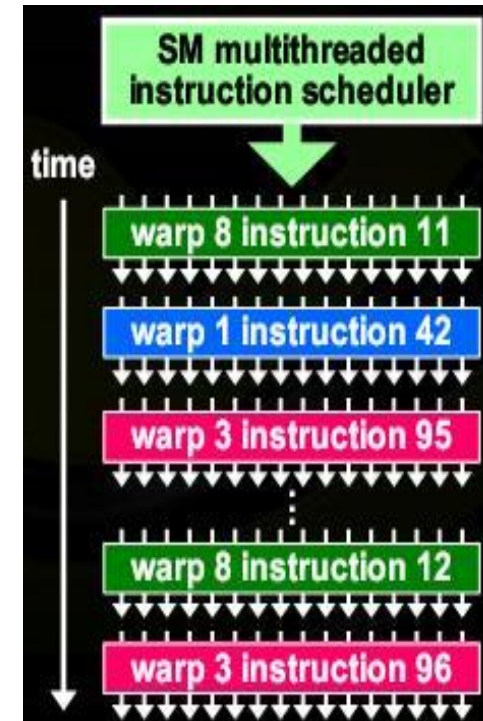
Hetegenerous Programming



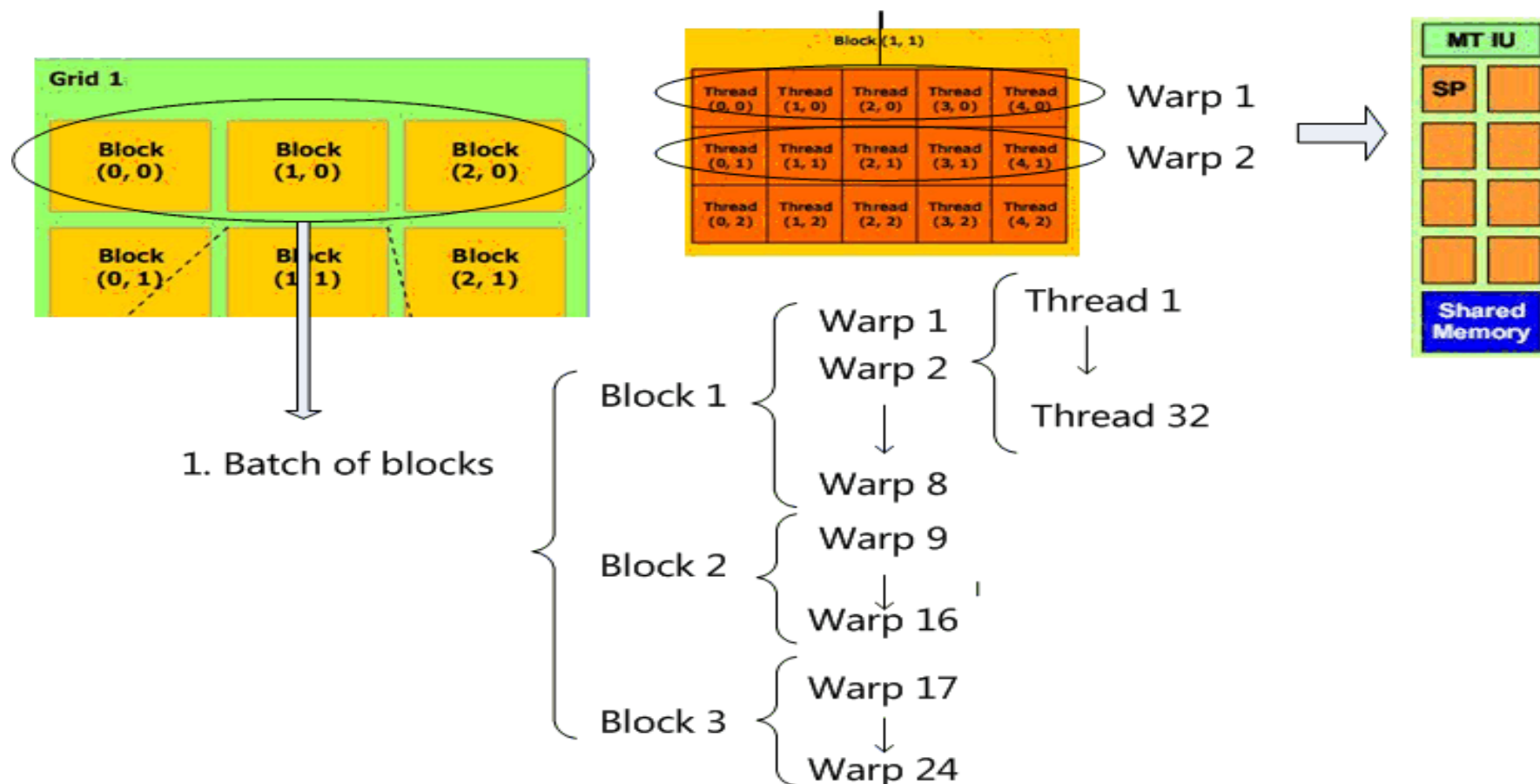
Execution Model

Single Instruction Multiple Thread (SIMT) Execution:

- Groups of 32 threads formed into **warps**
 - always executing same instruction
 - share instruction fetch/dispatch
 - hardware **automatically handles divergence**
- **Warps** are primitive unit of scheduling



Execution Model



GPU Memory Allocation / Release

```
cudaMalloc(void ** pointer, size_t nbytes)  
cudaMemset(void * pointer, int value, size_t count)  
cudaFree(void* pointer)
```

```
int n = 1024;  
int nbytes = 1024*sizeof(int);  
int *d_a = 0;  
cudaMalloc( (void**)&d_a, nbytes );  
cudaMemset( d_a, 0, nbytes);  
cudaFree(d_a);
```

Data copies

- `cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);`
 - Direction specifies locations (host or device) of src and dst
 - Blocks CPU thread: returns after the copy is complete
 - Doesn't start copying until previous CUDA calls complete
- `enum cudaMemcpyKind`
 - `cudaMemcpyHostToDevice`
 - `cudaMemcpyDeviceToHost`
 - `cudaMemcpyDeviceToDevice`

Copying between host and device

- Part1: Allocate memory for pointers d_a and d_b on the device.
- Part2: Copy h_a on the host to d_a on the device.
- Part3: Do a device to device copy from d_a to d_b.
- Part4: Copy d_b on the device back to h_a on the host.
- Part5: Free d_a and d_b on the host

Outline

- CUDA Installation
- **Kernel Launches**
- Hand-On

Executing Code on the GPU

- Kernels are C functions with some restrictions
 - Can only access GPU memory
 - Must have void return type
 - Not recursive
 - No static variables
- Function arguments automatically copied from CPU to GPU memory

Function Qualifiers

- `__global__` :
 - invoked from within host (CPU) code,
 - cannot be called from device (GPU) code
 - must return void
- `__device__` :
 - called from other GPU functions,
 - cannot be called from host (CPU) code
- `__host__` :
 - can only be executed by CPU, called from host
- `__host__` and `__device__` qualifiers can be combined
 - Sample use: overloading operators
 - Compiler will generate both CPU and GPU code

Variable Qualifiers

- `__device__`
 - Stored in device memory (large, high latency, no cache)
 - Allocated with `cudaMalloc` (`__device__` qualifier implied)
 - Accessible by all threads
 - Lifetime: application
- `__shared__`
 - Stored in on-chip shared memory (very low latency)
 - Allocated by execution configuration or at compile time
 - Accessible by all threads in the same thread block
 - Lifetime: kernel execution
- **Unqualified variables:**
 - Scalars and built-in vector types are stored in registers
 - What doesn't fit in register spills to local memory

Launching kernels

- Modified C function call syntax:
kernel<<<dim3 grid, dim3 block>>>(...)
- Execution Configuration ("`<<< >>>`"):
 - grid dimensions: x and y
 - thread-block dimensions: x, y, and z

```
dim3 grid(16, 16);  
dim3 block(16,16);  
kernel<<<grid, block>>>(...);  
kernel<<<32, 512>>>(...);
```

CUDA Built-in Device Variables

- All `__global__` and `__device__` functions have access to these automatically defined variables

`dim3 gridDim;`

- Dimensions of the grid in blocks (at most 2D)

`dim3 blockDim;`

- Dimensions of the block in threads

`dim3 blockIdx;`

- Block index within the grid

`dim3 threadIdx;`

- Thread index within the block

Minimal Kernel

```
__global__ void minimal( int* d_a)
{
    *d_a = 13;
}
```

```
__global__ void assign( int* d_a, int value)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    d_a[idx] = value;
}
```

Common Pattern!

Example: Increment Array Elements

Increment N-element vector a by scalar b



Let's assume $N=16$, $\text{blockDim}=4 \rightarrow 4$ blocks



$\text{blockIdx.x}=0$

$\text{blockDim.x}=4$

$\text{threadIdx.x}=0,1,2,3$

$\text{idx}=0,1,2,3$

$\text{blockIdx.x}=1$

$\text{blockDim.x}=4$

$\text{threadIdx.x}=0,1,2,3$

$\text{idx}=4,5,6,7$

$\text{blockIdx.x}=2$

$\text{blockDim.x}=4$

$\text{threadIdx.x}=0,1,2,3$

$\text{idx}=8,9,10,11$

$\text{blockIdx.x}=3$

$\text{blockDim.x}=4$

$\text{threadIdx.x}=0,1,2,3$

$\text{idx}=12,13,14,15$

$\text{int idx} = \text{blockDim.x} * \text{blockIdx.x} + \text{threadIdx.x};$
will map from local index threadIdx to global index

NB: blockDim should be ≥ 32 in real code, this is just an example

Example: Increment Array Elements

CPU program

```
void increment_cpu(float *a, float b, int N)
{
    for (int idx = 0; idx < N; idx++)
        a[idx] = a[idx] + b;
}
```

```
void main()
{
    ....
    increment_cpu(a, b, N);
}
```

CUDA program

```
__global__ void increment_gpu(float *a, float b, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N)
        a[idx] = a[idx] + b;
}
```

```
void main()
{
    ...
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize) );
    increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Host Synchronization

- All kernel launches are asynchronous
 - control returns to CPU immediately
 - kernel executes after all previous CUDA calls have completed
- `cudaMemcpy()` is synchronous
 - control returns to CPU after copy completes
 - copy starts after all previous CUDA calls have completed
- `cudaThreadSynchronize()`
 - blocks until all previous CUDA calls complete



Host Synchronization Example

```
// copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// execute the kernel
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b);

//run independent CPU code
run_cpu_stuff();

// copy data from device back to host
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);
```

Using shared memory

Size known at compile time

```
__global__ void kernel(...)  
{  
    ...  
    __shared__ float sData[256];  
    ...  
}  
  
int main(void)  
{  
    ...  
    kernel<<<nBlocks, blockSize>>>(...);  
    ...  
}
```

Size known at kernel launch

```
__global__ void kernel(...)  
{  
    ...  
    extern __shared__ float sData[];  
    ...  
}  
  
int main(void)  
{  
    ...  
    smBytes = blockSize*sizeof(float);  
    kernel<<<nBlocks, blockSize,  
        smBytes>>>(...);  
    ...  
}
```


GPU Thread Synchronization

- `void __syncthreads();`
 - Synchronizes all threads in a block
 - Generates barrier synchronization instruction
 - No thread can pass this barrier until all threads in the block reach it
- Allowed in conditional code only if the conditional is uniform across the entire thread block

```
int idx = blockIdx.x * blockDim.x + threadIdx.x
```

```
if (blockIdx.x == blockreverseArray) {  
    shared_data[blockDim.x - (threadIdx.x+1)] = a[idx]  
    __syncthreads();  
    a[idx] = shared_data[threadIdx.x];  
}
```

CUDA Error Reporting to CPU

- All CUDA calls return error code:
 - Except for kernel launches
 - `cudaError_t` type
- `cudaError_t cudaGetLastError(void)`
 - Returns the code for the last error (no error has a code)
 - Can be used to get error from kernel execution
- `char* cudaGetErrorString(cudaError_t code)`
 - Returns a null-terminated character string describing the error

```
printf("%s\n", cudaGetErrorString( cudaGetLastError() ) );
```

My first kernel

- Part1: Allocate device memory for the result of the kernel using pointer d_a.
- Part2: Configure and launch the kernel using a 1-D grid of 1-D thread blocks.
- Part3: Have each thread set an element of d_a as follows:
 $\text{idx} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$
 $\text{d_a}[\text{idx}] = 1000 * \text{blockIdx.x} + \text{threadIdx.x}$
- Part4: Copy the result in d_a back to the host pointer h_a.
- Part5: Verify that the result is correct

reverseArray_singleblock

- Given an input array $\{a_0, a_1, \dots, a_{n-1}\}$ in pointer d_a , store the reversed array $\{a_{n-1}, a_{n-2}, \dots, a_0\}$ in pointer d_b
- Only one thread block launched, to reverse an array of size $N = \text{numThreads} = 256$ elements
- Part 1 (of 1): All you have to do is implement the body of the kernel "reverseArrayBlock()"
- Each thread moves a single element to reversed position
 - Read input from d_a pointer
 - Store output in reversed location in d_b pointer

reverseArray_multiblock

- Given an input array $\{a_0, a_1, \dots, a_{n-1}\}$ in pointer d_a , store the reversed array $\{a_{n-1}, a_{n-2}, \dots, a_0\}$ in pointer d_b
- Multiple 256-thread blocks launched
 - To reverse an array of size N , $N/256$ blocks
- Part 1: Compute the number of blocks to launch
- Part 2: Implement the kernel `reverseArrayBlock()`
- Note that now you must compute both
 - The reversed location within the block
 - The reversed offset to the start of the block

THANK YOU