EN 600.320/420 Parallel Programming

CUDA C Programming
What Is CUDA?

- **CUDA**: Compute Unified Device Architecture
- Created by NVIDIA

- A way to perform computation on the GPU

- Specification for:
  - A computer architecture
  - A language
  - An application interface (API)
A CUDA device is a highly parallel processor
- We assume it can execute many hundreds of threads in parallel
- Threads to Stream Processors ratio > 1

- When writing CUDA software, think in terms of threads, not processors
- Startup and Context Switching costs per thread are very low!!
The CUDA programming model imposes a data decomposition approach.

- The **grid** is the data domain (1D, 2D or 3D)
- The grid is decomposed into **thread blocks**
- A thread block is decomposed into **threads**
CUDA Data Decomposition

Device

<table>
<thead>
<tr>
<th>B (0,0)</th>
<th>B (0,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B (1,0)</td>
<td>B (1,1)</td>
</tr>
<tr>
<td>B (2,0)</td>
<td>B (2,1)</td>
</tr>
</tbody>
</table>

... Thread Block (1,1) ...

<table>
<thead>
<tr>
<th>T (0,0)</th>
<th>T (0,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T (1,0)</td>
<td>T (1,1)</td>
</tr>
<tr>
<td>T (2,0)</td>
<td>T (2,1)</td>
</tr>
</tbody>
</table>
Thread blocks and threads are given unique identifiers

Identifiers can be 1D, 2D or 3D

Used by the kernel to identify which part of a problem to work on

- E.g. which data from memory to read, etc.
- A *kernel* is a program that processes a single data element
- A thread runs the kernel on a data element
Grid -> Blocks -> Threads
Thread Blocks

- A thread block may have up to 512 threads
- All threads in a thread block are run on the same multi-processor
  - Thus can communicate via shared memory
  - And synchronize

- Threads of a block are multiplexed onto a multi-processor as **warps**
Thread Block Scheduling

Multithreaded CUDA Program

Block 0  Block 1  Block 2  Block 3
Block 4  Block 5  Block 6  Block 7

GPU with 2 SMs

SM 0  SM 1
Block 0  Block 1
Block 2  Block 3
Block 4  Block 5
Block 6  Block 7

GPU with 4 SMs

SM 0  SM 1  SM 2  SM 3
Block 0  Block 1  Block 2  Block 3
Block 4  Block 5  Block 6  Block 7
Warps are groups of 32 threads
- Warps are the fundamental scheduling unit of the processor
  - Dispatched two at a time to 16 processors each (on Fermi)
- Each warp forms a SIMD group
  - Thread blocks are not SIMD, Warps are!
Dual issue instructions

- 32 threads
- 16 cores
- 2 units/SM
Memory Addressability

- Thread
  - Per-thread local memory

- Thread Block
  - Per-block shared memory
CUDA Memory Hierarchy

- Multi-Processor
  - CUDA Processor
    - Registers
    - Local Memory
  - Shared Memory
- Global Memory
- Constant Memory
- Texture Memory
CUDA Memory Hierarchy

CUDA Processors have access to:

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Access</th>
<th>Sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Read/Write</td>
<td>Private</td>
</tr>
<tr>
<td>Local Memory</td>
<td>Read/Write</td>
<td>Private</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>Read/Write</td>
<td>Multi-Processor</td>
</tr>
<tr>
<td>Global Memory</td>
<td>Read/Write</td>
<td>Device</td>
</tr>
<tr>
<td>Constant Memory</td>
<td>Read</td>
<td>Device</td>
</tr>
<tr>
<td>Texture Memory</td>
<td>Read</td>
<td>Device</td>
</tr>
</tbody>
</table>
Registers

- Large number of registers per stream processor (1024)
- Zero-clock cycle access
- Store either 64 bit integer or 64 bit float
Shared Memory

- A block of memory that is shared by all stream processors in a multi-processor
  - 48 or 16K per SM in Fermi

- 16KB per block, stored in 16x1KB banks
- Very fast to access (i.e. as fast as registers!) without bank conflicts
Global Memory

- The large block of memory shared by all multi-processors on the compute device
- Size depends on device – 256MB to 24GB (Tesla K80 is 2x 12GB)
- High bandwidth (K80 is 480 GB/s)

- Slow to access – several hundred clock cycle latency.
Fermi Memory Model

Fermi Memory Hierarchy
- Thread
- Shared Memory
- L1 Cache
- L2 Cache
- DRAM
First CUDA Program
We will create a simple CUDA program to add two vectors

- $U = \{u_0, u_1, \ldots, u_n\}$
- $V = \{v_0, v_1, \ldots, v_n\}$
- $W = U + V = \{u_0 + v_0, u_1 + v_1, \ldots, u_n + v_n\}$
Vector-Vector Addition

\[ \mathbf{W} = \mathbf{U} + \mathbf{V} \quad \mathbf{W}_i = \mathbf{U}_i + \mathbf{V}_i \]

- Easy to parallelize: Each element is independent!
Threads: Each element is computed by a separate thread
**Thread Blocks**

- **Blocks:** Group sets of adjacent elements into blocks
  - To conform with device parameters
- **Grid**: The entire vector

\[
\begin{align*}
U & \quad + \\
V & \quad = \\
W &
\end{align*}
\]
Our program will be a single source file, with two parts:

Device Code

- A kernel to perform the addition of two elements
__global__ void VectorAdditionKernel(
    const float* pVectorA,
    const float* pVectorB,
    float* pVectorC)
{
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
}
CUDA defines a language that is similar to C/C++

CUDA source code contains a mix of *host* and *device* code and data
CUDA Language

- CUDA defines a language that is similar to C/C++

- Important Differences:
  - Runtime Library
  - Functions
  - Classes, Structs, Unions
CUDA Runtime Library

- Code that runs on the device can’t use normal C/C++ Runtime Library functions
  - No printf, fread, malloc, etc
- Most math functions have device equivalent
CUDA Runtime Functions

- There are a number of device specific functions/intrinsics available:
  - `__syncThreads`
  - `__mul24`
  - `atomicAdd, atomicCAS, atomicMin, ...`
Functions

- On a CUDA device, there is no stack
- By default, all function calls are inlined

- All local variables, function arguments are stored in registers
- **NO** function recursion

- No function pointers
CUDA defines a language that is similar to C/C++

- Syntactic extensions:
  - Declaration Qualifiers
  - Built-in Variables
  - Built-in Types
  - Execution Configuration
Declspec = declaration specifier / declaration qualifier

A modifier applied to declarations of:
- Variables
- Functions

Examples: const, extern, static
CUDA uses the following declspecs for functions:

- __device__
- __host__
- __global__
__device__

- Declares that a function is compiled to, and executes on the device
- Callable only from another function on the device
__host__

- Declares that a function is compiled to and executes on the host
- Callable only from the host
- Functions without any CUDAdeclspec are host by default
`__global__`

- Declares that a function is compiled to and executes on the device
- Callable from the host
- Used as the entry point from host to device
__global__ void VectorAdditionKernel(
    const float* pVectorA,
    const float* pVectorB,
    float* pVectorC)
{
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
}
Vector Types

- Can construct a vector type with special function:
  ```
  make_{typename}(v_0, v_1, ...)
  ```

- Can access elements of a vector type with ".x", ".y", ".z", ".w": `vecvar.x`

- `dim3` is a special vector type for grids, same as `uint3`
CUDA provides four global, built-in variables:
- `threadIdx`, `blockIdx`, `blockDim`, `gridDim`

- Typed as a ‘`dim3`’ or ‘`uint3`’

- Accessible only from device code
- Cannot take address
- Cannot assign value
Our program will be a single source file, with two parts:

- Allocate GPU memory for vector
- Copy vector from host to device memory
- Launch kernel
- Copy vector from device to host memory
__global__ void VectorAdditionKernel(
    const float* pVectorA,
    const float* pVectorB,
    float* pVectorC)
{
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
}
float* pDeviceVectorA = 0;
float* pDeviceVectorB = 0;
float* pDeviceVectorC = 0;

cudaMalloc((void**)&pDeviceVectorA, VectorSize);
cudaMalloc((void**)&pDeviceVectorB, VectorSize);
cudaMalloc((void**)&pDeviceVectorC, VectorSize);

cudaMemcpy(pDeviceVectorA, pHostVectorA, VectorSize, cudaMemcpyHostToDevice);
cudaMemcpy(pDeviceVectorB, pHostVectorB, VectorSize, cudaMemcpyHostToDevice);
cudaMemcpy(pDeviceVectorC, pHostVectorC, VectorSize, cudaMemcpyHostToDevice);

...
There is no `malloc` or `free` function that can be called from device code

→ How can we allocate memory?

- From the host with `cudaMalloc()`
  - And copy data in from host to initialize
bool VectorAddition(
    unsigned N,
    const float* pHostVectorA,
    const float* pHostVectorB,
    float* pHostVectorC)
{
    const unsigned BLOCKSIZE = 512;
    unsigned ThreadCount = N;
    unsigned BlockCount = N / BLOCKSIZE;
    unsigned VectorSize = ThreadCount * sizeof(float);

    ...
... 

VectorAdditionKernel<<<BlockCount,BLOCKSIZE>>>(
    pDeviceVectorA,
    pDeviceVectorB,
    pDeviceVectorC);

...

Host Code – Execute Kernel
Execution Configuration

- CUDA provides syntactic sugar to launch the execution of kernels

```c
Func<<<GridDim, BlockDim>>>(Arguments, ...)
```
Execution Configuration

```
Func<<<GridDim, BlockDim>>> (Arguments, ...)
```

- Func is a **global** function
GridDim is a ‘dim3’ typed expression giving the size of the grid (i.e. problem domain)
Execution Configuration

```
Func<<<GridDim, BlockDim>>>(Arguments, ...)
```

- `BlockDim` is a ‘dim3’ typed expression giving the size of a thread block
The compiler turns this type of statement into a block of code that configures, and launches the kernel.
Host Code – Read result from GPU

```c
... cudaMemcpyp(pHostVectorC, pDeviceVectorC, VectorSize, cudaMemcpyDeviceToHost);
 ...
}
```
CUDA uses the following declaration qualifiers for variables:

- __device__
- __shared__
- __constant__

Only apply to global variables
Declares that a global variable is stored on the device

- The data resides in global memory
- Has lifetime of the entire application
- Accessible to all GPU threads
- Accessible to the CPU via API
__shared__

- Declares that a global variable is stored on the device
- The data resides in shared memory
- Has lifetime of the thread block
- Accessible to all threads, one copy per thread block
If not declared as volatile, reads from different threads are not visible unless a synchronization barrier used

- Not accessible from CPU
Declares that a global variable is stored on the device

- The data resides in constant memory
- Has lifetime of entire application
- Accessible to all GPU threads (read only)
- Accessible to CPU via API (read-write)
Vector Size

- What if the vector size is not an integral number of blocks?

- **Option 1:**
  Perform bounds checking in kernel

- **Option 2:**
  Pad out the vector to correct length
Grid is 1-dimensional
Maximum of 512 threads in a block
Maximum of 65536 blocks in a 1D grid
→ Maximum vector size is 65536 times 512

How do we operate on a vector larger than 16M elements?
Maximum Vector Size

- How do we operate on a vector larger than 16M elements?

- **Option 1:**
  Use a 2-D indexing scheme

- **Option 2:**
  Compute with several grids
The GPU is faster than the CPU
But, computing on the GPU involves overhead:
→ Must get data to/from the GPU

Where is the “break-even” point?
CUDA Vector-Vector Addition Performance
CUDA Vector-Vector Addition Performance

- CPU
- Memory Copy
- GPU
Matrix addition is not a good CUDA program
  - Why? In terms of Roofline? Not enough operational intensity

Never overcome the data transfer
  - Not enough computation
  - Better on the CPU