CIS 665: GPU Programming

The CUDA Programming Model
References

• Borrowed (Infringed) from
  – Nvidia (Kitchen)
  – David Kirk + Wen-Mei Hwu (UIUC)
  – Gary Katz and Joe Kider
Historical GPGPU

• General Purpose computation using GPU and graphics API
  – GPU accelerates critical parts of application

• Data parallel algorithms leverage GPU strengths
  – Large data arrays, streaming throughput
  – Fine-grain SIMD parallelism
  – Low-latency floating point (FP) computation

• Applications – see //GPGPU.org
  – Game effects (FX) physics, image processing
  – Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting
Previous GPGPU Constraints

- Working with (against) graphics API
  - Memory through texture access and render-to-texture
- Limited by shader capabilities
  - Constrained outputs
  - Graphics-focused instruction sets
- Instruction sets
- Inter-thread communication limited
  - Need multiple passes to communicate
  - *Gather*, not *scatter*
CUDA

• “Compute Unified Device Architecture”
• Architecture and programming model
  – User kicks off batches of threads on the GPU
  – GPU becomes dedicated super-threaded, massively data parallel co-processor
• Targeted software stack and drivers
  – Compute oriented drivers, language, and tools – No more graphics API
  – Standalone driver optimized for computation
  – Interface designed for compute – No more hijacking!
  – Data sharing with OpenGL buffer objects
  – Explicit host/device memory management
    • Optimized memory transfer (pinned memory, asynchronous transfer)
Overview

• CUDA programming model
  – Concepts and data types
• Host interface
• Examples to get your feet wet!
• Not covered:
  – Performance tuning
  – Advanced libraries
What Programmer Expresses in CUDA

• Where computation occurs
• Where data is, who can access
  – Device v. Host
  – Manual data moving
• How do threads work together
  – __syncthreads()
Almost C!

- **Host + Device**
  - **Host C code**
    - Serial components, management, launch
  - **Device** kernels execute highly parallel components

Serial Code (host)

Parallel Kernel (device)
KernelA\(\lll n_{Blk}, n_{Tid} \rrr\)(args);

Serial Code (host)

Parallel Kernel (device)
KernelB\(\lll n_{Blk}, n_{Tid} \rrr\)(args);
CUDA Devices and Threads

- A compute **device**
  - Is a **coprocessor** to the CPU or **host**
  - Has its own DRAM (**device memory**)  
  - Runs many **threads in parallel**
  - Not explicitly a GPU

- Data-parallel portions of an application are expressed as device **kernels** which run on many threads
GPU v. CPU Threads

- GPU threads are extremely lightweight
- Very little creation overhead
- GPU needs 1000s of threads for full efficiency
  - Goal: Saturate the GPU
  - Threads used to hide latency and memory access
- Multi-core CPU needs only a few
Extended C

• Declspecs
  – global, device, shared, local, constant

• Keywords
  – threadIdx, blockIdx

• Intrinsics
  – __syncthreads

• Runtime API
  – Memory, symbol, execution management

• Function launch

```c
__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);
```
Compilation Model

Integrated source
(foo.cu)

cudacc
EDG C/C++ frontend
Open64 Global Optimizer

GPU Assembly
(foo.s)

OGC

G80 SASS
(foo.sass)

CPU Host Code
(foo.cpp)

gcc / cl

Mark Murphy, “
NVIDIA’s Experience with Open64,”
www.capsl.udel.edu/conferences/open64/2008/Papers/101.doc
CUDA Advantages

• 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
Arrays of Parallel Threads

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

```c
float x = input[threadID];
float y = func(x);
output[threadID] = y;
...```

```
... threadID 0 1 2 3 4 5 6 7
...```
Data Model $\iff$ Thread Model

- How do we map our data parallel computation to CUDA thread model?
Thread Blocks: Scalable Cooperation

- Major advantage of CUDA v. shading languages
  - Threads within a block cooperate via \textit{shared memory, atomic operations} and \textit{barrier synchronization}
  - Threads in different blocks cannot cooperate
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

![Diagram of thread batching in grids and blocks](https://example.com/thread-batching-diagram.png)

Courtesy: NVIDIA
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
  - …
Grid and Blocks

• Threads are organized in a 2-level structure
  – Grid of blocks of threads

• Proper use of blocks is *critical*
  – Shared memory!
  – Design algorithms to take advantage of fast local communication
  – Amortize global memory access across block
  – How you block affects allocation of computation to SM's
Synchronization

• Threads within a block can communicate through __shared__ memory
  – Allocated per-block
  – Register-speed

• __syncthreads() intrinsic
  – Barrier synchronization within a block
  – Let all threads wait for a coordinated operation to complete

• New! On Recent Cards!
  – Vote (any, all) intrinsic
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
Global, Constant, and Texture Memories (Long Latency Accesses)

- **Global memory**
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads

- **Texture and Constant Memories**
  - Initialized by host
  - Contents visible to all threads
CUDA Memory Model Overview

• Global memory
  – Main means of communicating R/W Data between host and device
  – Contents visible to all threads
  – Long latency access
• We will focus on global memory for now
  – Constant and texture memory will come later
CUDA Device Memory Allocation

- **cudaMalloc()**
  - Allocates object in the device **Global Memory**
  - Requires two parameters
    - Address of a pointer to the allocated object
    - Size of allocated object
  - **cudaFree()**
  - Frees object from device **Global Memory**
    - Pointer to freed object
CUDA Keywords
## CUDA Function Declarations

<table>
<thead>
<tr>
<th></th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong>  float DeviceFunc()</td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><strong>global</strong>  void KernelFunc()</td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><strong>host</strong>   float HostFunc()</td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>

- **__global__** defines a kernel function
  - Must return **void**
- **__device__** and **__host__** can be used together
Some C++ features

- Structs, no classes
- Operator overloading
- Templates
Calling a Kernel Function – The heart of CUDA

• A kernel function is launched with an execution configuration:

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

dim3 DimGrid(100, 50);    // 5000 thread blocks

dim3 DimBlock(4, 8, 8);   // 256 threads per block

size_t SharedMemBytes = 64; // 64 bytes of shared memory

KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

• Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking
Limitations of `__device__` functions

- For `__device__` functions...
  - No function pointers
  - No recursion

- Why
  - Device functions are almost always inlined
CUDA Device Memory Allocation (cont.)

• Code example:
  – Allocate a 64 * 64 single precision float array
  – Attach the allocated storage to Md
  – Keep track of device vs. host memory!

```
TILE_WIDTH = 64;
Float* Md
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);
cudaMalloc((void**)&Md, size);
cudaFree(Md);
```
CUDA Host-Device Data Transfer

• cudaMemcpy()
  – memory data transfer
  – Requires four parameters
    • Pointer to destination
    • Pointer to source
    • Number of bytes copied
    • Type of transfer
      – Host to Host
      – Host to Device
      – Device to Host
      – Device to Device

• Asynchronous transfers available
CUDA Host-Device Data Transfer (cont.)

- Code example:
  - Transfer a 64 * 64 single precision float array
  - M is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToDeviceToHost are symbolic constants

```c
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
...(computation) ...
cudaMemcpy(M, Md, size, cudaMemcpyDeviceToDeviceToHost);
```
GeForce 8800 Series Technical Specs

- Card capabilities limit possible configurations
- Maximum number of threads per block: 512
- Maximum size of each dimension of a grid: 65,535
- Number of streaming multiprocessors (SM):
  - GeForce 8800 GTX: 16 @ 675 MHz
  - GeForce 8800 GTS: 12 @ 600 MHz
- Device memory:
  - GeForce 8800 GTX: 768 MB
  - GeForce 8800 GTS: 640 MB
- Shared memory per multiprocessor: 16KB divided in 16 banks
- Constant memory: 64 KB
- Warp size: 32 threads (16 Warps/Block)
  - Warps are the groups of threads actually processed in parallel by an SM
GPU Memory Allocation / Release

Host (CPU) manages device (GPU) memory:
- `cudaMalloc (void ** pointer, size_t nbytes)`
- `cudaMemset (void * pointer, int value, size_t count)`
- `cudaFree (void* pointer)`

```c
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);`

- returns after the copy is complete
- blocks CPU thread until all bytes have been copied
- doesn’t start copying until previous CUDA calls complete

```c
enum cudaMemcpyKind
```
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice

Non-blocking memcopies are provided
Code Walkthrough 1

- Allocate CPU memory for $n$ integers
- Allocate GPU memory for $n$ integers
- Initialize GPU memory to 0s
- Copy from GPU to CPU
- Print the values
Code Walkthrough 1

#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers
```c
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a = 0, *h_a = 0; // device and host pointers
    h_a = (int*)malloc(num_bytes);
    cudaMalloc((void**)&d_a, num_bytes);

    if (0==h_a || 0==d_a)
    {
        printf("couldn't allocate memory\n");
        return 1;
    }
```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemcpy( d_a, 0, num_bytes );
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a = 0, *h_a = 0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMemcpy( (void**)&d_a, num_bytes);

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemcpy(d_a, 0, num_bytes);
    cudaMemcpy(h_a, d_a, num_bytes, cudaMemcpyDeviceToHost);

    for(int i=0; i<dimx; i++)
        printf("%d ", h_a[i]);
    printf("\n");

    free(h_a);
    cudaFree(d_a);
    return 0;
}
Basic Kernels and Execution on GPU
CUDA Programming Model

- Parallel code (kernel) is launched and executed on a device by many threads
- Threads are grouped into thread blocks
- Parallel code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in thread and block ID variables
Thread Hierarchy

- Threads launched for a parallel section are partitioned into thread blocks
  - Grid = all blocks for a given launch
- Thread block is a group of threads that can:
  - Synchronize their execution
  - Communicate via shared memory
Code executed on GPU

C function with some restrictions:
- Can only access GPU memory
- No variable number of arguments
- No static variables
- No recursion

Must be declared with a qualifier:
- \_global\_ : launched by CPU, cannot be called from GPU must return void
- \_device\_ : called from other GPU functions, cannot be launched by the CPU
- \_host\_ : can be executed by CPU
- \_host\_ and \_device\_ qualifiers can be combined

sample use: overloading operators
A Simple Running Example
Matrix Multiplication

• A simple matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs
  – Leave shared memory usage until later
  – Local, register usage
  – Thread ID usage
  – Memory data transfer API between host and device
  – Assume square matrix for simplicity
Programming Model: Square Matrix Multiplication Example

- $P = M \times N$ of size $\text{WIDTH} \times \text{WIDTH}$
- Without tiling:
  - One thread calculates one element of $P$
  - $M$ and $N$ are loaded $\text{WIDTH}$ times from global memory
Memory Layout of a Matrix in C

\[ M \]

\[
\begin{array} {cccc}
M_{0,0} & M_{1,0} & M_{2,0} & M_{3,0} \\
M_{0,1} & M_{1,1} & M_{2,1} & M_{3,1} \\
M_{0,2} & M_{1,2} & M_{2,2} & M_{3,2} \\
M_{0,3} & M_{1,3} & M_{2,3} & M_{3,3}
\end{array}
\]
// Matrix multiplication on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width) {
    for (int i = 0; i < Width; ++i) {
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * Width + k];
                double b = N[k * Width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
    }
}
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);
    float* Md, Nd, Pd;

    1. // Allocate and Load M, N to device memory
       cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
       cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

    cudaMalloc(&Pd, size);
    cudaMemcpy(Pd, P, size, cudaMemcpyHostToDevice);

    // Allocate P on the device
    cudaMalloc(&Pd, size);
Step 3: Output Matrix Data Transfer (Host-side Code)

2. // Kernel invocation code – to be shown later
   ...

3. // Read P from the device
   cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

   // Free device matrices
   cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
Step 4: Kernel Function

// Matrix multiplication kernel – per thread code

__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
}
Step 4: Kernel Function (cont.)

for (int k = 0; k < Width; ++k) {
    float Melement = Md[threadIdx.y*Width+k];
    float Nelement = Nd[k*Width+threadIdx.x];
    Pvalue += Melement * Nelement;
}

Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
Step 5: Kernel Invocation
(Host-side Code)

// Setup the execution configuration
dim3 dimGrid(1, 1);
dim3 dimBlock(Width, Width);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
Only One Thread Block Used

- One Block of threads compute matrix Pd
  - Each thread computes one element of Pd
- Each thread
  - Loads a row of matrix Md
  - Loads a column of matrix Nd
  - Perform one multiply and addition for each pair of Md and Nd elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- Size of matrix limited by the number of threads allowed in a thread block (512)
Step 7: Handling Arbitrary Sized Square Matrices

- Have each 2D thread block to compute a \((\text{TILE\_WIDTH})^2\) sub-matrix (tile) of the result matrix
  - Each has \((\text{TILE\_WIDTH})^2\) threads
- Generate a 2D Grid of \((\text{WIDTH}/\text{TILE\_WIDTH})^2\) blocks

You still need to put a loop around the kernel call for cases where \(\text{WIDTH}/\text{TILE\_WIDTH}\) is greater than max grid size (64K)!
Some Useful Information on Tools
Compiling a CUDA Program

- Parallel Thread eXecution (PTX)
- Virtual Machine and ISA
- Programming model
- Execution resources and state

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

```
ld.global.v4.f32  {$f1,$f3,$f5,$f7}, [$r9+0];
mad.f32           $f1, $f5, $f3, $f1;
```
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 outputs:
  • C code (host CPU Code)
    • Must then be compiled with the rest of the application using another tool
  • PTX
    • Object code directly
    • Or, PTX source, interpreted at runtime
Linking

- Any executable with CUDA code requires two dynamic libraries:
  - The CUDA runtime library (*cudart*)
  - The CUDA core library (*cuda*)
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

• 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 could 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.
Nexus (Beta)

- Visual Studio plug-in
- Real debugging on hardware
  - Memory inspection
  - Breakpoints
  - Watchpoints
- Activity tracing on a single timeline
  - API calls (OpenGL, CUDA, Cg)
  - Mem transfers
- Haven't tried it yet, you can apply online for Beta key