Graphical Processing Units and CUDA

Lecture for CPSC 5155
Edward Bosworth, Ph.D.
Computer Science Department
Columbus State University
The Graphics Coprocessor

• From the earliest VGA designs, the graphics unit has been designed as a special purpose processor, attached to the CPU using a high-speed I/O-type link.

• There are many CPU instructions that a GPU need not implement. This simplifies design of a GPU.

• A modern NVIDIA graphics system would include a high-performance dual-processor main CPU, a few GB of local memory, a high-end disk drive, and one or more graphics cards.
Chapter 7 — Multicores, Multiprocessors, and Clusters — 3

Graphics in the System
Why does graphics hardware exist?

Special-purpose hardware tends to disappear over time

- Lisp machines and CAD workstations of the 80s
- CISC CPUs

iAPX432
(circa 1982)
www.dvorak.org/blog/

Symbolics Lisp Machines
(circa 1984)
www.abstractscience.freeserve.co.uk/symbolics/photos/
GPU Architectures

- Processing is highly data-parallel
  - GPUs are highly multithreaded
  - Use thread switching to hide memory latency
    - Less reliance on multi-level caches
  - Graphics memory is wide and high-bandwidth

- Trend toward general purpose GPUs
  - Heterogeneous CPU/GPU systems
  - CPU for sequential code, GPU for parallel code

- Programming languages/APIs
  - DirectX, OpenGL
  - C for Graphics (Cg), High Level Shader Language (HLSL)
  - Compute Unified Device Architecture (CUDA)
Why does graphics hardware exist?

Graphics acceleration has been around for 40 years.

Why do GPUs remain? Confluence of four things:

- **Performance differentiation**
  - GPUs are much faster than CPUs at 3-D rendering tasks

- **Work-load sufficiency**
  - The accelerated 3-D rendering tasks make up a significant portion of the overall processing (thus Amdahl’s law doesn’t limit the resulting performance increase).

- **Strong market demand**
  - Customer demand for 3-D graphics performance is strong
  - Driven by the games market

- **Ubiquity**
  - With the help of standardized APIs/architectures (OpenGL and Direct3D) GPUs have achieved ubiquity in the PC market
  - Inertia now works in favor of continued graphics hardware
GPU and GPGPU

• GPU is a graphics processing unit
• Originally driven for better computer graphics performance
• GPUs were originally meant as graphics accelerator chips to help the CPU
• General Purpose GPU (GPGPU) programming refers to the now common case where the GPU can be used to accelerate other (non-graphical) calculations
GPU Evolution (1)

- VGA – Video Graphics Array controllers – originally a memory controller and display generator connected to DRAM
- Variations in 1990’s to add more functionality
- Circa 1997 3D accelerator functions:
  - Triangle setup and rasterization
  - Texture mapping and shading (decals)
- GPU term coined circa 2000 when typical graphics chip already did most of the standard graphics pipeline operations
• Programmable processor (cores) replaced fixed dedicated logic
• GPUs became massively parallel processors
• Floating point and (recently) double precision
• Hundreds of cores, thousands of threads...
• Recently become programmable in eg C++ and variants like CUDA and OpenCL...
Origin of CUDA

• The Compute Unified Device Architecture, developed by NVIDIA Corporation, arose from a series of experiments in the early 2000’s.
• Graphics processors were becoming very fast.
• It was discovered that many numerical simulation problems could be forced into a form that could be adapted to execute on a graphics card.
• The difficulty was that the GPU had to be controlled using an API designed for graphics.
GPGPU and CUDA

- GPGPU stands for General Purpose computation on a Graphics Processing Unit.
- As mentioned above, this style used the traditional graphics API and graphics pipeline in a way that was only accidentally useful.
- The CUDA was developed intentionally to allow direct access to the graphics hardware, with programming in a variant of C/C++. 
GPU Trends

• Implement OpenGL and DirectX
• New GPUs every 12-18 months
• Coming together of parallel computing and graphics in a new and exciting way
• Heterogeneous computing:
  – Data parallelism on the GPU
  – More coarse-grained parallelism on the (multi-core) CPU
Parallel Computing on a GPU

- 8-series GPUs deliver 25 to 200+ GFLOPS on compiled parallel C applications
  - Available in laptops, desktops, and clusters
- GPU parallelism is doubling every year
- Programming model scales transparently
- Programmable in C with CUDA tools
- Multithreaded SPMD model uses application data parallelism and thread parallelism
Example: NVIDIA Tesla

Streaming multiprocessor

8 × Streaming processors
A Fixed Function GPU Pipeline
Programmable Vertex and Pixel Processors

An example of separate vertex processor and fragment processor in a programmable graphics pipeline
Unified Graphics Pipeline
Multi-threading hides latency

Memory data available (dependency resolved)

Memory reference (or resulting data dependency)

struct {
    float x, y, z, w;
    float r, g, b, a;
} vertex;

Processor stalls if no threads are ready to run.
Possible result of large thread context (too many live registers)

Instruction fetch and execute

Ready to Run Threads

Blocked Threads

Memory data available (dependency resolved)
Overview

• CUDA programming model – basic concepts and data types

• CUDA application programming interface - basic

• Simple examples to illustrate basic concepts and functionalities

• Performance features will be covered later
CUDA – C with no shader limitations!

- Integrated host+device app C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD kernel C code

**Serial Code (host)**

**Parallel Kernel (device)**

```c
KernelA<<< nBlk, nTid >>>(args);
```

**Serial Code (host)**

**Parallel Kernel (device)**

```c
KernelB<<< nBlk, nTid >>>(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
  – Is typically a GPU but can also be another type of parallel processing device

• Data-parallel portions of an application are expressed as device kernels which run 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
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);
```
Extended C

Integrated source
(foo.cu)

cudacc
EDG C/C++ frontend
Open64 Global Optimizer

GPU Assembly
foo.s

OCG

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
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
Thread Blocks: Scalable Cooperation

- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks cannot cooperate
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 the allocated object

- **cudaFree()**
  - Frees object from device **Global Memory**
    - Pointer to freed object
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 transfer
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
CUDA Function Declarations (cont.)

• \texttt{__device__} functions cannot have their address taken

• For functions executed on the device:
  – No recursion
  – No static variable declarations inside the function
  – No variable number of arguments
Sample Problem: Matrix Multiply

• In this section, we take a simple problem from standard sequential computation and adapt it for optimal execution on a CUDA device.
• Let A, B, and C be N-by-N square matrices, with each index in the range [0, (N-1)].
• The original code uses a triple loop, so its time complexity is $O(N^3)$.
• Note the use of variable SUM to avoid multiple references to $C[I][J]$. 

The Sequential Code

For I = 0 to (N - 1) Do
    For J = 0 to (N - 1) Do
        Sum = 0 ;
        For K = 0 to (N - 1) Do
            SUM = SUM + A[I][K]•B[K][J] ;
        End For
        C[I][J] = SUM ;
    End For
End For
End For
Memory Layout of a Matrix in C

\[
\begin{array}{cccc}
M_{0,0} & M_{1,0} & M_{2,0} & M_{3,0} \\
M_{0,1} & M_{1,1} & M_{2,1} & M_{3,1} \\
M_{0,2} & M_{1,2} & M_{2,2} & M_{3,2} \\
M_{0,3} & M_{1,3} & M_{2,3} & M_{3,3} \\
\end{array}
\]
1D Representation of a 2D Array

• Assume a 2D array A[N][N] laid out in row major order, as above.
• The array can be accessed either as a 2D array or as a 1D array.
• The element A[I][J] is referenced in one dimension as A[I*N + J].
• This transformation is exactly what a modern compiler will do in handling the array access.
Multiplication with 1D Arrays

For I = 0 to (N - 1) Do
    For J = 0 to (N - 1) Do
        Sum = 0;
        For K = 0 to (N - 1) Do
            SUM = SUM + A[I\cdot N + K]\cdot B[K\cdot N + J] ;
        End For
        C[I\cdot N + J] = SUM ;
    End For
End For
Efficiency in Computing the Index

• Consider the statement
  \[ \text{SUM} = \text{SUM} + A[I\cdot N + K] \cdot B[K\cdot N + J] \]

• This involves two multiplications to generate the indices into the arrays A and B.

• In general, we want to avoid multiplication when there is a simpler approach that is obvious and easy to understand.

• We now evolve the more efficient algorithm.
Modifying the Index Calculation

This modification affects only the inner loop of the example code. The original code is

```
For K = 0 to (N - 1) Do
    SUM = SUM + A[I•N + K]•B[K•N + J] ;
End For
```

We now modify that code as follows

```
For K = 0 to (N - 1) Do
    L = I•N + K ;
    M = K•N + J ;
End For
```
Sequence of the Indices

• Here we watch L and M as K is incremented.

For K = 0 to (N - 1) Do
    \[ L = I \cdot N + K ; \]
    \[ M = K \cdot N + J ; \]
    \[ \text{SUM} = \text{SUM} + A[L] \cdot B[M] ; \]
End For

For K = 0 \hspace{1cm} L = I \cdot N \hspace{1cm} M = J

For K = 1 \hspace{1cm} L = I \cdot N + 1 \hspace{1cm} M = J + N

For K = 2 \hspace{1cm} L = I \cdot N + 2 \hspace{1cm} M = J + 2 \cdot N

For K = 3 \hspace{1cm} L = I \cdot N + 3 \hspace{1cm} M = J + 3 \cdot N
The Optimized Sequential Code

For I = 0 to (N - 1) Do
  For J = 0 to (N - 1) Do
    Sum = 0;
    L = I • N;
    M = J;
    For K = 0 to (N - 1) Do
      SUM = SUM + A[L] • B[M];
      L = L + 1;
      M = M + N;
    End For
    C[I • N + J] = SUM;
  End For
End For
A Square Array of Processors

- Processor \( P[I][J] \) handles array element \( C[I][J] \)

\[
\begin{align*}
\text{Sum} & = 0 ; \\
L & = I \cdot N ; \\
M & = J ; \\
\text{INJ} & = L + M ; \quad \text{// This is } I \cdot N + J. \\
\text{For } K = 0 \text{ to } (N - 1) \text{ Do} \\
\text{SUM} & = \text{SUM} + A[L] \cdot B[M] ; \\
L & = L + 1 ; \\
M & = M + N ; \\
\text{End For} \\
C[\text{INJ}] & = \text{SUM} ; \quad \text{// This is } C[I][J]
\end{align*}
\]
Block IDs and Thread IDs

- Each thread uses IDs to 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
  - …
Revised Matrix Multiplication Kernel using Multiple Blocks

```c
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width) {
    // Calculate the row index of the Pd element and M
    int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
    // Calculate the column index of Pd and N
    int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

    Pd[Row*Width+Col] = Pvalue;
}
```