Introduction to GPU (Graphics Processing Unit) Architecture & Programming

CS240A. 2017
T. Yang

Some of slides are from M. Hall of Utah
CS6235
Overview

• Hardware architecture
• Programming model
• Example
**FIGURE A.2.1 Historical PC.** VGA controller drives graphics display from framebuffer memory. Copyright © 2009 Elsevier, Inc. All rights reserved.
FIGURE A.2.2 Contemporary PCs with Intel and AMD CPUs. See Chapter 6 for an explanation of the components and interconnects in this figure. Copyright © 2009 Elsevier
GPU Evolution

Highly parallel, highly multithreaded multiprocessor optimized for graphic computing and other applications

- New GPU are being developed every 12 to 18 months
- **Number crunching**: 1 card ~= 1 teraflop ~= small cluster.

- 1980’s – No GPU. PC used VGA controller
- 1990’s – Add more function into VGA controller
- 1997 – 3D acceleration functions:
  - Hardware for triangle setup and rasterization
  - Texture mapping
  - Shading
- 2000 – A single chip graphics processor (beginning of GPU term)
- 2005 – Massively parallel programmable processors
GPU Programming API

- CUDA (Compute Unified Device Architecture): parallel GPU programming API created by NVIDIA
  - Hardware and software architecture for issuing and managing computations on GPU
- Massively parallel architecture. over 8000 threads is common
- API libraries with C/C++/Fortran language
  - Numerical libraries: cuBLAS, cuFFT,
- OpenGL – an open standard for GPU programming
- DirectX – a series of Microsoft multimedia programming interfaces
GPU Architecture

**SP**: scalar processor  
‘CUDA core’

Executes one thread

**SM**: streaming multiprocessor  
32xSP (or 16, 48 or more)

Fast local ‘shared memory’  
(shared between SPs)  
16 KiB (or 64 KiB)
• **GPU:**

- **SMs**
  - 30xSM on GT200,
  - 14xSM on Fermi
- **For example, GTX 480:**
  - 14 SMs x 32 cores
  = 448 cores on a GPU

**GDDR memory**
512 MiB - 6 GiB

**GLOBAL MEMORY (ON DEVICE)**

**HOST**

**SM**

- S S S S S
- S S S S S
- S S S S S
- S S S S S
- **SHARED MEMORY**
FIGURE A.2.5 Basic unified GPU architecture. Example GPU with 112 streaming processor (SP) cores organized in 14 streaming multiprocessors (SMs); the cores are highly multithreaded. It has the basic Tesla architecture of an NVIDIA GeForce 8800. The processors connect with four 64-bit-wide DRAM partitions via an interconnection network. Each SM has eight SP cores, two special function units (SFUs), instruction and constant caches, a multithreaded instruction unit, and a shared memory. Copyright © 2009 Elsevier, Inc. All rights reserved.
CUDA essentials

• developer.nvidia.com. Download
  ▪ Driver
  ▪ Toolkit (compiler nvcc)
  ▪ SDK (examples) (recommended)
  ▪ CUDA Programmers guide

Other tools:
• ‘Emulator’. Executes on CPU. Slow
• Simple profiler
• cuda-gdb (Linux)
How To Program For GPUs

- **Parallelization**
  - Decomposition to threads
- **Memory**
  - shared memory, global memory
- Enormous processing power
- **Thread communication**
  - Synchronization, no interdependencies
Application Thread blocks

- Threads grouped in thread blocks
  - 128, 192 or 256 threads in a block

- One thread block executes on one SM
  - All threads sharing the ‘shared memory’
  - 32 threads are executed simultaneously (‘warp’)

<table>
<thead>
<tr>
<th>BLOCK 1</th>
<th>BLOCK 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>THREAD (0,0)</td>
<td>THREAD (0,1)</td>
</tr>
<tr>
<td>THREAD (0,1)</td>
<td>THREAD (0,2)</td>
</tr>
<tr>
<td>THREAD (0,2)</td>
<td>THREAD (0,3)</td>
</tr>
<tr>
<td>THREAD (0,3)</td>
<td>THREAD (0,4)</td>
</tr>
<tr>
<td>THREAD (1,0)</td>
<td>THREAD (1,1)</td>
</tr>
<tr>
<td>THREAD (1,1)</td>
<td>THREAD (1,2)</td>
</tr>
<tr>
<td>THREAD (1,2)</td>
<td>THREAD (1,3)</td>
</tr>
<tr>
<td>THREAD (1,3)</td>
<td>THREAD (1,4)</td>
</tr>
<tr>
<td>THREAD (2,0)</td>
<td>THREAD (2,1)</td>
</tr>
<tr>
<td>THREAD (2,1)</td>
<td>THREAD (2,2)</td>
</tr>
<tr>
<td>THREAD (2,2)</td>
<td>THREAD (2,3)</td>
</tr>
<tr>
<td>THREAD (2,3)</td>
<td>THREAD (2,4)</td>
</tr>
</tbody>
</table>
Application Thread blocks

- Blocks execute on SMs
  - execute in parallel
  - execute independently!

- Blocks form a **GRID**
- **Thread ID**
  unique within block
- **Block ID**
  unique within grid
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
What Programmer Expresses in CUDA

- **Computation partitioning (where to run)**
  - Declarations on functions __host__, __global__, __device__
  - Mapping of thread programs to device: compute <<<gs, bs>>>(<args>)

- **Data partitioning (where does data reside, who may access it and how?)**
  - Declarations on data __shared__, __device__, __constant__, ...

- **Data management and orchestration**
  - Copying to/from host: e.g., cudaMemcpy(h_obj,d_obj, cudaMemcpyDeviceToHost)

- **Concurrency management**
  - E.g. __syncthreads()
**Code that executes on GPU: Kernels**

- **Kernel**
  - a simple C function
  - executes on GPU **in parallel**
    - as many times as there are threads
  - The keyword `__global__` tells the compiler `nvcc` to make a function a kernel (and compile/run it for the GPU, instead of the CPU)
  - It's the functions that you may call from the host side using CUDA kernel call semantics (`<<<...>>>`).

*Device functions* can only be called from other device or global functions. `__device__` functions cannot be called from host code.
Minimal Extensions to C + API

• Declspecs
  § global, device, shared, local, constant
  __device__ float filter[N];
  __global__ void convolve (float *image) {
    __shared__ float region[M]; ...

• Keywords
  § threadIdx, blockIdx
  region[threadIdx] = image[i];
  __syncthreads()
  ...
  image[j] = result;

• Intrinsics
  __syncthreads

• Runtime API
  § Memory, symbol, execution management
  // Allocate GPU memory
  void *myimage = cudaMalloc(bytes)

• Function launch
  // 100 blocks, 10 threads per block
  convolve<<<100, 10>>> (myimage);
Setup and data transfer

- **cudaMemcpy**
  - transfer data to and from GPU (global memory)
- **cudaMalloc**
  - Allocate memory on GPU (global memory)

- GPU is the ‘device’, CPU is the ‘host’

- Kernel call syntax
NVCC Compiler’s Role: Partition Code and Compile for Device

**mycode.cu**

```c
int main_data;
__shared__ int sdata;

Main() {}
__host__ hfunc () {
  int hdata;
  <<<gfunc(g,b,m)>>>();
}

__global__ gfunc() {
  int gdata;
}
__device__ dfunc() {
  int ddata;
}
```

Compiled by native compiler: gcc, icc, cc

Compiled by nvcc compiler

```c
int main_data;
__shared__ sdata;

Main() {}
__host__ hfunc () {
  int hdata;
  <<<gfunc(g,b,m)>>>();
}

__global__ gfunc() {
  int gdata;
}
__device__ dfunc() {
  int ddata;
}
```
CUDA Programming Model: How Threads are Executed

- 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
Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
    - Block ID: 1D or 2D (blockIdx.x, blockIdx.y)
    - Thread ID: 1D, 2D, or 3D (threadIdx.{x,y,z})
- Simplifies memory addressing when processing multidimensional data

Courtesy: NDVIA
Simple working code example

- **What does it do?**
  - Scan elements of array of numbers (any of 0 to 9)
  - How many times does “6” appear?
  - Array of 16 elements, each thread examines 4 elements, 1 block in grid, 1 grid

```plaintext
threadIdx.x = 0 examines in_array elements 0, 4, 8, 12
threadIdx.x = 1 examines in_array elements 1, 5, 9, 13
threadIdx.x = 2 examines in_array elements 2, 6, 10, 14
threadIdx.x = 3 examines in_array elements 3, 7, 11, 15
```

Known as a cyclic data distribution
MAIN PROGRAM:

Initialization

• Allocate memory on host for input and output
• Assign random numbers to input array

Call host function

Calculate final output from per-thread output

Print result

GLOBAL FUNCTION:

Thread scans subset of array elements
Call device function to compare with “6”

Compute local result

HOST FUNCTION:

Allocate memory on device for copy of input and output

Copy input to device

Set up grid/block

Call global function

Synchronize after completion

Copy device output to host

DEVICE FUNCTION:

Compare current element and “6”

Return 1 if same, else 0
Main Program: Preliminaries

**MAIN PROGRAM:**

**Initialization**
- Allocate memory on host for input and output
- Assign random numbers to input array

**Call host function**

**Calculate final output from per-thread output**

**Print result**

```c
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4

int main(int argc, char **argv)
{
    int *in_array, *out_array;
    ...
}
```
**MAIN PROGRAM:**

Initialization (OMIT)
- Allocate memory on host for input and output
- Assign random numbers to input array

Call *host* function

Calculate final output from per-thread output

Print result

```c
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4
__host__ void outer_compute (int *in_arr, int *out_arr);

int main(int argc, char **argv)
{
    int *in_array, *out_array;
    /* initialization */ ...
    outer_compute(in_array, out_array);
    ...
}
```
**Main Program: Calculate Output & Print Result**

**MAIN PROGRAM:**

Initialization (OMIT)
- Allocate memory on host for input and output
- Assign random numbers to input array

Call *host* function

Calculate final output from per-thread output

Print result

```c
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4
__host__ void outer_compute (int *in_arr, int *out_arr);

int main(int argc, char **argv)
{
    int *in_array, *out_array;
    int sum = 0;
    /* initialization */ …
    outer_compute(in_array, out_array);
    for (int i=0; i<BLOCKSIZE; i++) {
        sum+=out_array[i];
    }
    printf ("Result = %d\n",sum);
}
```
HOST FUNCTION:
Allocate memory on device for copy of input and output
Copy input to device
Set up grid/block
Call global function
Synchronize after completion
Copy device output to host

```c
__host__ void outer_compute (int *h_in_array, int *h_out_array) {

    int *d_in_array, *d_out_array;

    cudaMalloc((void **) &d_in_array, SIZE*sizeof(int));
    cudaMalloc((void **) &d_out_array, BLOCKSIZE*sizeof(int));

    ...
}
```
Host Function: Copy Data To/From Host

HOST FUNCTION:
Allocate memory on device for copy of **input** and **output**
Copy input to **device**
Set up grid/block
Call **global** function
Synchronize after completion
Copy **device** output to host

```c
__host__ void outer_compute (int *h_in_array, int *h_out_array) {
    int *d_in_array, *d_out_array;
    cudaMemcpy(d_in_array, h_in_array, SIZE*sizeof(int),
               cudaMemcpyHostToDevice);
    cudaMemcpy((void **) &d_in_array, SIZE*sizeof(int));
    cudaMemcpy((void **) &d_out_array, BLOCKSIZE*sizeof(int));
    cudaMemcpy(d_in_array, h_in_array, SIZE*sizeof(int),
               cudaMemcpyHostToDevice);
    cudaMemcpy(h_out_array, d_out_array, BLOCKSIZE*sizeof(int),
               cudaMemcpyDeviceToHost);
    ... do computation ...
}
```
HOST FUNCTION:
Allocate memory on device for copy of input and output
Copy input to device
Set up grid/block
Call global function
Synchronize after completion
Copy device output to host

__host__ void outer_compute (int *h_in_array, int *h_out_array) {
    int *d_in_array, *d_out_array;
    cudaMemcpy((void **) &d_in_array, SIZE*sizeof(int));
    cudaMemcpy((void **) &d_out_array, BLOCKSIZE*sizeof(int));
    cudaMemcpy(d_in_array, h_in_array, SIZE*sizeof(int), cudaMemcpyHostToDevice);
    compute<<<(1,BLOCKSIZE)>>>(d_in_array, d_out_array);
    cudaThreadSynchronize();
    cudaMemcpy(h_out_array, d_out_array, BLOCKSIZE*sizeof(int), cudaMemcpyDeviceToHost);
}

CS6235
GLOBAL FUNCTION:

Thread scans subset of array elements

Call `device` function to compare with “6”

Compute local result

```c
__global__ void compute(int *d_in, int *d_out) {
    d_out[threadIdx.x] = 0;
    for (int i=0; i<SIZE/BLOCKSIZE; i++) {
        int val = d_in[i*BLOCKSIZE + threadIdx.x];
        d_out[threadIdx.x] += compare(val, 6);
    }
}
```

threadIdx.x = 0 examines in_array elements 0, 4, 8, 12
threadIdx.x = 1 examines in_array elements 1, 5, 9, 13
threadIdx.x = 2 examines in_array elements 2, 6, 10, 14
threadIdx.x = 3 examines in_array elements 3, 7, 11, 15

Cyclic distribution
Device Function

**DEVICE FUNCTION:**

*Compare current element and “6”*

*Return 1 if same, else 0*

```c
__device__ int compare(int a, int b) {
    if (a == b) return 1;
    return 0;
}
```
Summary of Lecture

• **Introduction to CUDA:** C + API supporting heterogeneous data-parallel CPU+GPU execution
  - Computation partitioning
  - Data partitioning (parts of this implied by decomposition into threads)
  - Data organization and management
  - Concurrency management

• **Compiler nvcc takes as input a .cu program and produces**
  - C Code for host processor (CPU), compiled by native C compiler
  - Code for device processor (GPU), compiled by nvcc compiler