CUDA: Introduction

Christian Trefftz / Greg Wolffe
Grand Valley State University

Supercomputing 2008
Education Program

(modifications by Jernej Barbic)
Terms

What is GPGPU?
- General-Purpose computing on a Graphics Processing Unit
- Using graphic hardware for non-graphic computations

What is CUDA?
- Compute Unified Device Architecture
- Software architecture for managing data-parallel programming
Motivation

GFLOPS

G80 = GeForce 8800 GTX
G71 = GeForce 7900 GTX
G70 = GeForce 7800 GTX
NV40 = GeForce 6800 Ultra
NV35 = GeForce FX 5950 Ultra
NV30 = GeForce FX 5800

3.0 GHz Intel Core2 Duo
CPU vs. GPU

- **CPU**
  - Fast caches
  - Branching adaptability
  - High performance

- **GPU**
  - Multiple ALUs
  - Fast onboard memory
  - High throughput on parallel tasks
    - Executes program on each fragment/vertex

- CPUs are great for *task* parallelism
- GPUs are great for *data* parallelism
CPU vs. GPU - Hardware

- More transistors devoted to data processing
Traditional Graphics Pipeline

Vertex processing

⇒

Rasterizer

⇒

Fragment processing

⇒

Renderer (textures)
Pixel / Thread Processing

- Input Registers
  - Fragment Program
    - Texture
    - Constants
    - Registers
  - Output Registers
- Thread Number
  - Parallel Data Cache
    - Texture
    - Constants
    - Registers
  - Thread Program
  - Global Memory
Processing Element

- Processing element = thread processor
GPU Memory Architecture

Uncached:
- Registers
- Shared Memory
- Local Memory
- Global Memory

Cached:
- Constant Memory
- Texture Memory
Data-parallel Programming

- Think of the GPU as a massively-threaded co-processor
- Write “kernel” functions that execute on the device -- processing multiple data elements in parallel
- Keep it busy! ⇒ massive threading
- Keep your data close! ⇒ local memory
Hardware Requirements

- CUDA-capable video card
- Power supply
- Cooling
- PCI-Express
Acknowledgements

- NVidia Corporation
  developer.nvidia.com/CUDA

- NVidia
  Technical Brief – Architecture Overview
  CUDA Programming Guide

- ACM Queue
  http://www.acmqueue.org/
A Gentle Introduction to CUDA Programming
Credits

- The code used in this presentation is based on code available in:
  - the Tutorial on CUDA in Dr. Dobbs Journal
  - Andrew Bellenir’s code for matrix multiplication
  - Igor Majdandzic’s code for Voronoi diagrams
  - NVIDIA’s CUDA programming guide
Software Requirements/Tools

- CUDA device driver
- CUDA Toolkit (compiler, CUBLAS, CUFFT)
- CUDA Software Development Kit
  - Emulator

Profiling:

- Occupancy calculator
- Visual profiler
To compute, we need to:

- **Allocate** memory for the computation on the GPU (incl. variables)
- **Provide input data**
- **Specify the computation** to be performed
- **Read** the results from the GPU (output)
Initially:

- CPU Memory
- GPU Card’s Memory
Allocate Memory in the GPU card

array

Host’s Memory

array_d

GPU Card’s Memory
Copy content from the host’s memory to the GPU card memory

array
  Host’s Memory

array_d
  GPU Card’s Memory
Execute code on the GPU

- array
  - Host’s Memory

- array_d
  - GPU Card’s Memory

- GPU MPs
Copy results back to the host memory

array

Host’s Memory

array_d

GPU Card’s Memory
The Kernel

- The code to be executed in the stream processors on the GPU
- Simultaneous execution in several (perhaps all) stream processors on the GPU
- How is every instance of the kernel going to know which piece of data it is working on?
Grid and Block Size

- Grid size: The number of blocks
  - Can be 1 or 2-dimensional array of blocks

- Each block is divided into threads
  - Can be 1, 2, or 3-dimensional array of threads
Let’s look at a very simple example

The code has been divided into two files:
- simple.c
- simple.cu

simple.c is ordinary code in C

It allocates an array of integers, initializes it to values corresponding to the indices in the array and prints the array.

It calls a function that modifies the array

The array is printed again.
```c
#include <stdio.h>
#define SIZEOFARRAY 64
extern void fillArray(int *a, int size);

/* The main program */
int main(int argc, char *argv[]) {
    /* Declare the array that will be modified by the GPU */
    int a[SIZEOFARRAY];
    int i;
    /* Initialize the array to 0s */
    for (i = 0; i < SIZEOFARRAY; i++) {
        a[i] = 0;
    }
    /* Print the initial array */
    printf("Initial state of the array:\n");
    for (i = 0; i < SIZEOFARRAY; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
    /* Call the function that will in turn call the function in the GPU that will fill
    the array */
    fillArray(a, SIZEOFARRAY);
    /* Now print the array after calling fillArray */
    printf("Final state of the array:\n");
    for (i = 0; i < SIZEOFARRAY; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
    return 0;
}
```
simple.cu

- simple.cu contains two functions
  - fillArray(): A function that will be executed on the host and which takes care of:
    - Allocating variables in the global GPU memory
    - Copying the array from the host to the GPU memory
    - Setting the grid and block sizes
    - Invoking the kernel that is executed on the GPU
    - Copying the values back to the host memory
    - Freeing the GPU memory
fillArray (part 1)

#define BLOCK_SIZE 32
extern "C" void fillArray(int *array, int arraySize)
{
    int * array_d;
    cudaError_t result;

    /* cudaMalloc allocates space in GPU memory */
    result =
            cudaMemcpy((void**)&array_d,array,sizeof(int)*arraySize);

    /* copy the CPU array into the GPU array_d */
    result = cudaMemcpy(array_d,array,sizeof(int)*arraySize, cudaMemcpyHostToDevice);
fillArray (part 2)

/* Indicate block size */
dim3 dimblock(BLOCK_SIZE);
/* Indicate grid size */
dim3 dimgrid(arraySize / BLOCK_SIZE);

/* Call the kernel */
cu_fillArray<<<dimgrid, dimblock>>>(array_d);

/* Copy the results from GPU back to CPU memory */
result = cudaMemcpy(array, array_d, sizeof(int) * arraySize, cudaMemcpyDeviceToHost);

/* Release the GPU memory */
cudaFree(array_d);
}
simple.cu (cont.)

The other function in simple.cu is cu_fillArray():

- This is the GPU kernel
- Identified by the keyword: __global__
- Built-in variables:
  - blockIdx.x : block index within the grid
  - threadIdx.x: thread index within the block
__global__ void cu_fillArray(int * array_d)
{
    int x;
    x = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    array_d[x] = x;
}

__global__ void cu_addIntegers(int * array_d1, int * array_d2)
{
    int x;
    x = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    array_d1[x] += array_d2[x];
}
To compile:

- `nvcc simple.c simple.cu -o simple`
- The compiler generates the code for both the host and the GPU
- Demo on cuda.littlefe.net …
What are those blockIds and threadIds?

- With a minor modification to the code, we can print the blockIds and threadIds.
- We will use two arrays instead of just one.
  - One for the blockIds
  - One for the threadIds
- The code in the kernel:
  ```
  x = blockIdx.x * BLOCK_SIZE + threadIdx.x;
  block_d[x] = blockIdx.x;
  thread_d[x] = threadIdx.x;
  ```
In the GPU:

Processing Elements

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
</table>

Array Elements

Block 0

Block 1
Hands-on Activity

- Compile with (one single line)
  
  ```
  nvcc blockAndThread.c blockAndThread.cu
  -o blockAndThread
  ```

- Run the program
  ```
  ./blockAndThread
  ```

- Edit the file `blockAndThread.cu`

- Modify the constant BLOCK_SIZE. The current value is 8, try replacing it with 4.

- Recompile as above

- Run the program and compare the output with the previous run.
This can be extended to 2 dimensions

- See files:
  - blockAndThread2D.c
  - blockAndThread2D.cu

- The gist in the kernel
  
x = blockIdx.x * BLOCK_SIZE + threadIdx.x;
y = blockIdx.y * BLOCK_SIZE + threadIdx.y;
pos = x * sizeOfArray + y;
block_dX[pos] = blockIdx.x;

- Compile and run blockAndThread2D
  
  - nvcc blockAndThread2D.c blockAndThread2D.cu
  - o blockAndThread2D
  - ./blockAndThread2D
When the kernel is called:

dim3 dimblock(BLOCK_SIZE,BLOCK_SIZE);
nBlocks = arraySize/BLOCK_SIZE;
dim3 dimgrid(nBlocks,nBlocks);

cu_fillArray<<<dimgrid,dimbloc>>>(... params...);
Another Example: saxpy

- SAXPY (Scalar Alpha X Plus Y)
  - A common operation in linear algebra
- CUDA: loop iteration ⇒ thread
void saxpy_serial(int n,
    float alpha,
    float *x,
    float *y)
{
    for(int i = 0; i < n; i++)
        y[i] = alpha * x[i] + y[i];
}
__global__ void saxpy_parallel(int n, 
    float alpha, 
    float *x, 
    float *y) 
{
    int i = blockIdx.x*blockDim.x+threadIdx.x;
    if (i<n)
        y[i] = alpha*x[i] + y[i];
}
“Warps”

- Each block is split into SIMD groups of threads called "warps".

- Each warp contains the same number of threads, called the "warp size"
Keeping multiprocessors in mind...

- Each multiprocessor can process multiple blocks at a time.

- How many depends on the number of registers per thread and how much shared memory per block is required by a given kernel.

- If a block is too large, it will not fit into the resources of an MP.
Performance Tip: Block Size

- Critical for performance
- Recommended value is 192 or 256
- Maximum value is 512
- Should be a multiple of 32 since this is the warp size for Series 8 GPUs and thus the native execution size for multiprocessors
- Limited by number of registers on the MP
- Series 8 GPU MPAs have 8192 registers which are shared between all the threads on an MP
Performance Tip: Grid Size

- Recommended value is at least 100, but 1000 would scale for many generations of hardware

- Actual value depends on problem size

- It should be a multiple of the number of MPs for an even distribution of work (not a requirement though)

- Example: 24 blocks
  - Grid will work efficiently on Series 8 (12 MPs), but it will waste resources on new GPUs with 32MPs
Memory Alignment

- Memory access faster if data aligned at 64 byte boundaries

- Hence, allocate 2D arrays so that every row starts at a 64-byte boundary

- Tedious for a programmer
Allocating 2D arrays with “pitch”

- CUDA offers special versions of:

  - Memory allocation of 2D arrays so that every row is padded (if necessary): `cudaMallocPitch()`
  
  - Memory copy operations that take into account the pitch: `cudaMemcopy2D()`
Pitch

Rows

Columns

Pitch

Padding
A simple example:

- See pitch.cu
- A matrix of 30 rows and 10 columns
- The work is divided into 3 blocks of 10 rows:
  - Block size is 10
  - Grid size is 3
Key portions of the code (1)

```c
result = cudaMallocPitch(
    (void **) &devPtr,
    &pitch,
    width * sizeof(int),
    height);
```
result = cudaMemcpy2D(
    devPtr,
    pitch,
    mat,
    width*sizeof(int),
    width*sizeof(int),
    height,
    cudaMemcpyHostToDevice);
In the kernel:

```c
__global__ void myKernel(int *devPtr,
                        int pitch,
                        int width,
                        int height)
{
    int c;
    int thisRow;
    thisRow = blockIdx.x * 10 + threadIdx.x;
    int *row = (int *)((char *)devPtr +
                       thisRow*pitch);
    for(c = 0; c < width; c++)
        row[c] = row[c] + 1;
}
```
The call to the kernel

myKernel<<<3,10>>>(
    devPtr,
    pitch,
    width,
    height);


pitch ⇒ Divide work by rows

- Notice that when using pitch, we divide the work by rows.
- Instead of using the 2D decomposition of 2D blocks, we are dividing the 2D matrix into blocks of rows.
Dividing the work by blocks:

- Rows
- Columns
- Pitch

Block 0
Block 1
Block 2
An application that uses pitch: Mandelbrot

- The Mandelbrot set: A set of points in the complex plane, the boundary of which forms a fractal.
- A complex number, \( c \), is in the Mandelbrot set if, when starting with \( x_0 = 0 \) and applying the iteration

\[
x_{n+1} = x_n^2 + c
\]

repeatedly, the absolute value of \( x_n \) never exceeds a certain number (that number depends on \( c \)) however large \( n \) gets.
Performance Tip: Code Divergence

- Control flow instructions diverge (threads take different paths of execution)
- Example: if, for, while
- Diverged code prevents SIMD execution – it forces serial execution (kills efficiency)
- One approach is to invoke a simpler kernel multiple times
- Liberal use of __syncthreads()
Performance Tip: Memory Latency

- 4 clock cycles for each memory read/write plus additional 400-600 cycles for latency
- Memory latency can be hidden by keeping a large number of threads busy
- Keep number of threads per block (block size) and number of blocks per grid (grid size) as large as possible
- Constant memory can be used for constant data (variables that do not change).
- Constant memory is cached.
Performance Tip: Memory Reads

- Device is capable of reading a 32, 64 or 128-bit number from memory with a single instruction.
- Data has to be aligned in memory (this can be accomplished by using `cudaMallocPitch()` calls).
- If formatted properly, multiple threads from a warp can each receive a piece of memory with a single read instruction.
Watchdog timer

- OS may force programs using the GPU to time out if running too long

- Exceeding the limit can cause CUDA program failure.

- Possible solution: run CUDA on a GPU that is NOT attached to a display.
Resources on line

- “Computation of Voronoi diagrams using a graphics processing unit” by Igor Majdandzic et al. available through IEEE Digital Library, DOI: 10.1109/EIT.2008.4554342
A Real Application

- The Voronoi Diagram: A fundamental data structure in Computational Geometry
Definition

Definition: Let S be a set of n sites in Euclidean space of dimension d. For each site p of S, the Voronoi cell V(p) of p is the set of points that are closer to p than to other sites of S. The Voronoi diagram V(S) is the space partition induced by Voronoi cells.
The classical sequential algorithm has complexity $O(n \log n)$ where $n$ is the number of sites (seeds).

If one only needs an approximation, on a grid of points (e.g. digital display):

- Assign a different color to each seed
- Calculate the distance from every point in the grid to all seeds
- Color each point with the color of its closest seed
Lends itself to implementation on a GPU...

- The calculation for every pixel is a good candidate to be carried out in parallel...
- Notice that the locations of the seeds are read-only in the kernel
- Thus we can use the texture map area in the GPU card, which is a fast read-only cache to store the seeds:
  ```
  __device__ __constant__ ...
  ```