GPU Program Optimization
(CS 680)

Parallel Programming with CUDA*

Jeremy R. Johnson

*Parts of this lecture was derived from chapters 1-5 in Sanders & Kandrot, Wikipedia, and the CUDA C Programming Guide

Introduction

- **Objective: To introduce GPU Programming with CUDA**

- **Topics**
  - **GPU (Graphics Processing Unit)**
  - **Introduction to CUDA**
    - **Kernel functions**
    - **Compiling CUDA programs with nvcc**
    - **Copy to/from device memory**
    - **Blocks and threads**
    - **Global and shared memory**
    - **Synchronization**
  - **Parallel vector add**
  - **Parallel dot product**
  - **Float and Double (local GPU servers – 2 X GeForce GTX 580)**

GPUs

- Specialized processor designed for real-time high resolution 3D graphics
  - Compute intensive, data parallel tasks
- Highly parallel multi core systems that can operate on large blocks of data in parallel
- More effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel

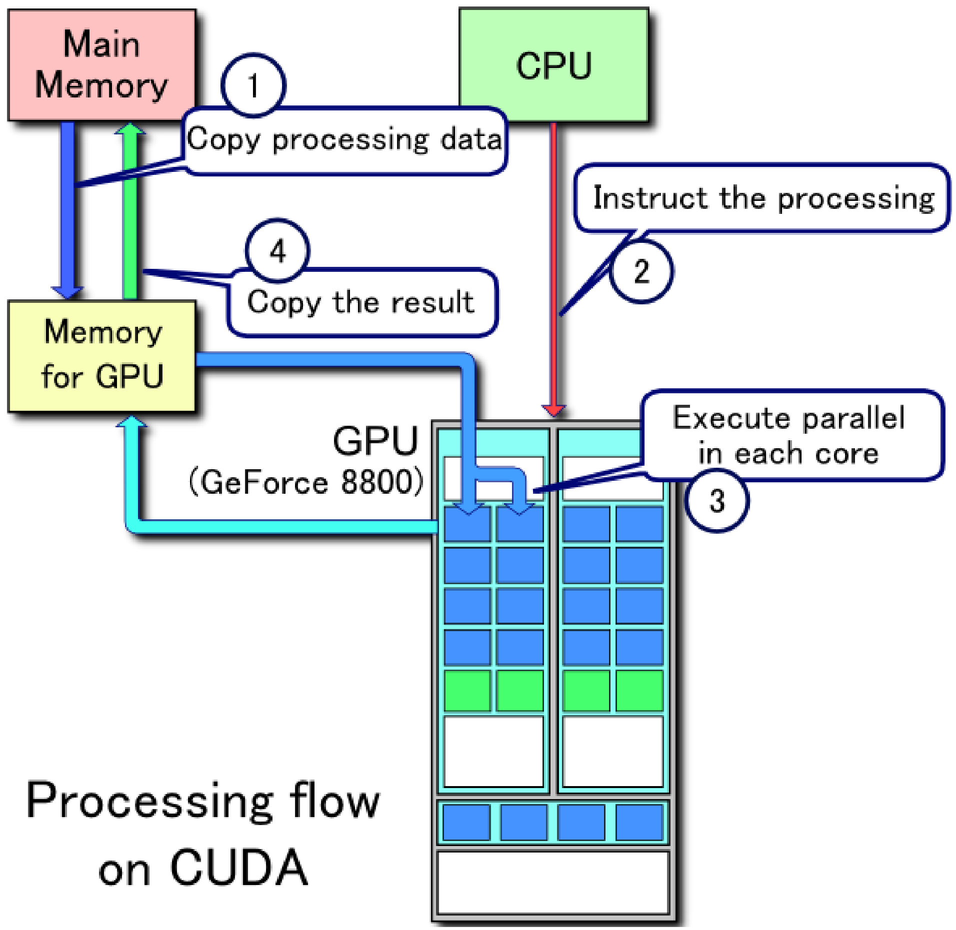
GTX 580

- **Third Generation Streaming Multiprocessor (SM)**
  - **32 CUDA cores per SM**
  - **16 SM for a total of 512 cores**
  - **Dual Warp Scheduler simultaneously schedules and dispatches instructions from two independent warps**
  - **64 KB of RAM with a configurable partitioning of shared memory and L1 cache**

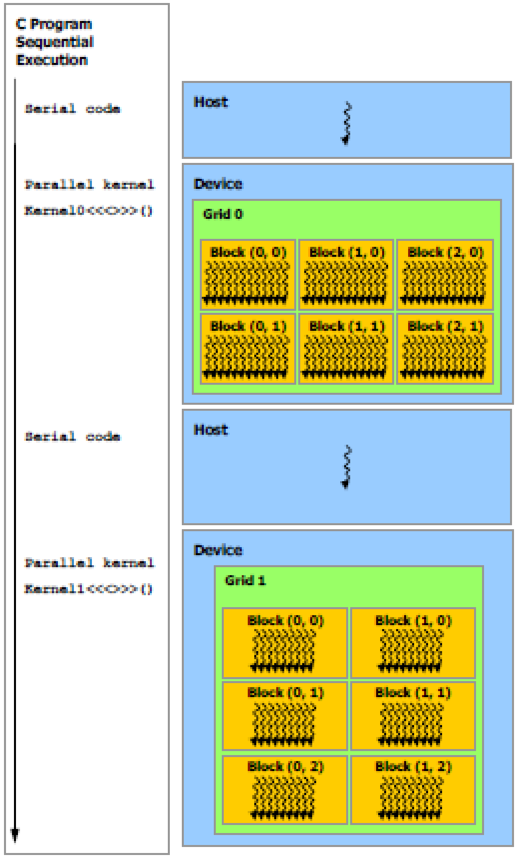
CUDA

- CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA
  - CUDA is the computing engine in Nvidia graphics processing units (GPUs)
  - CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs.
  - CUDA provides both a low level API and a higher level API
  - Minimal set of extensions to C (define kernels to run on device and specify thread setup)
  - Runtime library to allocate and transfer memory
  - Bindings for other languages available
- Must have a CUDA enabled device
- Install drivers and CUDA SDK

CUDA Model


Main Memory
CPU
1
Copy processing data
Instruct the processing
2
4
Copy the result
Memory for GPU
GPU
(GeForce 8800)
Execute parallel in each core
3
Processing flow on CUDA

Programming Model



Serial code executes on the host while parallel code executes on the device.

Figure 2-3. Heterogeneous Programming

CUDA Tool Chain

- **Input files *.cu contain mix of host and device code**

- **Compiling with nvcc**
  - **Separate host and device code**
  - **Compile device code to PTX (CUDA ISA) or cubin (CUDA binary)**
  - **Modify host code to replace <<<>>> notation to calls to the runtime library to load and launch compiled kernels**

- **cudart (runtime library)**
  - **C functions that execute on the host to allocate and deallocate device memory, transfer data between host memory and device memory, manage systems with multiple devices, etc.**

CS Department GPU Servers

- float.cs.drexel.edu   and   double.cs.drexel.edu
  - Dual Intel Xeon Processor (8 cores total) – 2.13 GHz
  - 12M Cache, 28 GB RAM
  - 2 X GeForce GTX 580 GPUs
- CUDA SDK
- /usr/local/cuda
  - bin (executables including nvcc)
  - doc (CUDA, nvcc, ptx, CUBLAS, CUFFT, OpenCL, etc.)
  - include
  - lib (32 bit libraries)
  - lib64 (64 bit libraries)
- /usr/local/NVIDIA_GPU_Computing_SDK

Using CUDA on float/double

- Make sure /usr/local/cuda/bin is in your path
  - PATH=$PATH:/usr/local/cuda/bin
  - export PATH
- Make sure /usr/local/cuda/lib64 is in your library path
  - LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/lib64
  - export LD_LIBRARY_PATH
- You should make these changes in your .bashrc file
  - If you edit .bashrc and want to use CUDA right away you need to reexecute .bashrc
  - source .bashrc
- Make sure this was done properly
  - which nvcc
  - echo $PATH
  - echo $LD_LIBRARY_PATH

Host Hello

```
#include <stdio.h>

int main( void ) {
    printf( "Hello, World!\n" );
    return 0;
}
```

To compile and run

```
[jjohnson@float chapter03]$ nvcc hello_world.cu -o hello_world
[jjohnson@float chapter03]$ ./hello_world
Hello, World!
```

GPU Hello

```
#include "../common/book.h"

#define N   15

__global__ void hello( char *dev_greetings) {
  dev_greetings[0] = 'h';  dev_greetings[1] = 'e';  dev_greetings[2] = 'l';
  dev_greetings[3] = 'l';  dev_greetings[4] = 'o';  dev_greetings[5] = ' ';
  dev_greetings[6] = 'f';  dev_greetings[7] = 'r';  dev_greetings[8] = 'o';
  dev_greetings[9] = 'm';  dev_greetings[10] = ' ';  dev_greetings[11] = 'g';
  dev_greetings[12] = 'p';  dev_greetings[13] = 'u';  dev_greetings[14] = '\0';
}
```

GPU Hello

```
int main( void ) {
    char greetings[N];
    char *dev_greetings;

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_greetings, N ) );
    hello<<<1,1>>>( dev_greetings);
    // copy the greetings back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( greetings, dev_greetings, N,
                              cudaMemcpyDeviceToHost ) );
    // display the results
    printf( "%s\n", greetings);
    // free the memory allocated on the GPU
    HANDLE_ERROR( cudaFree( dev_greetings ) );
    return 0;
}
```

CUDA Syntax & Functions

- **Function type qualifiers**
  - __global__
  - __device__
- **Calling a kernel function**
  - name<<<blocks,threads>>>()
- **Runtime library functions**
  - cudaMalloc
  - cudaMemcpy
  - cudaFree

CUDA Device Properties

```
#include "../common/book.h"

int main( void ) {
    cudaDeviceProp  prop;

    int count;
    HANDLE_ERROR( cudaGetDeviceCount( &count ) );
    for (int i=0; i< count; i++) {
        HANDLE_ERROR( cudaGetDeviceProperties( &prop, i ) );
        printf( "   --- General Information for device %d ---\n", i );
        printf( "Name:  %s\n", prop.name );
        printf( "Compute capability:  %d.%d\n", prop.major, prop.minor );
        printf( "Clock rate:  %d\n", prop.clockRate );
```

CUDA Device Properties (cont)

```
printf( "Device copy overlap:  " );
if (prop.deviceOverlap)
    printf( "Enabled\n" );
else
    printf( "Disabled\n");
printf( "Kernel execution timeout :  " );
if (prop.kernelExecTimeoutEnabled)
    printf( "Enabled\n" );
else
    printf( "Disabled\n" );
```

CUDA Device Properties (cont)

```
printf( "   --- Memory Information for device %d ---\n", i );
printf( "Total global mem:  %ld\n", prop.totalGlobalMem );
printf( "Total constant Mem:  %ld\n", prop.totalConstMem );
printf( "Max mem pitch:  %ld\n", prop.memPitch );
printf( "Texture Alignment:  %ld\n", prop.textureAlignment )
printf( "   --- MP Information for device %d ---\n", i );
printf( "Multiprocessor count:  %d\n", prop.multiProcessorCount );
printf( "Shared mem per mp:  %ld\n", prop.sharedMemPerBlock );
printf( "Registers per mp:  %d\n", prop.regsPerBlock );
printf( "Threads in warp:  %d\n", prop.warpSize );
printf( "Max threads per block:  %d\n", prop.maxThreadsPerBlock );
printf( "Max thread dimensions:  (%d, %d, %d)\n",
     prop.maxThreadsDim[0], prop.maxThreadsDim[1], prop.maxThreadsDim[2] );
printf( "Max grid dimensions:  (%d, %d, %d)\n",
      prop.maxGridSize[0], prop.maxGridSize[1], prop.maxGridSize[2] );
printf( "\n" ); }       }
```

Grid of Thread Blocks

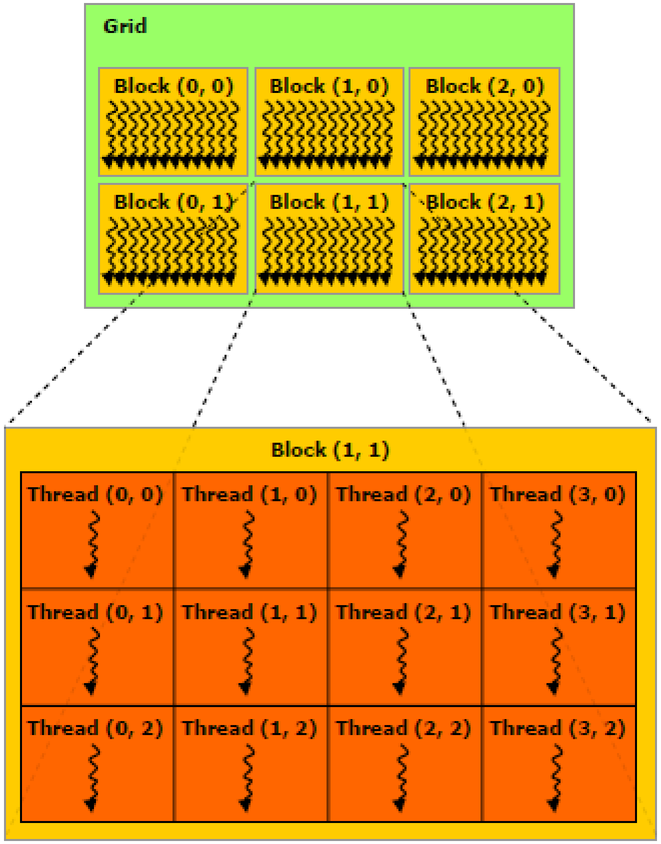


Figure 2-1. Grid of Thread Blocks

Parallel Vector Add (blocks)

```
#define N   10

__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;    // this thread handles the data at its thread id
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

Parallel Vector Add

```
int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );
    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;   b[i] = i * i;    }
    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int),
                              cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int),
                              cudaMemcpyHostToDevice ) );
```

Parallel Vector Add (blocks)

```
    add<<<N,1>>>( dev_a, dev_b, dev_c );

    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int),
                              cudaMemcpyDeviceToHost ) );
    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }
    // free the memory allocated on the GPU
    HANDLE_ERROR( cudaFree( dev_a ) );
    HANDLE_ERROR( cudaFree( dev_b ) );
    HANDLE_ERROR( cudaFree( dev_c ) );

    return 0;
}
```

Parallel Vector Add Long (blocks)

```
#define N   (32 * 1024)

__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += gridDim.x;
    }
}
 …

 add<<<128,1>>>( dev_a, dev_b, dev_c );
```

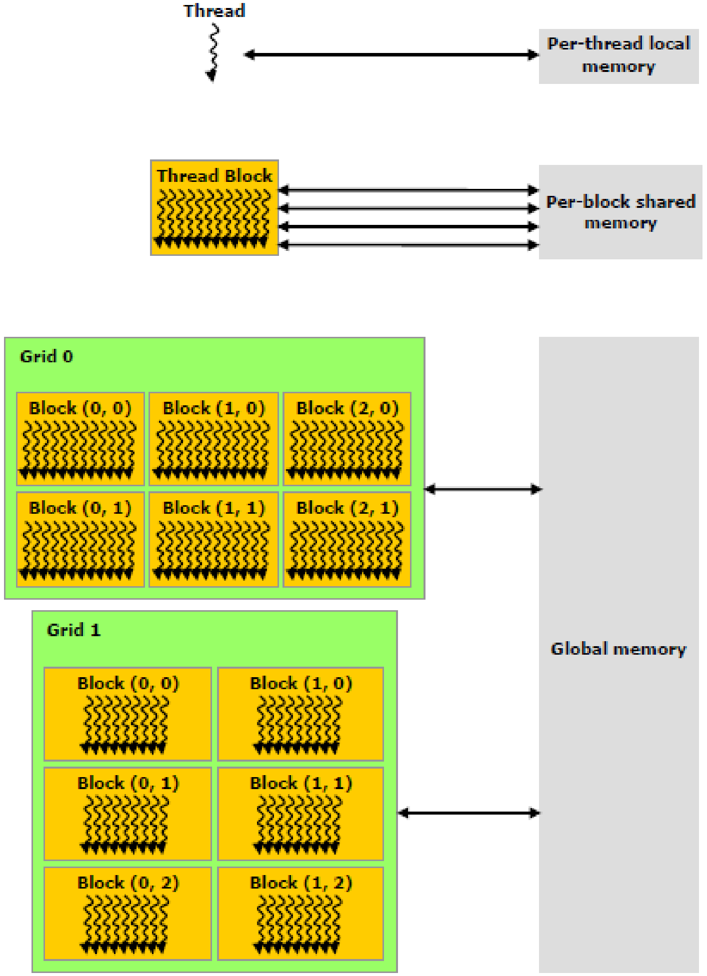
Parallel Vector Add Long (threads)

```
#define N   (33 * 1024)

__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}
 …

 add<<<128,128>>>( dev_a, dev_b, dev_c );
```

CUDA Builtin Variables

- **gridDim (number of blocks)**
- **blockIdx (block index within grid)**
- **blockDim (number of threads per block)**
- **threadIdx (thread index within block)**

Memory Hierarchy



Parallel Dot Product

```
__global__ void dot( float *a, float *b, float *c ) {
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    float   temp = 0;
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }
        // set the cache values
    cache[cacheIndex] = temp;
        // synchronize threads in this block
    __syncthreads();
    // for reductions, threadsPerBlock must be a power of 2
    // because of the following code
    int i = blockDim.x/2;
    while (i != 0) {
```

Parallel Dot Product

```
// for reductions, threadsPerBlock must be a power of 2
    // because of the following code
    int i = blockDim.x/2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
    }

    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}
```

Parallel Dot Product

```
int main( void ) {
    float   *a, *b, c, *partial_c;
    float   *dev_a, *dev_b, *dev_partial_c;

…

dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b, dev_partial_c );
…
    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c,
                            blocksPerGrid*sizeof(float), cudaMemcpyDeviceToHost ) );
    // finish up on the CPU side
    c = 0;
    for (int i=0; i<blocksPerGrid; i++) {
        c += partial_c[i];
    }
```

CUDA Shared Memory

- **Memory qualifier**
  - __shared__

- **Synchronization**

  __syncthreads();

Further Information

- http://en.wikipedia.org/wiki/CUDA
- http://developer.nvidia.com/category/zone/cuda-zone
- http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Getting_Started_Linux.pdf
- http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf
- http://www.nvidia.com/docs/IO/100940/GeForce_GTX_580_Datasheet.pdf

- **Textbook**
  - **Jason Sanders and Edward Kandro, CUDA by Example: An Introduction to General-Purpose GPU Programming , Addison-Wesley Professional, 2010.**