CUDA Study & Lab Report

Part 1. CUDA Study Notes

A. What is CUDA & Why we need it

Motivation
CUDA (Compute Unified Device Architecture) is a general purpose parallel computing architecture introduced by NVIDIA. It is designed to address problems that can be expressed as data-parallel computations with high arithmetic intensity. The motivation for introducing this new computing architecture is that traditional computing units have reached their limits in the way that power consumption is limiting the increase of CPU clock frequency and the level of productive activities that can be performed in each clock period within a single CPU. Facing both “power wall” and “instruction-level parallelism wall”, one possible solution is to take advantage of the computing capability of multi-core/many-core computing units and make program execution highly parallel. CUDA is one instance of architectures exploring data-level parallelism using many-core computing units (GPUs).

Where Does The Performance Gap Come From?
The large performance gap between many-core GPUs and general-purpose multi-core CPUs is due to the design philosophy between these two types of processors.

CPU:
- Optimized for sequential code performance; improvement in performance mainly comes from instruction-level parallelism (pipelining, out-of-order execution, branch prediction)
- More resources are used for cache
- Work as control processor ‡ More suitable for programs with irregular data structures, intensive branches, unpredictable memory accesses or recursive calls.

GPU:
- Optimized for execution of massive number of threads; Improvement in performance comes from data-level parallelism.
- Small cache memory; More resources are used for arithmetic calculation.
- Work as data processor ‡ More suitable for programs with predictable memory accesses and regular data structures.

Organization of GPU:
GPU is organized into 16 highly threaded Streaming Multiprocessors. Each SM has 8 Streaming Processors. Each SP has a Multi-Add unit and an additional multiply unit. Each GPU has about 1.5MB of DRAM.

How Does This New Architecture Affect Software Design?
Since hardware design is switching switches to multi-core and many-core models to seek new solutions to improve performance; consequently, software design is
switching from sequential programming to parallel programming. The applications software that will continue to enjoy performance improvement with each new generation of microprocessors will be parallel programs, in which multiple threads of execution cooperate to achieve the functionality faster. CUDA introduces an effective programming model for programmers to write highly parallel codes that will make good use of computing resources.

B. CUDA Programming Model

Heterogeneous Execution of Program
Despite of GPU’s high computing ability, it is specifically designed for numeric computing and thus may not be able to perform well on certain tasks as CPU (for instance, tasks involving complex control logic or irregular data structures). Thus, most applications will use both CPUs and GPUs (CPU for sequential parts and GPU for data-intensive parts). CUDA programming model is designed to support heterogeneous execution of program by running application alternatively on CPU and GPU.

Computing system architecture in CUDA programming model
Host + Device
Host: traditional CPU dealing with complicated control flow
Device: massively parallel processors equipped with a large number of arithmetic execution units for data-level parallelism

A CUDA program consists of one or more phases that are executed on either the host or a device. The phases that exhibit little data parallelism are implemented in host code, while the phases with rich amount of data parallelism are implemented in device code.

Compiling Model
Host code: straight C, compiled with the host’s standard C compilers
Device code: ANSI C extended with keywords for labeling data-parallel functions (kernels) and their associated data structures. Compiled by NVCC and executed on a GPU device.

Two Important Aspects of CUDA Programming Model
• Where to put the parallel executed code: Kernel Function
• How does device code access memory: CUDA memory hierarchy

C. CUDA Kernel Function

What does Kernel Function Do
Fine-grained, data-parallel threads are the fundamental means of parallel execution in CUDA. The kernel function specifies the code to be executed by all threads of a parallel phase. CUDA programming is an instance of SPMD programming style
(single program multiple data), which means all thread of a parallel phase execute the same code; this piece of code is implemented in kernels. When a kernel is launched, it is executed as grid of parallel threads. Each CUDA thread grid comprises thousands to millions of GPU threads. Compared with CPU thread, a GPU thread is much more light-weighted and is more efficiently assigned and scheduled.

Declaration of Kernel Function
Kernel functions are declared using CUDA keyword “__global__”

Organization and Specification of CUDA Thread
All threads in an execution grid execute the same kernel function and they rely on unique coordinates to distinguish themselves from each other and to identify the appropriate portion of data to process. Threads in a grid are organized into a two-level hierarchy:
-- Each grid consists of one or more thread blocks; all thread blocks must have the same number of threads organized in the same manner. Each thread block has a unique two dimensional coordinate (keywords blockId.x, blockId.y)
-- Each thread block is further organized as a three dimensional array of threads. The coordinates of threads in a block are defined by (threadId.x, threadId.y, threadId.z)

When launching a kernel function, we use <<<>>> to pass in special parameters that specify the organization of threads. The first special parameter (dimGrid) of a kernel launch specifies the grid in terms of number of blocks; the second (dimBlock) specifies the dimensions of each block in terms of number of threads. In situations where a kernel does not need one of the dimensions given, that field of dimension parameter can be initialized to 1.
KernelFunctionName <<<dimGrid, dimBlock>>> (func_param1, func_param2...)

One common usage for threadId and blockId is to determine the area of data that a thread is to work on. When the total number of threads are less than 512 (meaning that they can be organized in one grid), the kernel function does not use blockId. But when we need to process a large amount of data (for instance, a 1024 * 1024 matrix), then multiple blocks are used and we need blockId to specify the data portion of a certain thread.

Synchronization of CUDA Threads
CUDA allows threads in same block to coordinate their activities using a barrier synchronization function syncthreads(). When kernel function calls syncthreads(), all threads in a block will be held at the calling location until everyone else in the block reaches the location. This ensures that all threads in a block have completed a phase of their execution of kernel before they all move on to the next phase. Barrier synchronization is a popular method of coordinating parallel activities.
To avoid the situation that synchronization may result in long wait latency for some threads, all threads in the same block are assigned to the same computing resources (shared variable mechanism, which will be discussed later) to ensure the time proximity of all threads in a block and prevent excessive waiting time during barrier synchronization.

One thing to notice is that syncthreads() only works for threads in the same block. Threads in different blocks are not allowed for barrier synchronization. This means that CUDA does not to deal with any constraint while executing different blocks. Blocks can execute in any order relative to each other. The flexibility enables scalable implementations (the ability to execute the same application code at a wide range of speeds).

How Are CUDA Threads Scheduled
Each thread block is assigned to a Streaming Multiprocessor (SM) and further divided into 32-thread units called Warps. Warp is the unit of thread scheduling in SM. Each warp consists of 32 threads of consecutive threadId values. Since there can be at most 768 threads in each SM, at most 24 warps can reside in a SM. SMs are designed so that only one of these warps will be executed at any point of time. This is how SMs efficiently execute long latency operations such as access to global memory. When an instruction executed by threads in a warp needs to wait for the result of a previously initiated long-latency operation, the warp is placed into a waiting area so that the overall execution throughput will not be slowed down. This is similar to how traditional CPU deals with processes that need access to IO.

D. CUDA Memory
Separated Memory Space
Hosts and devices have separate memory spaces. In order to execute a kernel on a device, the programmer needs to allocate memory on the device and transfer the pertinent data from the host memory to the allocated device memory. After device execution, the programmer needs to transfer result data from device back to the host and free up the device memory that is no longer needed.

API functions for device memory allocation and data transfer:

Allocate and free GPU global memory
    cudaMalloc()
        parameters: the pointer to object array; the size of array
    cudaFree()
        parameter: the pointer to object array

Data transfer between host and device
    cudaMemcpy()
        parameters: pointer to destination
                    pointer to source
                    number of bytes copied
                    type of transfer
Device Memory Hierarchy:
The reason that CUDA needs memory hierarchy is in a way similar to that traditional CPU needs multiple levels of cache. Since access to global memory allocated on device is quite slow, we need a mechanism to filter out a majority of data requests to the global memory (which will result in long latencies).

Concerning device memory access, one heuristic of parallel program design would be to increase CGMA (computing to global memory access) thus to hide memory latency and achieve higher level of performance; Also, making good use of different types of CUDA memory is also important for efficient memory access.

CUDA memory types:
__shared__: declaration of a shared variable.
• Must reside within a kernel function (since it is related to a block of threads)
• All threads in a block see the same version of a shared variable.
• An efficient means for threads within a block to collaborate with each other.
• Extremely fast and highly parallel.
• Often used to hold the portion of global memory data that are heavily used in an execution phase of kernel function.

__constant__: declaration of constant variables.
• Must reside outside any function body.
• Have a scope of all grids: all threads in all grids see the same version of a constant variable.
• Often used for variables that provides input values to kernel functions.
• Extremely faster access speed, but limited size.

__device__: declaration of global variables (placed in global memory)
• Visible to all threads of all kernels.
• A means for threads to collaborate across blocks. Often used to pass information from one kernel execution to another kernel execution.
• Very slow access; programmers must make wise use of device memory hierarchy to reduce global memory traffic
• Common strategy of reducing global memory access: partition the data into subsets (tiles) so that each subset fits into the shared memory [remember that kernel computation on these data subsets must be done independently of each other].

E. A CUDA Program Template and a Comprehensive Example

Program template

```
main()
{
    //allocate memory on device
    Float * Md;
    cudaMalloc((void**)&Md, size);
```

//copy data from host to device
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
//call GPU kernel function
KernelFunc <<<dimGrid, dimBlock>>>(arguments);
//Copy data from device back to host
CopyFromDeviceMatrix(M, Md);
//free device memory
FreeDeviceMatrix(Md);
}

CUDA code example: Matrix Multiplication

//set tile size (tiles are used so that each data subset may fit into shared memory)
int TILE_SIZE = 16;

void runMatrixMulOnDevice(float* M1, float* M2, float* P, int Width)
{
    //matrix size
    int size = Width * Width * sizeof(float);

    //allocate device memory
    float *Md1, *Md2, *Pd;
    cudaMalloc(Md1, size);
    cudaMalloc(Md2, size);

    //load M1 and M2 to device memory
    cudaMemcpy(Md1, M1, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Md2, M2, size, cudaMemcpyHostToDevice);

    //declare special parameters
    //block size = tile_size * tile_size
    //grid_size = matrix_size / block_size = width/tile_size * width/tile_size
    dim3 dimBlock(TILE_SIZE, TILE_SIZE);
    dim3 dimGrid(Width/TILE_SIZE, Width/TILE_SIZE);

    //launch Kernel Function
    //pass in dimGrid and dimBlock as special parameters
    //pass in pointers to allocated device memory space as function arguments
    MatrixMulKernel <<<dimGrid, dimBlock>>> (Md1, Md2, Pd, Width);

    //Read Result (P) from device
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

    //free device memory
    cudaFree(Md1);
    cudaFree(Md2);
cudaFree(Pd);

//Kernel Function
__global__ void MatrixMulKernel(float* Md1, float* Md2, float* Pd, int Width) {

    //allocate share memory for data subsets of the two matrices
    __shared__ float Mds1[TILE_SIZE][TILE_SIZE];
    __shared__ float Mds2[TILE_SIZE][TILE_SIZE];

    //save the threadId and blockId values into thread registers for fast access
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    //identify the row and column of the Pd element to work on
    //each block covers tile_size elements in x dimension and tile_size elements in y dimension
    //a thread with bx and tx is responsible for covering the Pd element whose
    //x index is bx*tile_size + tx
    //similarly, a thread with by and ty is responsible for covering Pd element
    //whose y index is by*tile_size + ty
    //the above formulas are used to determine the row and column index for
    //the current Pd element
    int row = by * TILE_SIZE + ty;
    int col = bx * TILE_SIZE + tx;

    //iterate through all the phases of calculating the final product
    //total size of matrix is width * width, while block size is tile_size * tile_size
    //we need width/tile_size phases to get the final product
    int prod = 0;
    for (int m = 0; m < Width / TILE_SIZE; ++m) {
        //During each phase, a total number of tile_size * tile_size threads
        //collaborate to load tile_size * tile_size matrix elements into shared
        //memory

        //assign each thread to load one element
        //start point:
        //m * tile_size (elements already covered by previous phases)
        //thus in current phase, each thread just needs to load the element
        //identified by the threadId tx, ty (calculated as follows)
        Mds1[tx][ty] = Md1[m * TILE_SIZE + tx][row];
        Mds2[tx][ty] = Md2[col][m * TILE_SIZE + ty];
    }
}
/\texttt{calculate the dot product of the current phase in a loop}
for (int k = 0; k < TILE_SIZE; ++k)
{
    prod += Mds1[tx][k] * Mds2[k][ty];
}
Pd[row][col] = prod;

Part 2. Comparison of CUDA program and Standard CPU program
A. Benchmark Algorithm
1. Bitonic Sorting Network algorithm on which comparison will be made
   • Algorithm Description
     Definition from "Introduction to Algorithm":
     A comparison network is composed solely of wires and comparators. A comparator is a device with two inputs, $x$ and $y$, and two outputs, $x'$ and $y'$, that performs the following function: $x' = \min(x, y)$, $y' = \max(x, y)$. A sorting network is a comparison network for which the output sequence is monotonically increasing (that is, $b_1 \leq b_2 \leq \cdots \leq b_n$) for every input sequence.

   Example of Sorting Network:

   Parallel Execution:
   Notice that sorting network is perfect for parallel execution since comparators may calculate result at the same time as long as the inputs are ready. Because of this property, bitonic sort can be considered as a sorting algorithm designed specially for parallel machines.

   Zero-one principle (theoretical foundation for all sorting network algorithms):
   If a sorting network works correctly when each input is drawn from the set \{0, 1\}, then it works correctly on arbitrary input numbers. (The numbers can be integers, reals, or, in general, any set of values from any linearly ordered set.)
Bitonic Sorting Algorithm:
Bitonic sorting network is a comparison network that can sort any bitonic sequence: a sequence that monotonically increases and then monotonically decreases, or can be circularly shifted to become monotonically increasing and then monotonically decreasing.

- A Recursive Approach
Important property of bitonic sequence: a bitonic sequence of length 2n, elements in positions [0,2n), can be divided into two halves, [0,n) and [n,2n), such that: each half is a bitonic sequence, and every element in half [0,n) is less than or equal to (or greater than or equal to) each element in [n,2n).

Based on this property, a recursive algorithm can be designed to sort bitonic sequences (assume that the sequence is to be sorted in ascending order):

- If the number of elements in the sequence is larger than 1, divide the sequence in half; sort the lower half into non-decreasing order and the upper half into descending order (in this way we get a bitonic sequence)
- Merge the two subsequences in the following fashion: compare elements in the corresponding positions in the two halves and exchange them if they are out of order.
Recursively bitonically merge each half until all the elements are sorted; Or we can unroll the merge in an iterative fashion as follows:

Time Complexity (Depth) of Bitonic Sorting Network:

\[
T_{\text{Sort}}(n) = T_{\text{Sort}}(n/2) + T_{\text{Merge}}(n)
\]

\[
T_{\text{Merge}}(n) = T_{\text{Merge}}(n/2) + 1
\]

\[
T_{\text{Merge}}(n) = \log(n)
\]

\[
T_{\text{Sort}}(n) = \log(n)^2
\]

Each stage of the sorting network consists of \( \frac{n}{2} \) comparators. On the whole, these are \( O(n \cdot \log(n)^2) \) comparators.

B. Standard CPU Code

*Written in C*

```c
#define N 1048576
#define numValues 65536
int a[N];
const int ASCENDING = 1;
const int DESCENDING = 0;

// initialize n elements with random number
void init() {
    int i;
    for (i = 0; i < N; i++) {
        a[i] = rand() % numValues;
    }
}
```
// comparator function
// switch two elements if they are not in order “dir”
void compare(int i, int j, int dir)
{
    if (dir==a[i]>a[j])
    {
        int h=a[i];
        a[i]=a[j];
        a[j]=h;
    }
}

void bitonicMerge(int start, int stride, int order)
{
    if (stride > 1)
    {
        int k = stride/2;
        int i;
        // compare the k element pairs
        for (i = start; i < start + stride; i++)
            compare(i, i + stride, order);
        // recursively merge the two subsequences of length k
        bitonicMerge(start, stride, order);
        bitonicMerge(start + stride, stride, order);
    }
}

// core sort function
void bitonicSort(int start, int stride, int order)
{
    if (stride > 1)
    {
        int k = stride/2;
        // recursively sort the first subsequence in ascending order
        bitonicSort(start, stride, ASCENDING);
        // recursively sort the second subsequence in descending order
        bitonicSort(start + stride, stride, DESCENDING);
        // call bitonicMerge
        bitonicMerge(start, stride, order);
    }
}

// sort N elements in ascending order
int main(int argc, char **argv)
{
    int();
    bitonicSort(0, N ASCENDING);
}

Witten in Java (from http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonic.en.html)
public class BitonicSorter {
    private int[] a;
    // sorting direction:
    private final static boolean ASCENDING=true, DESCENDING=false;

    public void sort(int[] a) {
        this.a=a;
        bitonicSort(0, a.length, ASCENDING);
    }

    private void bitonicSort(int lo, int n, boolean dir) {
        if (n>1) {
            int m=n/2;
            bitonicSort(lo, m, ASCENDING);
            bitonicSort(lo+m, m, DESCENDING);
            bitonicMerge(lo, n, dir);
        }
    }

    private void bitonicMerge(int lo, int n, boolean dir) {
        if (n>1) {
            int m=n/2;
            for (int i=lo; i<lo+m; i++)
                compare(i, i+m, dir);
            bitonicMerge(lo, m, dir);
            bitonicMerge(lo+m, m, dir);
        }
    }

    private void compare(int i, int j, boolean dir) {
        if (dir==(a[i]>a[j]))
            exchange(i, j);
    }

    private void exchange(int i, int j) {
        int t=a[i];
        a[i]=a[j];
        a[j]=t;
    }
}

C. CUDA Code (kernel function)
Use the Bitonic project from CUDA SDK

__device__ inline void swap(int &a, int &b) {

// Alternative swap doesn't use a temporary register:
// a ^= b;
// b ^= a;
// a ^= b;

int tmp = a;
a = b;
b = tmp;

__global__ static void bitonicSort(int * values)
{
    // extern __global__ int shared[];

    const int tid = threadIdx.x;
    // Copy input to shared mem
    shared[tid] = values[tid];
    __syncthreads();

    // Parallel bitonic sort.
    for (int k = 2; k <= NUM && k *= 2)
    {
        // Bitonic merge:
        for (int j = k / 2; j > 0; j /= 2)
        {
            int ixj = tid ^ j;

            if (ixj > tid)
            {
                if ((tid & k) == 0)
                {
                    if (shared[tid] > shared[ixj])
                    {
                        swap(shared[tid], shared[ixj]);
                    }
                }
                else
                {
                    if (shared[tid] < shared[ixj])
                    {
                        swap(shared[tid], shared[ixj]);
                    }
                }
            }
            __syncthreads();
        }
    }
    // Write result.
    values[tid] = shared[tid];
}
D. Runtime Comparison

I run both programs on a machine with NVIDIA Quadro 290 (driver version 177.94). Since I don’t have hardware support on my own machine, I have to use computers in the lab and couldn’t update the driver; the consequence is that I have to use CUDA 1.1, which unfortunately is not compatible with Visual Studio 2008. It took me some time to solve all these configuration problems but at last got something running.

The bitonic sort program shown above run well when the number of elements to sort is small. When NUM exceeded 512, there occurred exception and I doubt it has everything to do with the use of shared memory.

The standard CPU program took 605ms to sort 256 elements and 728ms to sort 512 elements, while CUDA took only 10ms. But in fact most of the time was spent on allocating device memory: after I adjusted the timer and recorded only the time for actual sort, CUDA took only around 0.019 ms for both input size.

Then I run the matrix multiplication code shown in Section B. This time it worked a lot better since the stride is easier to figure out. Here is the comparison for standard CPU program and CUDA program for multiplying two matrices:

<table>
<thead>
<tr>
<th>Matrix size (A)</th>
<th>Matrix size (B)</th>
<th>CPU program (considering device memory allocation)</th>
<th>CUDA (excluding device memory allocation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 * 80</td>
<td>80 * 128</td>
<td>0.728352 ms</td>
<td>52.925819 ms</td>
</tr>
<tr>
<td>240 * 400</td>
<td>400 * 640</td>
<td>157.286072 ms</td>
<td>53.651299 ms</td>
</tr>
<tr>
<td>480 * 800</td>
<td>800 * 1280</td>
<td>2405.618896 ms</td>
<td>68.132011 ms</td>
</tr>
<tr>
<td>960 * 1600</td>
<td>1600 * 2560</td>
<td>22464.158203 ms</td>
<td>76.089073 ms</td>
</tr>
</tbody>
</table>

It can be observed that even though device memory allocation and memory copy between host and device take relatively a long time for a CUDA program, CUDA still has a huge advantage in total execution time compared with standard sequential program, especially when input size is large enough. Once memory operations are finished and data is ready on device, CUDA threads are extremely fast in calculating the partial results they are responsible for.