**CO471 - Parallel Programming**

# A3 - NVIDIA CUDA

**Contents**



**Note:** Login details to a server with a K40m GPU will be given out separately.

This document starts by explaining the anatomy of a simple CUDA vector addition program. A suggested practice would be to refer to the NVIDIA CUDA Runtime API reference manual[1] for better understanding of the parameters and the return values. A good starting point to learn CUDA is the CUDA C Programming Guide. Kirk and Hwu's books is a standard reference for programming NVIDIA GPUs using CUDA[2]. Udacity runs an excellent course on CUDA programming[3].

## 1 Vector Addition Example

The following example CUDA program adds two large vectors A and B and stores the sum vector in C. Each of A, B, C are present in the CPU memory. The brief steps in writing the CUDA version of this code are:

- Allocate space for A, B, and C in CPU memory. Populate A and B with random values.
- Allocate space for the vectors on the GPU using `cudaMalloc`. The following code allocates `vector_size` bytes in the GPU memory and returns the handle d_A.

  ```
  float *d_A;
  cudaMalloc((void **) &d_A, vector_size);
  ```

- Transfer the contents from the CPU to GPU using `cudaMemcpy`. Conventionally data stored in the device (GPU) memory are declared with a `d_` prefix and host memory variables are prefixed with a `h_`. The fourth parameter specifies the direction of copy.

  ```
  cudaMemcpy(d_A, h_A, vector_size, cudaMemcpyHostToDevice);
  ```

- The code segment that execute on the GPU is called the *kernel*. A kernel call is similar to the C call with one difference - the number of threads to spawned is specified within the angular brackets (<<<, >>>). The first parameter is the total number of threads blocks (called grid size) and the second parameter is the number of threads per thread block. All threads from a thread block run concurrently on one Streaming Multiprocessor (SM).

  ```
  vectoradd<<<vector_size/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_A, d_B, d_C, vector_size);
  ```

- The `vectoradd` kernel:
  - The kernel is declared with the `__global__` compiler directive. Kernel declaration:

    ```
    __global__ void addvectors(float *A, float *B, float *C, int N)
    ```

  - Thread identification:

    ```
    int block_id = blockIdx.x + gridDim.x * blockIdx.y;
    int thread_id = blockDim.x * block_id + threadIdx.x;
    ```

    The above code calculates the identity of the thread (`thread_id`) assuming the following:

* All the thread blocks are arranged in a 2 dimensional grid (in X and Y dimensions). The identity of each thread block is calculated by its position in the grid. `gridDim.x` is the total number of thread blocks in the X dimension. `blockIdx.x` and `blockIdx.y` are the positions of the blocks in the X and the Y dimensions respectively (recall the mesh figure from the MPI assignment). The block number (`block_id`) is calculated in the first line of the code segment. Chapter 4 - Cuda Threads from the Programming Massively Parallel Processors is a good reference to understand CUDA thread organization[2].
* Threads in one block are arranged in one dimension (X dimension only). The identity of a thread in a block is given by `threadIdx.x`.

To illustrate the previous code segment consider the following example: Lets say that a thread block contained 64 threads. There are 16 such thread blocks. The thread blocks themselves are arranged in the form of a 4×4 grid. Each block can be identified using a pair of block ids specifying their positions in the 2D grid. Block (2, 1) is the block in the third row and the second column in the grid. Block (0,0) is the top left block and Block (3,3) is the bottom right block. The block number = 2*4 + 1 = 9 (there are 4 blocks in the row). The thread with id 21 in Block (2,1) would be identified with the thread id 597 (9*64 + 21 = 597). Block (2,1) contains threads with ids starting from 576 (9*64 + 0) and ending up to 639 (9*64 + 63).

– add the corresponding array elements and update the sum array element.

```
C[thread_id] = A[thread_id] + B[thread_id];
```

You might want to add code to check if the above line does not cause an array out of bounds exception.

One way to check for errors is to encapsulate the cuda function call in the `checkCudaErrors()` call in the lines of the host program. Example:

```
checkCudaErrors(cudaMalloc(&d_A,   sizeof(float)*ARRAY_SIZE));
```

Another method is to check for errors after the function has been called:

```
cudaMalloc(&d_A,   sizeof(float)*ARRAY_SIZE);
checkCudaErrors(cudaGetLastError());
```

- Copy the results back to the host memory.

```
cudaMemcpy(h_C, d_C, vector_size, cudaMemcpyDeviceToHost);
```

- Before exiting, deallocate the arrays (from the host memory and from the device memory) that were allocated at the beginning of the program.

```
free(h_C);
cudaFree(d_C);
```

**Installation and Compilation**

Compile and run the Hello World progam.

If you have a NVIDIA GPU device on your laptop/desktop, download CUDA from the following link. The CUDA installer includes drivers as well as compilers and runtime libraries for running CUDA programs on your GPU - CUDA Download page. Compilation is done using the `nvcc` compiler.

```
$ nvcc -o helloworld ./helloworld.cu
$ ./helloworld
```

**Q1. Squares of Integers**

Squares of Integers

Write a program to update an array of squares of integers. Each thread calculates one square. Create 1024 threads in one thread block. Allocate a 1024 item `long long int` array in the GPU global memory. The kernel identifies the thread id. Calculate the square of the thread id and update in an array. Thread i updates the value $i^2$ in the array (`d_Square`). The central operation in the kernel:

```
d_Square[i] = i*i;
```

The host copies back the result array (using `cudaMemcpy()`) and prints the array.

**Q2. Device Query**

Device Query

Write a program to identify the working environment on your PC. It would be useful to know the amount of cores available, how much CPU/GPU memory is available and what types of capabilities the device has, etc. Get the number of CUDA enabled devices connected to your Motherboard. `cudaGetDeviceCount()`. Print out the following info for each device (all of these are available in `struct cudaDeviceProp` returned by `cudaGetDeviceProperties()` function):

- An ASCII string identifying the device
- The amount of global memory (GPU memory) on the device
- The maximum amount of shared memory a single block may use
- The number of 32-bit registers available per block
- The number of threads in a warp
- The maximum number of threads that a block may contain
- The maximum number of threads allowed along each dimension of a block
- The number of blocks allowed along each dimension of a grid
- The amount of available constant memory
- The major and minor revision of the device's compute capability
- The number of multiprocessors on the device
- Any other information you think is relevant and important.

**Q3. Vector Addition Program**

Vector Addition Program

Write a CUDA program to add two large vectors of size 1,000,000. Every thread updates one element of the array. Calculate the time taken for:

- Copying the 2 arrays from the host to the device
- Kernel execution
- Copying the result array from the device back to the host
- Total execution time.

Vary the thread block size from 16 to max threads per block. Plot the time taken by each of the previous events. Explain the trends. Identifying the effects of local, shared and global memory in thread execution would be good starting point to understand the trends. Refer Chapter 5 - CUDA Memories from [2].

**Q4. Synchronization operation in CUDA**

Synchronization operation in CUDA

Barrier Synchronization can be applied to threads in a thread block. The call `__syncthreads()` forces all threads in a block to synchronize (not applicable for threads across thread blocks).

In this program each program updates its own array element after reading from the next element. Achieve the following:

```
A[thread_id] = A[thread_id+1];
```

**Q5. SSSP**

SSSP

Implement the Single Source Shortest Path Algorithm using CUDA. Given a directed, weighted graph and a source vertex, compute the shortest path of each vertex from the source using Bellman-Ford Algorithm. For the input graph, use the SNAP's Wikipedia vote network. Implement two versions of the SSSP code.

1. Version 1 uses an adjacency matrix representation of the graph.
2. Version 2 uses the an adjacency list representation (use your favorite).

The main function reads a graph from the graph file into CPU's memory, copies the graph to GPU's memory, calls the GPU kernel and waits for it to finish computing the shortest paths. The final distances are then copied back from GPU to CPU.

You can explore the total blocks (and the grid dimensions) and the thread blocks size (and its dimensions) to arrive at the fastest implementation in both versions. Plot the runtimes of the kernel and the time taken to copy the input graphs from the host to device and the output from the device to the host.

**Q6. Boyer-Moore Algorithm**

Boyer-Moore Algorithm

Implement the Boyer-Moore algorithm [4] in CUDA. The standard C implementation is available in [4]. The page explains the algorithm, presents a reference C implementation and has a few important references related to the algorithm. Spend extra effor in optimizing your algorithm either for space or for time.

For the text string (T), generate a $10^7$ long string containing the characters A, C, G, T. For the pattern (P) to be searched, generate a random 100 character long string with the same alphabet.

**Q7. Sorting**

Sorting

Implement CUDA versions of the Quick sort and the Radix sort algorithms. Generate a random array of $10^6$ elements for input. The algorithms can be found in Chapters 7 and 8 in CLRS[5]. The parallel version of the algorithm is introduced in pages 42 – 45 in Blelloch and Maggs' chapter[] (download here).

**Q8. Blur an Image**

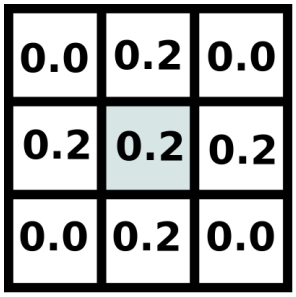
Blur an Image

Task to accomplish is to blur a BMP image. Read all about the bitmap file format. Use your favorite BMP image. Details follow:

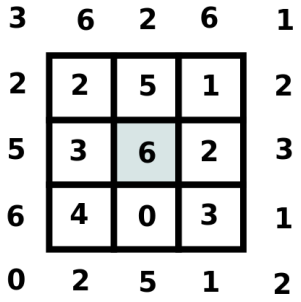
- Read the BMP file into your program. An example implementation for reading a bitmap in C is here[6]. A 24-bit/32-bit color BMP image contains 1 byte each of RGB values (ignore the Alpha channel (last byte) in a 32-bit image). After the BMP file is read, each pixel will be stored in a struct containg byte sized values of R, G, and B (W*H number of R values, W*H number of G values, and W*H number of B values - where W and H are the width and height of the BMP image, respectively).
- Separating out the R values, G values and B values from the image will simplify the averging task. There are now 3 arrays R[W*H], G[W*H], and B[W*H].
- Blurring operation is simply overwriting the current pixel value with an average of neighbouring pixel values. For a good blur effect, the averaging is done for individual colors separately (on individual items in the R, G, and B arrays). The averaging is done using an array of weights (also called the *filter*). The filter is applied on every pixel on the image (in other words - is overlayed on every pixel). Figure 1a shows a 3×3 filter to be overlayed on the centre pixel (the shaded pixel is to be averaged). To compute the blurred value, (a) multiply each weight with the pixel value underneath it, (b) Sum up the weighted values and replace the original pixel value with the newly computed average. For example, consider the array of pixel values in Figure 1b. Calculating the blur value:

$$0.0 \times 2 + 0.2 \times 5 + 0.0 \times 1 + 0.2 \times 3 + 0.2 \times 36 + 0.2 \times 2 + 0.0 \times 4 + 0.2 \times 0 + 0.0 \times 3 = 3.2$$

Repeat this process for all the pixels in the image (for each R, G, and B values). This blurring effect is called the *Gaussian Blur*.




(a) The filter to be overlayed on individual pixels to obtain the Gaussian blur value.




(b) An example array of pixels on which the filter is to be applied. The shaded value is the one to be replaced.

Figure 1: Illustrations for the BMP blur question(Q8.).

- Create the result BMP image. The output image can have the header identical to the original image. Use the blurred R, G, B values to populate the pixel values.

This problem is the same as the assignment problem at the end of module 2 from the Udacity course[3]. Refer to the course material for a better understanding of the problem and some startup code.

**Q9. Graph Coloring using GPUs**

Graph Coloring using GPUs

Implement a graph coloring algorithm. An example implementation is presented in this poster[7]. A journal version of the poster is [8]. You may use identical inputs as used in the papers.

**Epilogue**

The CUDA Fermi Compute Architecture whitepaper is an excellent source to understand GPU architecture[9]. The CUDA certification nexam link has a few informative lecture videos[10]. The CUDA by example book is an good starting point for CUDA programming[11].

**References**

[1] "NVIDIA CUDA Runtime API," http://docs.nvidia.com/cuda/cuda-runtime-api/#axzz3ssvPkVE2, NVIDIA Developer Zone.

[2] D. B. Kirk and W. mei W. Hwu, *Programming Massively Parallel Processors - A Hands-on Approach*, 2nd ed. Morgan Kaufmann.

[3] NVIDIA and Udacity, "Intro to Parallel Programming," https://www.udacity.com/wiki/cs344.

[4] T. Lecroq, "Boyer-Moore Algorithm," http://www-igm.univ-mlv.fr/~lecroq/string/node14.html.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[6] Stackoverflow, "Read Bitmap into a Structure," http://stackoverflow.com/questions/14279242/read-bitmap-file-into-structure.

[7] A. V. P. Grosset, P. Zhu, S. Liu, S. Venkatasubramanian, and M. Hall, "Evaluating graph coloring on gpus," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 297–298. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941597

[8] ——, "Evaluating graph coloring on gpus," *SIGPLAN Not.*, vol. 46, no. 8, pp. 297–298, 2011. [Online]. Available: http://doi.acm.org/10.1145/2038037.1941597

[9] "NVIDIA Fermi Compute Architecture," http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, nVIDIA Whitepaper.

[10] NVIDIA, "Syllabus for the CUDA Certification Exam," http://www.nvidia.com/object/io_1266605227307.html.

[11] J. Sanders and E. Kandrot, *CUDA by Example*. Addison Wesley, 2011.