**Assignments for GPU Programming Course (IT477)/ Lab**

**This document will be updated regularly.** All assignments have to be submitted by a given deadline (through moodle).

Assignments include submission of codes [optimized serial and parallel codes (multiple versions if applicable) with necessary comments inside the code]. **All the assignments** have to be supplemented with a **brief write-up with the following details** (wherever necessary):

1. Context:

* Brief description of the problem.
* Complexity of the algorithm (serial).
* Possible speedup (theoretical).
* Optimization strategy.
* Problems faced in parallelization and possible solutions.

1. Hardware details: GPU model, no of cores, device properties. Compute capability.
2. Input parameters. Output. Make sure results from serial and parallel are same.
3. Naïve implementation description. Possible improvements over naïve implementation.
4. Problem Size vs Time (Serial, parallel)**curve**.**Speedup curve.** Observations and comments about the results.
5. If more than one implementation, curves for all algorithms in the same plot.
6. Wherever necessary use log scale and auxiliary units.
7. Effect of block dimensions and grid launch on speedup.
8. Effect of tile size if you are using tiling.

**Assignment 1 (14th Aug) Deadline: 21st August**

Part-1 Device Query (all the details not to be included in report)

# Objective

# As discussed in the class, this is for introduction to the CUDA hardware resources along with their capabilities. Understand the process of compiling and running code (discussed in lectures) that will be used in subsequent modules.

# Instructions

The code provided below queries the GPU hardware on the system. Do not concentrate on the API calls, but on functions starting with wb.

The wbLog function logs results, specifically we log the following hardware features:

* GPU card's name
* GPU computation capabilities
* Maximum number of block dimensions
* Maximum number of grid dimensions
* Maximum size of GPU memory
* Amount of constant and share memory
* Warp size

# Questions

1. What is the compute capability of the NVIDIA architecture you are using?

The compute capability comprises a major revision number X and a minor revision number Y and is denoted by X.Y. Understand this.

ANSWER:

1. What are the maximum block dimensions for GPUs with this compute capability?

ANSWER:

1. Suppose you are launching a one dimensional grid and block. If the hardware's maximum grid dimension is 65535 and the maximum block dimension is 512, what is the maximum number threads can be launched on the GPU?
2. ANSWER:

(4)Is double precision supported on GPUs with the GPUs compute capability you are using?

ANSWER:

# Code Template

The following code is suggested as a starting point.

#include <stdio.h>

// Print device properties

void printDevProp(cudaDeviceProp devProp)

{

printf("Major revision number: %d\n", devProp.major);

printf("Minor revision number: %d\n", devProp.minor);

printf("Name: %s\n", devProp.name);

printf("Total global memory: %lu\n", devProp.totalGlobalMem);

printf("Total shared memory per block: %lu\n", devProp.sharedMemPerBlock);

printf("Total registers per block: %d\n", devProp.regsPerBlock);

printf("Warp size: %d\n", devProp.warpSize);

printf("Maximum memory pitch: %lu\n", devProp.memPitch);

printf("Maximum threads per block: %d\n", devProp.maxThreadsPerBlock);

for (int i = 0; i < 3; ++i)

printf("Maximum dimension %d of block: %d\n", i, devProp.maxThreadsDim[i]);

for (int i = 0; i < 3; ++i)

printf("Maximum dimension %d of grid: %d\n", i, devProp.maxGridSize[i]);

printf("Clock rate: %d\n", devProp.clockRate);

printf("Total constant memory: %lu\n", devProp.totalConstMem);

printf("Texture alignment: %lu\n", devProp.textureAlignment);

printf("Concurrent copy and execution: %s\n", (devProp.deviceOverlap ? "Yes" : "No"));

printf("Number of multiprocessors: %d\n", devProp.multiProcessorCount);

printf("Kernel execution timeout: %s\n", (devProp.kernelExecTimeoutEnabled ?"Yes" : "No"));

return;

}

int main()

{

int devCount;

cudaGetDeviceCount(&devCount);

printf("CUDA Device Query...\n");

printf("There are %d CUDA devices.\n", devCount);

for (int i = 0; i < devCount; ++i)

{

// Get device properties

printf("\nCUDA Device #%d\n", i);

cudaDeviceProp devProp;

cudaGetDeviceProperties(&devProp, i);

printDevProp(devProp);

}

return 0;

}

Part-2; CUDA Vector Add

# Objective

To implement vector addition by writing the GPU kernel code as well as the associated host code.

Before starting this make sure You have completed the "Device Query" part

# Instructions

Edit the code in the code tab to perform the following:

* Allocate device memory
* Copy host memory to device
* Initialize thread block and kernel grid dimensions
* Invoke CUDA kernel
* Copy results from device to host
* Free device memory
* Write the CUDA kernel

Instructions about where to place each part of the code is demarcated by the //@@ comment lines.

# Questions

1. How many floating operations are being performed in your vector add kernel? EXPLAIN.

ANSWER:

1. How many global memory reads are being performed by your kernel? EXPLAIN.

ANSWER:

(3) How many global memory writes are being performed by your kernel? EXPLAIN.

ANSWER:

(4)Describe what possible optimizations can be implemented to your kernel to achieve a performance speedup.

ANSWER:

1. Name three applications of vector addition.

ANSWER: Big Data analysis, statistics, and …………………….

# Code Template

The following code is suggested as a starting point.

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

// CUDA kernel. Each thread takes care of one element of c

\_\_global\_\_ void vecAdd(double \*a, double \*b, double \*c, int n)

{

// Get our global thread ID

int id = blockIdx.x\*blockDim.x+threadIdx.x;

// Make sure we do not go out of bounds

if (id < n)

c[id] = a[id] + b[id];

}

int main( int argc, char\* argv[] )

{

// Size of vectors

int n = 100000;

// Host input vectors

double \*h\_a;

double \*h\_b;

//Host output vector

double \*h\_c;

// Device input vectors

double \*d\_a;

double \*d\_b;

//Device output vector

double \*d\_c;

// Size, in bytes, of each vector

size\_t bytes = n\*sizeof(double);

// Allocate memory for each vector on host

h\_a = (double\*)malloc(bytes);

h\_b = (double\*)malloc(bytes);

h\_c = (double\*)malloc(bytes);

// Allocate memory for each vector on GPU

cudaMalloc(&d\_a, bytes);

cudaMalloc(&d\_b, bytes);

cudaMalloc(&d\_c, bytes);

int i;

// Initialize vectors on host

for( i = 0; i < n; i++ ) {

h\_a[i] = sin(i)\*sin(i);

h\_b[i] = cos(i)\*cos(i);

}

// Copy host vectors to device

cudaMemcpy( d\_a, h\_a, bytes, cudaMemcpyHostToDevice);

cudaMemcpy( d\_b, h\_b, bytes, cudaMemcpyHostToDevice);

int blockSize, gridSize;

// Number of threads in each thread block

blockSize = 1024;

// Number of thread blocks in grid

gridSize = (int)ceil((float)n/blockSize);

// Execute the kernel

vecAdd<<<gridSize, blockSize>>>(d\_a, d\_b, d\_c, n);

// Copy array back to host

cudaMemcpy( h\_c, d\_c, bytes, cudaMemcpyDeviceToHost );

// Sum up vector c and print result divided by n, this should equal 1 within error

double sum = 0;

for(i=0; i<n; i++)

sum += h\_c[i];

printf("final result: %f\n", sum/n);

// Release device memory

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_c);

// Release host memory

free(h\_a);

free(h\_b);

free(h\_c);

return 0;

}

**Part 3 (Linear algebra)**

1. Write a CUDA program (based on previous 2 parts) that adds a number X to all elements of a one-dimensional array A.
2. The elements of A and X should be single precision floating-point numbers.
3. Using the necessary timer calls, have your program report the time needed to copy data from the CPU to the GPU, the time needed to add X to all elements of A in the GPU, and the time needed to copy the data back from the GPU to the CPU.
4. The elements of A should be initialized with some value (not random).So that comparison with serial code is possible.
5. Vary the number of elements in A from min of 1Million to the maximum number that can be supported by single invocation of a GPU kernel in power of two steps, i.e., 1M, 2M, 4M, 16M, etc.
6. For every different array size, have your program print three time measurements: the time required to copy A from the CPU to the GPU, the time taken by the kernel, and the time required to copy the data from the GPU to the CPU.
7. The output should be reported in tabular form, like:

Elements(M) ; CPUtoGPU(ms) ; Kernel(ms) ; GPUtoCPU(ms)

Explore the following possibilities for profiling.

* cutStartTimer(myTimer)
* Events

1. Comment about CGMA ratio in the case of above program.
2. Extend your kernel specifying how many times X should be added to each element. Do not use multiplication for these additions. Create a loop.
3. For the maximum number of elements that can be supported by a single kernel invocation have your program print out the three time measurements above as a function of the number of times X is added. Do so, for a range of 1 through 256 in power of two steps. Your program’s output should look as follows:

XaddedTimes; Elements(M);CPUtoGPU(ms) ; Kernel(ms);GPUtoCPU(ms)

**What to submit/ report (part-3):**

* Submit the version that prints both measurements. (i.e., time as a function of element count and time as a function of the number of additions).
* Make sure it compiles and runs correctly.
* Presentation/write-up of around 10 pages/slides summarizing the 9 points (applicable points in page-1 for this assignment) and other observations.