National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Department of Computing

CS213: Advanced Programming

Class: BESE – 4B

Lab 7: Parallel Programming Patterns

Date: April 24th, 2016

Time: 10:00 AM to 12:50 PM

Instructor: Mr. Numair Khan

Lab 7: Parallel Programming Patterns

Introduction

Programming patterns are sequences of programmatic/algorithmic structures that recur frequently when writing efficient programs. Consider a program that prints all elements of an array. One approach, which you have probably used countless times, is to use a loop:

```
for (int j = 0; j < N; j++)
     fprintf(stdout, "%d", arr[j]);
```

An alternative approach is as follows:

```
fprintf(stdout, "%d", arr[0]);
fprintf(stdout, "%d", arr[1]);
fprintf(stdout, "%d", arr[2]);
fprintf(stdout, "%d", arr[3]);
…
```

You can see how difficult the latter piece of code can become to maintain (although, in this case, the *second* piece of code is probably more efficient from a performance point of view – we will study ***loop unrolling*** as an optimization technique later in the course). By choosing the appropriate algorithm and programming language structures (which, together, we may call a *pattern),* you can implement a much simpler, cleaner, and in most cases, efficient solution to the same problem. Recursion (which we will also study in detail later in the course) is another pattern. So, in short, a pattern is a boilerplate solution to commonly occurring programming problems.

The goal of this lab is to introduce students to *parallel* programming patterns that can be used in multi-threaded programs.

Objectives

After performing this lab students will be able to:

- Identify patterns in everyday programming tasks
- Implement parallel variants of common serial parallel programming patterns

Tools/Software Requirement

- NVIDIA CUDA Parallel Programming

Deadline

- The deadline for submitting Lab 7 is 11:55PM on Tuesday, 26th April.

Lab Task 1 – Map Pattern

The *Map* pattern is probably the simplest of all patterns to implement in parallel. A function (often called the *elemental function*) is applied to all elements of a data collection. The order in which the function is applied to elements of the collection does not affect the result.

Consider the example of finding the second power of each element of an array. In this case our elemental function will be `pow(x, 2)`. The serial implementation of the map pattern is given below

```
int array[N];

for(int j = 0; j < N; j++)

    array[j] = pow(array[j], 2);
```

The result may overwrite the original collection or be stored in a new location – it doesn't matter. The important thing to note is that the order in which the power function is applied to the elements of the array does not matter – `array[32]` may be calculated before `array[2]`. Also, the elemental function does not have any side-effects (look up what this means). If these two conditions for the elemental function are satisfied, you can apply the map pattern. (In a slightly complicated variant of map, a different elemental function may be applied to each element of the data collection. That's alright as long as it has no side-effects and its order of application does not alter the final result).

Your first task for this lab is to implement a CUDA version of the map pattern. Your kernel should have the following signature:

```
typedef float (*func) (float);

__global__ void map (float *data, float *result, int N, func e);
```

By now, you should be in a position to tell what is happening here.

An important thing to keep in mind is how much work each thread is doing. Remember that creating threads takes time and GPU resources. If your elemental function is very simple (which it usually is), what does that mean for how you divide the task of calculating the map pattern amongst the threads (Hint: tiles).

Lab Task 2 – Reduction Pattern

In the reduction pattern, a binary function called a *combiner* is used to combine all elements of a data collection into a single value. Here's a serial example of the reduce pattern where the combiner function is simple addition (+).

```
int array[N];

int sum = 0;

for(int j = 0; j < N; j++)

    sum = sum + array[j];
```

Another way to view the application of the binary combiner function is as follows:

$$\mathbf{R} = ((((((x_1 + x_2) + x_3) + x_4) + x_5) + x_6) + x_7)$$

In each instance the combiner (+) is being applied to two operands to reduce the elements into a single value **R**.

Implement the CUDA version of reduction. Your kernel should have the following signature:

```
typedef float (*func) (float, float);

__global__ void reduce (float *array, int N, float *result, func c);
```

Two important things to bear in mind as you implement your kernels are 1. How much thread divergence is there in each warp? and 2. How many threads per warp remain idle?

Lab Task 3

Slightly modify the map and reduction kernels you implemented above to calculate the dot product of two vectors of sizes 4, 512, 2048, 16384, 65536, 524288, 1048576. Which part of calculating the dot product is best handled by map and which by reduction?

Note the running time (this includes the time it takes to transfer data to the GPU's global memory) in each case. Write a sequential dot product calculation function. Compare its running time to your parallel version and note the speedup (if any) you get from the parallel one for each vector size. Graph your results and provide a brief explanation of the trend you see.

Deliverables

1. Your CUDA source code
2. A PDF or Word file with the speedup graph and your explanation of the trend in the graph.