CS3210 – Parallel Computing (AY 2015/2016 Sem 1)

Assignment 1 – Performance Improvements and Speedup Analysis (15 marks)

Individual Submission due on 4 October 2015

This assignment, covering lectures 1 to 4, is designed to enhance your understanding of GPU performance and speedup analysis. In question 1, you will convert a naive matrix multiplication (MM) program into a CUDA program, and based on your knowledge of MM and the GPU organization you will work towards reducing the execution time. The second question focuses on fixed-workload and fixed-time performance speedup. All timing measurements submitted for this assignment must be collected on the three computer nodes in our lab. For question 1, you can develop the CUDA program offline on your own notebook or PC but the results submitted must be measured on the Jetson TK1 system in our lab.

[*Optional Performance Challenge:* For Question 1, besides the above, you are encouraged to explore other forms of optimization and bonus marks will be awarded. This is an opportunity to score more than full marks.]

1. Performance Improvements on Jetson TK1 GPU (8 marks)

Figure 1 is a C program (*mm-seq.c)* that computes the product of two square matrices with single precision *float* data types. The program executes the $O(n^3)$ matrix multiplication algorithm, using three for loops:

```
for (inti = 0; i< n ; i++)
   for (int j = 0; j < n; j++)
      for (int k = 0; k < n; k++)
          result[i][j]  += a[i][k] * b[k][j];
```

Figure 1: Naïve Matrix Multiplication Program

You task is to write the MM in CUDA for execution on the Jetson TK1 systems in our lab. You are required to start from the given sequential program, *mm-seq.c* and the CUDA version submitted will be named mm-cuda.cu. Your CUDA program should allocate and initialize the two square matrices on the Jetson CPU. Run mm-seq.c on the Jetson CPU to obtain the sequential execution time and mm-cuda.cu to obtain the GPU execution time. Derive the speedup and explain your optimizations (marks will not be awarded if the performance results obtained are not explained). The matrix size must be at least 512x512.

Marks:

- up to 6 marks are awarded if you reduce the execution time by at least **5 times**.
- up to 8 marks are awarded if you reduce the execution time by at least **10 times**.

Bonus:

- 2 bonus mark for the two best submissions in terms of performance subject to reducing the execution time beyond that for 8 marks.

2. Performance Evaluation (7 marks)

The aim of this assignment is to carry out a simple performance analysis of a shared-memory program on a multicore system. Assume we use the naïve matrix multiplication algorithm with three for loops as in Figure 1, where *mm-seq.c* is a sequential program and *mm-omp.c* is a shared-memory program implemented using OpenMP.

a. Run these two programs on node 1 in our lab (Dell Optiplex tower system with 8 cores), measure the program execution time and complete the table below:

number of cores	execution time	speedup	
		using T_1(sequential)	using T_1(parallel)
1	T_1(sequential), T_1(parallel)		
2	T_2		
4	T_4		
6	T_6		
8	T_8		

Set the matrix size to 4096 and partitioned the program into 64 threads (see instructions on page 4). T_1(sequential) is the execution time of mm-seq.c on one core. T_1(parallel) is the execution time of *mm-omp.c* (partitioned into 64 threads) on one core. T_n (n=2,4,6,8) is the execution time of *mm-omp.c* (partitioned into 64 threads) on n cores. [See instructions on page 4 on how to run on one core or a group of cores]

b. **Execution time:** Is there any difference between T_1(sequential) and T_1(parallel)? Explain your answer.

c. **Speedup:** Comment on the speedups obtained using T_1(sequential) and T_1(parallel).

d. **Number of Threads**: For the shared-memory program, what should be an appropriate number of threads if we use eight cores? Devise an experiment to determine the number of threads. Explain your answer and assumptions if any.

e. **Overhead of Threading**: Devise an experiment to determine the average threading overhead in terms of overhead time per thread.

Deliverables and Special Considerations

You must hand in one CUDA source file, *mm-cuda.cu*, with the CUDA version of the matrix multiplication. In your submission, remove program code such as printing statements that you used for debugging your programs. You must either provide a *Makefile* or detailed compilation instructions in a *README* file. The source files must be properly commented and indexed, and must contain readable and well-modularized code. There is a **penalty of 1 mark** if these criteria are not met.

You must also prepare a report in PDF format that documents the following:
1. Details about your CUDA program (how it works)
2. Any special consideration or implementation detail that you consider non-trivial.
3. Answers for Question 2 with supporting arguments and data.
4. Details on how to reproduce your experiments (for both Question 1 and 2), for example, **the matrix size used MM, how to measure execution time and to compute speedup, etc.**

There is a **penalty of 1 mark** for not including any of these aspects.

Instructions for Compilation & Execution of Programs

To compile the sequential matrix multiplication program, you can use the *Makefile* provided by us, or use the following command:

```
$ gcc -O3 mm-seq.c -o mm-seq -lrt
```

For your optimized version, you can start with the following CUDA compilation command:

```
$ nvcc –arch=sm_32 mm-cuda.cu –o mm-cuda –lcuda -lcudart
```

- `-arch=sm_32` – specify for which GPU architecture to generate the code; in our case, Jetson TK1 has a Kepler GPU with compute capability 3.2.
- `-lcuda –lcudart` – link with CUDA libraries.

You may use other compilation flags, as you see fit.

To run the sequential MM:

```
$ ./mm-seq <matrice-size>
```

To compile the OpenMP matrix multiplication program, you can use the *Makefile* provided by us, or use the following command:

```
$ gcc -O3 -fopenmp mm-omp.c -o mm-omp
```

To run the OpenMP MM:

```
$ ./mm-omp <matrice-size> <number-of-threads>
```

For example, to run the OpenMP MM with matrice size 4096 and partitioned into 64 threads:

```
$ ./mm-omp 4096 64
```

To run the program *mm-omp* on one core (for example, core 3):

```
$ numactl --physcpubind=3 ./mm-omp <matrice-size>
<number-of-threads>
```

To run the program *mm-omp* on four cores (from core 0 to core 3):

```
$ numactl --physcpubind=0-3 ./mm-omp <matrice-size>
<number-of-threads>
```

Instructions for IVLE Submission and Deadline

1. This assignment is to be done on an individual basis. You can discuss the assignment with others as much as is necessary but in the case of plagiarism both parties will be severely penalized.
2. Submit your answers in the form of a Zip file consisting of (1) a PDF for your report, (2) the source file of the optimized MM, (3) a makefile or complication instructions, and (4) other supporting materials (if any).
   Please name your submitted ZIP file as follows: **nusnetid-assignment-1.zip**

3. The archive must be uploaded to IVLE in the workbin folder "Assigment1" by **4 October 2015 23:59**. There is a penalty of 1 mark per day for late submissions.

References

1. CUDA C Programming Guide, http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3gg5Zo0w3

2. CUDA Runtime API, http://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf

3. CUDA samples (in Ubuntu, find them under `/usr/local/cuda/samples`)

Programs (see course webpage)

The programs, `mm-seq.c`, `mm-omp.c` and `mm-cuda.cu`, multiply two square matrices using three for loops in C, OpenMP and CUDA respectively.

---------- END ----------