CS3210 – Parallel Computing (AY 2016/2017 Sem 1)

Assignment 1
Performance Improvements and Speedup Analysis (15 marks)

Individual Submission due on **30 September 2016**

This assignment, covering lectures 1 to 4, is designed to enhance your understanding of GPU performance and speedup analysis.

A simple introduction to CUDA is given in the next section, followed by the two main questions for this assignment. In question 1, you are given a GPU-enabled matrix multiplication program written in CUDA (Compute Unified Device Architecture). Using your knowledge of memory access and the GPU organization, you will work towards reducing the execution time.

Question 2 focuses on fixed-workload and fixed-time performance speedup.

All timing measurements submitted for this assignment must be collected on the 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 node 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.]

Introduction to GPU Parallel Programming using CUDA

In this section, we will run a "Hello world" CUDA parallel program on the Jetson node. You can find more about GPU architecture in Appendix A.1 and CUDA programming in Appendix A.2.

Note: Files for this assignment can be found on IVLE workbin, or directly downloaded via `wget`:

```
$ wget www.comp.nus.edu.sg/~sooyj/cs3210/As1/<filename>
```

e.g.

```
$ wget www.comp.nus.edu.sg/~sooyj/cs3210/As1/mm-cuda.cu
```

The "Hello World" CUDA program transforms the initial string "hello@world" into "HELLO WORLD" using the GPU.

Here is the CUDA kernel for this program:

Cuda: **hello-cuda.cu**

```
__global__ void hello(char *a, int len)
{    int tid = threadIdx.x;
     if (tid >= len)
         return;
     a[tid] += 'A' - 'a';
}
```

To compile the code:

```
$ nvcc hello-cuda.cu –o hello-cuda –arch=sm_32
```

To run the program:

```
$ ./hello-cuda
```

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

You need to use the **Jetson TK1** node for this question. You can ssh into the node from either ***node1*** or ***node2***.

Compilation and Execution

Get the "**mm-seq.c**" and "**mm-cuda.cu**" from the web:

```
$ wget www.comp.nus.edu.sg/~sooyj/cs3210/As1/mm-seq.c
$ wget www.comp.nus.edu.sg/~sooyj/cs3210/As1/mm-cuda.cu
```

Compile the sequential version:

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

Run the sequential version with any matrix size, e.g. 512

```
$ ./mm-seq 512
```

Take note of the sequential timing $T_{seq}(512)$.

Compile the GPU (CUDA) version:

```
$ 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.

Run the cuda version with the same matrix size, e.g. 512

```
$ ./mm-cuda 512
```

Take note of the parallel execution time $T_{par}(512)$.

Your Task

- Optimize the cuda version for better execution time.
- Derive the speedup achieved and explain your optimizations. **Marks will not be awarded if the performance results obtained are not explained**.

Note:

- Use a matrix size of **at least 512 x 512.**
- Your CUDA program should allocate and initialize the two square matrices on the Jetson CPU.

Evaluation

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.

Question 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. We are going to reuse the **`mm-seq.c`** and **`mm-omp.c`** programs from lab 3.

Compilation and Execution

The mm-seq.c can be compiled and executed in the same way as question 1.

To compile the OpenMP verison:

```
$ 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 with 64 threads:

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

You can control the processor core(s) used in the execution via the **numactl** command.

For example, to run the program *mm-omp* on core 3:

```
$ numactl --physcpubind=3 ./mm-omp 4096 64
```

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

```
$ numactl --physcpubind=0-3 ./mm-omp 4096 64
```

Your Tasks

a. Run the two programs on **node 1** in our lab (Dell Optiplex tower system with Intel i7 8-core), measure the program execution time for a **matrix size of 4096** 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		

Note:

- **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`** with **64 threads** on **one core.**
- **T_n (n=2,4,6,8)** is the execution time of **`mm-omp.c`** with **64 threads** on **n 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 Submission

Prepare a **zip file** using your **student id (matric no), e.g.** A01234567X.zip, which contains:

`mm-cuda.cu`	Your optimized version with appropriate comments, modularization and indentation. Please **remove all debug messages**.
`makefile / readme`	For compiling your `mm-cuda.cu`.
`as1_report.pdf`	Contains: 1. Details about your CUDA program (how it works). **Include details on how to reproduce your result, e.g. matrix size, execution time measurement etc.** 2. Any special consideration or implementation detail that you consider non-trivial. 3. Answers for Question 2 with supporting arguments and data. Provide additional information for us to reproduce your experiment, e.g. **the matrix size used MM, how to measure execution time and to compute speedup, etc.**

There is a **penalty of up to 3 marks** for various violation of specification.

1. This assignment is to be done on an individual basis. You can discuss the assignment with others as necessary but in the case of plagiarism both parties will be severely penalized.

2. The zip archive must be uploaded to IVLE in the workbin folder "Assignment1" by **30th September 23:59. Penalty of 50% per day for late submissions will be applied.**

References

1. Modern GPU Architecture, ftp://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf

2. Jetson TK1, http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html

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

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

5. CUDA samples (in Jetson, find them under `/usr/local/cuda/samples`)

Appendix A – GPUs and CUDA programming

GPU architecture

A modern GPU consists of multiple Streaming Multiprocessors (SMs), memory and cache, and the connecting interface (usually PCI Express) as shown on the bottom left.

An SM consists of multiple compute cores, memories (registers, L1 cache, texture memory, and shared memory) and the logic for thread and instruction management, as shown on the right.

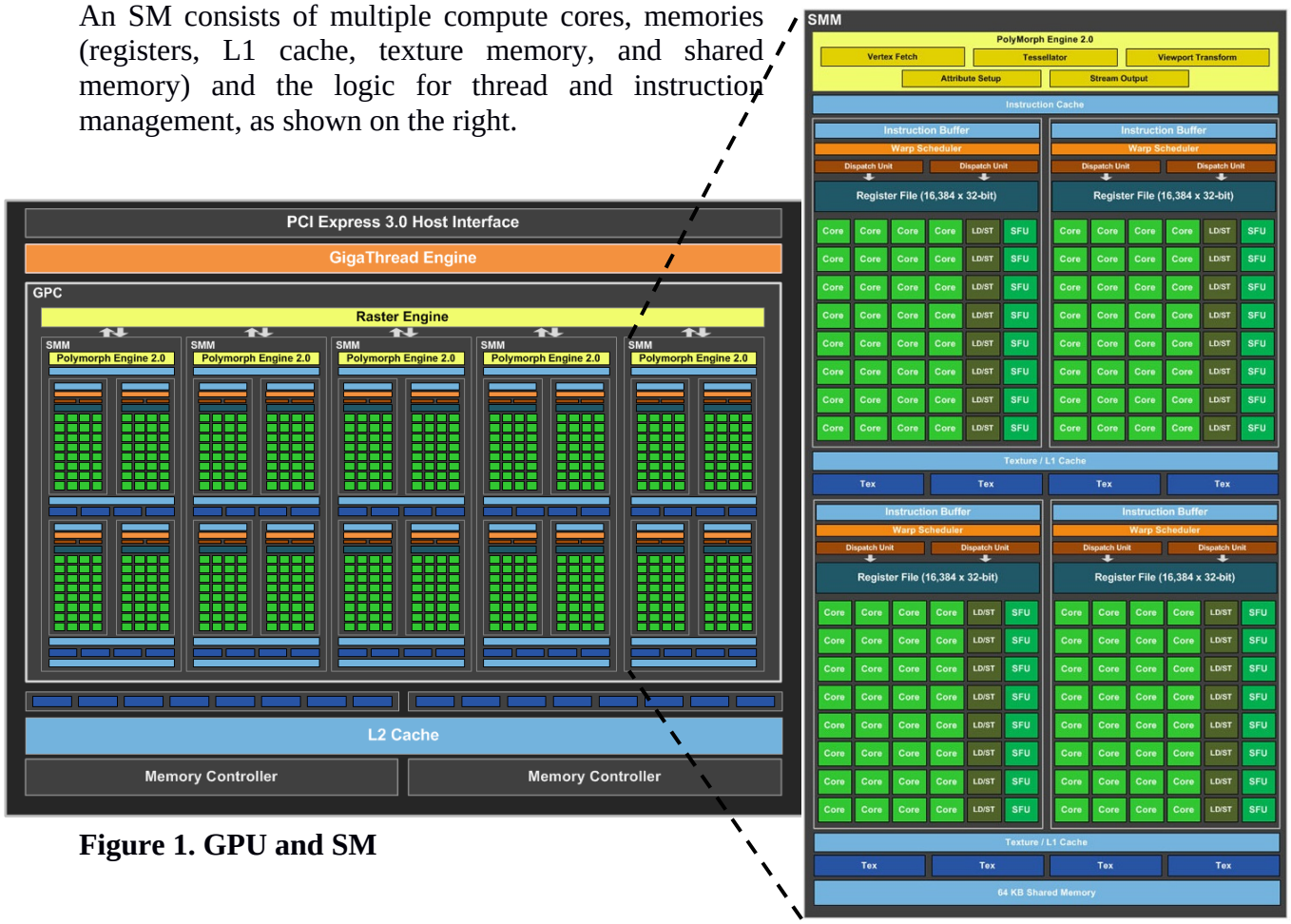
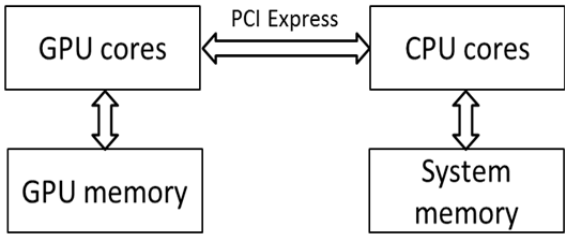


Figure 1. GPU and SM

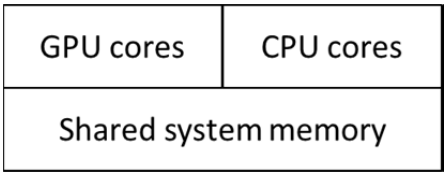
Table below summarizes the types of GPU memories and their characteristics.

Type	Scope	Access type	Speed	CUDA syntax	Explicit sync
Register	thread	RW	fastest	-	no
Local	thread	RW	very fast	`float x`	no
Shared	block	RW	fast	`__shared__ float x`	yes
Global	program	RW	slow	`__device__ float x;`	yes
Constant	program	R	slow	`__constant__ float x;`	yes
Texture	program	R	slow	`__texture__ float x;`	yes

Figures below shows the two main organizations between the CPU and GPU:

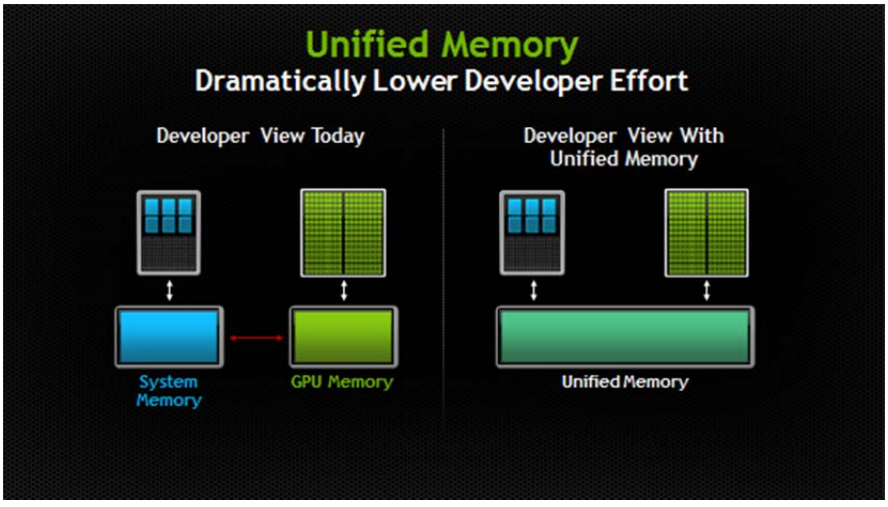


System with discrete GPU
e.g. Node2 (Dell Inspiron)



Systems with integrated GPUs
e.g Jetson TK1

Jetson TK1 integrates 4 ARM Cortex-A15 CPU cores, 192 Nvidia Kepler GPU cores and 2 GB of RAM on the same chip. On such a system, there is no need to explicitly transfer the data between CPU and GPU, since the memory is unified.



CUDA programming

CUDA is installed by default in `/usr/local/cuda` (on Jetson) or `/usr/local/cuda-6.5` (on Node2). This folder contains, along with libraries and binaries, example programs in `samples` folder. Please explore the content of this folder.

A CUDA program, with the extension **`.cu`**, is a modified C program with sections of CUDA code that are executed on the GPU, called *kernels*. In CUDA's terminology, the CPU is called the *host* and the GPU is called the *device*. For CUDA programs, there is a separation between the code executed on the host and the code executed on the device. The following keywords are used as function prefixes to differentiate these codes:

- `__global__` - the function is called by the host and executed on the device
- `__device__` - the function is called from within device code and executed on the device
- `__host__` - the function is executed on host

Programmer can define functions that are both `__device__` and `__host__`.

The GPU kernels are executed by CUDA threads which are organized in blocks and grids. In each block, threads can be organized in one dimension (1D), two dimensions (2D) or three dimensions (3D). Similarly, blocks are organized as grids in 1D, 2D or 3D. A kernel is called using its name and triple angular brackets specifying the grid and block dimensions.

For example, to call a kernel using threads organized in a cube of size 4 and blocks organized in a square matrix of size 8, the following declarations can be used:

```
dim3 threadsPerBlock(4, 4, 4);
dim3 blocksPerGrid(8, 8);
kernel<<< blocksPerGrid, threadsPerBlock>>>();
```

The kernel is executed on each Streaming Multiprocessor (SM) in groups of 32 threads, called *warps* in CUDA terminology.