Parallel Computer Architecture
Homework #1

This homework assignment may be performed *in teams of two students*. If you work with a teammate, please indicate in your report how you distributed the work within the team. You are allowed to post questions on the course mailing list; however, you should not use any resources other than the class material and NVIDIA documentation (http://docs.nvidia.com).

Submission instructions: Email your code to becchim@missouri.edu by the due date, using the following email subject: "ECE8270: HW1 code". In addition, print your report and bring it to class. Note that you will be evaluated both for the correctness of your code and for the clarity and completeness of your report. *Your report should not be longer than 5 pages* (including figures and charts).

Use either the workstations in Lafferre Hall C1246 lab or the `cuda1/cuda2.ece.missouri.edu` servers to complete the assignment.

Suggestions:

- You can use the `gettimeofday()` function to insert timestamps in your code and measure the execution time of different code portions: http://linux.die.net/man/2/gettimeofday. For example, you can use the following function in your code:

```
#include <sys/time.h>

double gettime() {
    struct timeval t;
    gettimeofday(&t,NULL);
    return t.tv_sec+t.tv_usec*1e-6;
}
```

- Recall that kernel calls are *asynchronous* CUDA functions: to correctly time the execution of a kernel function, it is necessary to insert a barrier synchronization after it.

Problem 1 – Vector operations on GPU

You have seen a simple vector addition kernel in class. That kernel has a limitation: it operates correctly (that is, it computes the whole vector addition) only if the execution configuration comprises at least as many threads as number of elements in the vector to be computed.

Write a new vector addition program for GPU as follows:

- The program must include a generic vector addition kernel whose correctness is independent of the execution configuration (that is, the result is not affected by the number of thread-blocks and of threads-per-block instantiated in the kernel launch).

- The execution configuration (number of blocks and of threads-per-block) must be a command line argument (and should not be hardcoded).

- Your code will query the memory capacity of the device, and then size the vectors to be added in such a way that a given fraction *mem_utilization* of the global memory will be occupied. *mem_utilization* should be a command line argument of your program.

  Suggestion: You can have a look at the CUDA Runtime API Device Management Functions:
  http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html#group__CUDART__DEVICE

- The input and output vectors should contain randomly generated, single precision floating point numbers.

  Suggestion: Use the `rand()` C library function:
  http://www.tutorialspoint.com/c_standard_library/c_function_rand.htm

- Your program should have a *–multigpu* option as follows. If this option is enabled, the program will query how many GPU devices are available on the system, what is their memory capacity, and it will split the vector addition across multiple devices.

  Suggestion: You can again have a look at the CUDA Runtime API Device Management Functions.

- Your program should include a *–validate* command line option used to validate the correctness of the results against a CPU implementation.

Guidelines for the report: Perform experiments with:
- *mem_utilization* = 10%, 30%, 50% and 70% of the global memory capacity.
- different, significant execution configurations (to select the execution configurations you intend to test, query the hardware configuration of your device).
- Compare the CPU and GPU execution time in all cases.

Select the results that you consider more relevant, and organize them in tables or charts. In particular, you may want to discuss how the size of the dataset and the execution configuration affect the performance (and, in particular, the performance difference between CPU and GPU). Please remember to indicate the GPU that you are using and its hardware configuration.

Problem 2 – Matrix Transpose on GPU

Write a program that transposes a matrix of size *N* on GPU. Your program must meet the following requirements:

- The program must offer two kernels: (1) a *transpose_global* kernel that uses only global memory, and (2) a *transpose_shared* kernel that uses also shared memory

- The kernels can use only *mono-dimensional thread-block and thread identifiers*.

- The kernel configuration should again be a command line argument, and the kernels should produce correct results independent of the number of thread-blocks and of threads-per-block instantiated in the kernel launch.

- The size of the matrix should depend on the *mem_utilization* command line argument defined above; in addition, the matrix should contain randomly generated single precision floating point numbers.

- Your program should have a *–validate* command line option used to validate the correctness of the results against a CPU implementation.

Guidelines for the report:

- Perform experiments with:
  - *mem_utilization* = 10%, 30%, 50% and 70% of the global memory capacity;
  - different, significant execution configurations.
- Compare the performance of a serial CPU implementation and of your GPU implementations (with and without shared memory).
- Use relevant tables/charts to present the results in a clear and compact way.
- Discuss your results. *Suggestion*: to explain the performance differences between the two GPU implementations, use the Nvidia profiler. What is the arithmetic intensity of the matrix transpose operation? (*Note*: The arithmetic intensity is defined as the number of operations performed per word of memory transferred). How does the arithmetic intensity affect the results?

Please remember to indicate the GPU that you are using and its hardware configuration. In fact, the performance obtained will be strictly related to the underlying hardware.