Dept of Computer Science & Engineering
Faculty of Engineering, University of Moratuwa

BSc Engineering, Semester 8 (2012 Intake)

CS4552 Scientific Computing
Assignment 1 (Due: 23rd December 2016)

Note: *Read this document completely before you start to ensure you fully understand the requirements. All questions are equally worth.*

General Requirements:

1. For all programs, use -O3 optimization level (or equivalent) for the compilation. All results must be from executing the programs on one machine with a GPU and at least a 4-core CPU. It is recommended to use the provided machine in the CSE Dept, but an equivalent or better alternative is acceptable provided the following machine details are provided: CPU name and model, # cores in CPU, CPU clock rate, GPU name and model, amount of RAM, size of CPU caches, OS version, compiler versions.
2. In all questions, you are asked to implement parallel CPU versions. You may use Pthreads or OpenMP for that; they should have reasonably good performance and scalability (to have fair comparisons).
3. **For Q2, Q3**: You have to implement within the same program (a) a sequential (CPU) version, (b) a parallel CPU version, and (c) a parallel CUDA-GPU version. Let the user select the version when the program is invoked, similar to the following examples:

```
$ ./prog -s          (run the sequential version)
$ ./prog -p 8        (run the parallel version using 8 CPU threads)
$ ./prog -c          (run the CUDA-GPU version)
```

Q1. You are given 4 CUDA source files, each of which computes (estimates) the value of PI . The files are pi-mystery.cu, pi-curand.cu, pi-myrand.cu and pi-curand-thrust.cu. Do the following.

(a) Explain how PI is computed in pi-mystery.cu (how/why does it work?). Note that it does not generate random-numbers and therefore not a "monte-carlo" approach.

(b) Modify pi-mystery.cu such that the estimated value of PI, the error and the program performance can be fairly compared with those of pi-curand.cu. You need to determine appropriate values for the number of bins, blocks and threads (among others changes, as needed) in pi-mystery.cu such that its amount of computation matches with the total number of trials in pi-curand.cu and the output format should match with that of pi-curand.cu.

(c) Modify pi-curand.cu by adding a CPU parallel computation of PI, where the number of threads is given as a command-line input and the total amount of computation (e.g., the total number of trials) is the same but now divided among the threads such that elapsed time is minimized. Output results similar to the other computations, but also with the number of threads included. Keep the existing "`host_monte_carlo( )`" since it, the CPU sequential computation, is the baseline.

(d) Modify pi-myrand.cu such that the estimated value of PI, the error and the performance can be compared with those of pi-curand.cu. Determine appropriate values for the number of blocks, number of threads and the number of trials per thread (among others changes, as needed) in pi-myrand.cu such that its computation matches with the total number of trials in pi-curand.cu and the output format should match with that of pi-curand.cu.

(e) Modify pi-curand-thrust.cu such that the estimated value of PI, the error and the performance can be compared with those of pi-curand.cu. Determine appropriate values for M and N (among others changes, as needed) in pi-curand-thrust.cu such that its computation matches with the total number of trials in pi-curand.cu and the output format should match with that of pi-curand.cu.

(f) The given source files use single-precision floating point ("float" type) for the computation. Modify them such that the user can choose between single-precision floating point ("float" type) and double-precision floating point ("double" type), at either compile-time or run-time. Ensure that output formats among programs match as closely as possible (including execution times, which should be reported in the same units and format).

(g) Submit the 4 modified source files pi-mystery.cu, pi-curand.cu, pi-myrand.cu and pi-curand-thrust.cu. Make sure to comment your code adequately.

(h) Run the 4 modified programs with the total number of trials set to 2^{24}, 2^{26} and 2^{28} with both single-precision (SP) and double-precision (DP) floating-point formats in each case. Report the results in tabular format as follows. "Time" is the elapsed time for the specific computation, which can be in seconds (s) or milliseconds (ms). Explain your results briefly but addressing all key observations.

# Trials	Single- or Double-Precision (SP/DP)		CPU Sequential [curand]	CPU Parallel [curand] (T=# threads)			GPU			
				T=2	T=4	T=8	mystery	myrand	curand	curand-thrust
2^{24}	SP	PI estimate								
		Error								
		Time								
	DP	PI estimate								
		Error								
		Time								
2^{26}	SP	PI estimate								
		Error								
		Time								
	DP	PI estimate								
		Error								
		Time								
2^{28}	SP	PI estimate								
		Error								

		Time								
	DP	PI estimate								
		Error								
		Time								

Q2. Implement the dot-product computation between two vectors (1D arrays) whose elements should be initialized at the start of the program with random real numbers between 1 and 2 (with the ability to choose either single-precision or double-precision, as in Q1). Given two vectors V_1 and V_2 , each with N elements, where $V_1[i]$ and $V_2[i]$ denote the *i*-th element in each, respectively, the dot product between them is the sum of $(V_1[i] \cdot V_2[i])$ over all i=1,2,...,N.

Run your programs and show timing results for the size of each vector, $N=10^8$, $5x10^8$ and 10^9. (If $N=10^9$ is too large for the system, then use 1/10 of each N, i.e., 10^7, $5x10^7$ and 10^8).

A command-line option should be provided to verify the accuracy of the parallel computation (e.g., `./prog -p 8 -v` will verify the 8-thread CPU version and `./prog -c -v` will verify the CUDA-GPU version). Verification should be against the sequential computation, by showing any difference between results is insignificant; note that the results would not be exactly equal between two computations because the vector elements are floating-point real numbers.

If necessary, use the `-mcmodel=large` option with gcc when statically allocating large arrays > 2 GB.

Report the elapsed times for the dot-product computation in milliseconds (ms) as follows. Make sure to exclude time spent on vector initialization and other such unrelated work (you may use the approach suggested at the end of this document). Explain your results briefly but addressing all key observations.

N	**Single- Or Double-Precision?**	**CPU Sequential**	**CPU Parallel (T=# threads)**			**GPU**
			T=2	**T=4**	**T=8**	
	SP					
	DP					
	SP					
	DP					
	SP					
	DP					

Q3. Implement Matrix multiplication, C=A·B, between two NxN square matrices A and B whose elements should be initialized at the start of the program with random real numbers between 1 and 2 (with the ability to choose either single-precision or double-precision, as in Q1, Q2). Matrices may have either linear 1D layout or 2D layout. Show timing results for N=600, 1200 and 1800. (If 1800 is too large for the system, then use 500, 1000 and 1500). A command-line option should be provided to verify the correctness of the parallel computation, as in Q2 above.

For the sequential CPU version, a simple straight-forward implementation is acceptable.

For the parallel CPU version, a straight-forward implementation where each thread computes its part (block, set of rows or set of columns) of the C matrix would be acceptable.

For GPUs, you must implement a basic, straight-forward implementation where each thread computes an element of the C matrix. For extra credit, you are urged to implement a performance-enhanced version (consider using a "tiling" approach for blocking of computation and using "shared memory" or any other technique, but making sure to state what you do).

Report the elapsed times for the matrix multiplication in milliseconds (ms) as follows. Make sure to exclude time spent on initialization and other such unrelated work (you may use the approach suggested at the end of this document). Explain your results briefly but addressing all key observations.

NxN	Single- Or Double-Precision?	CPU Sequential	CPU Parallel (T=# threads)			GPU	
			T=2	T=4	T=8	Basic	Enhanced
	SP						
	DP						
	SP						
	DP						
	SP						
	DP						

What to Submit:

Organize your source code in three separate directories named as q1, q2, q3 for the questions Q1, Q2, Q3, respectively. Keep additional files etc to a minimum in each directory, to make things simple and clear. You can include some form of brief documentation (e.g., instructions on compiling, running etc) inside your source files, or if really necessary in a README file inside each directory. Make sure there are no object files or executables in these directories. Prepare the answers for the questions in a separate PDF document in which your name and registration number should be included. If you work with other students on this assignment or get help from other resources, you must list all of them in your submitted PDF document.

Include the PDF document and the 3 directories q1, q2, q3 inside a top-level directory identified by your name (e.g., "yasiru"). Submit a zip file of that directory (e.g., "yasiru.zip") onto the Moodle.

Additional Information:

In Q2, Q3: To measure elapsed time from within a program and for reporting timing info in milliseconds, you may use something like the following:

```
#include <time.h>                // for clock_gettime( )
#include <errno.h>               // for perror( )
```

```
#define      GET_TIME(x);        if (clock_gettime(CLOCK_MONOTONIC, &(x)) < 0) \
                                 { perror("clock_gettime( ):"); exit(EXIT_FAILURE); }
#ifndef      N
#define      N      10000000     // problem size
#endif       // can specify N to be x at compile-time as: gcc -DN=x ...

int main(int argc, char **argv)
{
       // ...
       struct timespec     t0, t1, t2;
       unsigned long       sec, nsec;
       float               comp_time;   // in milli seconds

       GET_TIME(t0);
       // do initializations, setting-up etc
       GET_TIME(t1);
       // do computation
       GET_TIME(t2);
       comp_time = elapsed_time_msec(&t1, &t2, &sec, &nsec);
       printf("N=%d: Threads=%d: Time(ms)=%. 2f \n", N, numthreads, comp_time);
       // finishing stuff
}

float elapsed_time_msec(struct timespec *begin, struct timespec *end, long *sec, long *nsec)
{
       if (end->tv_nsec < begin->tv_nsec) {
              *nsec = 1000000000 - (begin->tv_nsec - end->tv_nsec);
              *sec  = end->tv_sec - begin->tv_sec -1;
       }
       else {
              *nsec = end->tv_nsec - begin->tv_nsec;
              *sec  = end->tv_sec - begin->tv_sec;
       }
       return (float) (*sec) * 1000 + ((float) (*nsec)) / 1000000;
}
```