Lab1C GPU/CUDA Implementation of Dense Matrix Multiplication

In this lab, you will implement the square matrix multiplication algorithms (on GPU using CUDA and CUBLAS library, and study their performance. The implementation should include three different versions of the kernels:

1) input matrix A and B are all stored in global memory and kernel computation access data directly from global memory,

2) input matrix A and B are read into shared memory of thread blocks and computation access data from shared memory,

3) implementation directly calls **sgemm** procedure of CUBLAS library to perform the computation.

The implementation need to include two CUDA kernels codes for version 1 and 2 and codes for memory allocation and data movement. For version 3, the codes for the implementation are mainly for memory allocation/data movement and call to the sgemm procedure. For version 2 and 3, you can leverage code in /opt/nvidia/cuda-7.5/samples/0_Simple/(matrixMul and matrixMulCUBLAS) on fornax, which literately are the solutions for the two kernels. You may also refer to the CUDA programming guide for the implementation (http://docs.nvidia.com/cuda/cuda-c-programming-guide/#shared-memory). The algorithms for kernels will need a 2-dimension topology of both threads of a block and blocks of the grid, and please choose 16x16 for the block size. Each thread will compute one element of the matrix C. To simplify, we will assume the matrix size N to be a number of power of 2 (64, 128, 256, 512, …).

The **matmul.cu** file provided includes helper functions:

- matmul_base function for the sequential implementation,

- matmul_openmp function that is the openmp-parallelized version for CPU. You should put the three versions along with the two CUDA kernels in the matmul.cu file. The main functions need to be modified to include code to drive and time the three implementation, and reports timing and error information. Arrays A, B and C are all now allocated on the heap on the host using malloc so we can run the experiments with bigger input.

The **matmul.cu** should be compiled using nvcc compiler with "**-Xcompiler –fopenmp**" to enable the compilation of the OpenMP version, e.g.

```
nvcc –Xcompiler -fopenmp matmul.cu -lpthread -lcublas -o matmul
```

Your executable should be able to run with two arguments: the first required argument is for the matrix size: N for NxN square matrix; the second optional argument is the # of OpenMP threads for parallelization on CPU, with default value 5 if not provided.

The output of your program should include both the error of computation, time(ms) and FLOP/s performance. Below is a screenshot of the output for running assignment 2'S code (is not parallelized yet) so you get idea of what normally we can put in the output.

```
yy8@yy8-vm:~/machome/Desktop/OaklandUniversity/teach/CSE536/assignment/assignment2$ ./matmul 1024 4
================================================================================================
Matrix Multiplication: A[M][K] * B[k][N] = C[M][N], M=K=N=1024, 4 threads/tasks
------------------------------------------------------------------------------------------------
Performance:            Runtime (ms)     MFLOPS                 Error (compared to base)
------------------------------------------------------------------------------------------------
matmul_base:            7618.000031      281.895988             0
matmul_pthread:         1437.999964      1493.382269            0
matmul_openmp:          7543.999910      284.661144             0
```

To study the performance, you will need to use the nvprof tool and please refer to the documentation page (http://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview) for how to use this profiler.

To profile a specific CUDA implementation, you may need to comment off other kernels calls in the main program.

The performance results collected for the report should be done on fornax.cse.sc.edu. There are four GPUs that you can use with device id 0, 1,2,3. By default, you all will use GPU 0 and you can use cudaSetDevice(1) call to select a different GPU to use. Please random choose a GPU to use so we do not all work on the same device.

Submission: The submission should include two files: the matmul.cu file that contains your implementations and a max 3-page report. The report should include:

1. Description on your implementation of the three versions.
2. One performance figure that reports the results for running the code with N=512, 1024, and 2048 matrix on fornax. The figure should show the execution time for the openmp version (matmul_openmp) that uses all the CPU cores of the machine (use lscpu command to check the total number of CPUs), and the execution time (both the kernel time and memory allocation/data movement time) of the three GPU versions.
3. One performance figure that shows the breakdown of the execution time for N=2048 of the three versions of the GPU kernels. The breakdown figure will show at least three timing information, the execution time for data movement from host to the GPU, kernel execution time, and time for data copy back. Ideally, the percentage of each of the

breakdown over the total execution time will give more information, but I hope the absolute value together will show that. You should collect that information using nvprof

4. Explanation of the performance results shown in the two figures. You report should include a detailed specification of the machine/GPU and software environment you are using, e.g. CPU vendor/model, the number of CPU cores, CPU memory size, GPU model/vendor, memory size, # of SM/cores, CUDA SDK (nvcc) version, gcc version (since nvcc use it) and the compiler flags used to build the executable. "cat /proc/cpuinfo" and "cat /proc/meminfo" commands will give you CPU/mem info and deviceQuery executable from /usr/local/cuda/samples/1_Utilities will print out the GPU information. The purpose of this information is for people who may want to do the same experiment as you.

The assignment3-plot.xlsx file will help you to generate the figures from the results you will collect. While the development can be done from your laptop or any other computers, the results in the report should be collected from the cluster. Please be noted that the machine is shared resource, overloaded use of the machine may cause incorrect performance results.

For the CUBLAS sgemm implementation, the following paper discussed in details the optimization applied to the library: "Vasily Volkov and James W. Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC '08). IEEE Press, Piscataway, NJ, USA, , Article 31 , 11 pages.

Grading:

Functions implementations

1. **40 points for implementation.**
2. **Report: 60 points.**

For non-compliable code, you only receive max 60% of function implementations points. For compliable, but with execution errors and incorrectness, you receive max 70% of function implementation points. Please refer to the next page for the policy of academic conduct.

More info for the assignment:

The GPU needs to be warmed up to collect realistic performance data. The way of doing it is simple: for each version, call the function first without timing it, and then call it with timer turned on. See below:

```
matmul_cuda_v1_vanilla (...); /* warm up the GPU */

elapsed_v1 = read_timer();

matmul_cuda_v1_vanilla( ... )
```

```
elapsed_v1 = (read_timer() - elapsed_v1);


matmul_cuda_v2_shmem (...); /* warm up the GPU */

elapsed_v2 = read_timer();

matmul_cuda_v2_shmem( ... )

elapsed_v2 = (read_timer() - elapsed_v2);


matmul_cuda_v3_cublas (...); /* warm up the GPU */

elapsed_v3 = read_timer();

matmul_cuda_v3_shmem( ... )

elapsed_v3 = (read_timer() - elapsed_v3);
```

To produce second figures using nvprof, for each version, you need to comment the other two versions in your code (both the warm call and the call to be timed), but leave the warm up call for this version, rebuild it and run it with "nvprof ./matmul 2048". It should produce something similar to the following. You will collect the "time" column of the first three rows (kernel time, HtoD time and DtoH time, and put them in the sheet provided to produce the breakdown timing figures.

```
==28987== Profiling result:

Time(%)      Time     Calls       Avg       Min       Max  Name

 95.30%  8.1613ms        31  263.27us  259.57us  267.76us  void magma_lds128_sgemm_kernel<bool=0, bool=0, int=6, int=5, int=3, int=3, int=3>(int, int, int, float const *, int, float const *, int, float*, int, int, int, float const *, float const *, float, float, int)

  3.21%  275.17us         3  91.722us  1.5360us  137.15us  [CUDA memcpy HtoD]

  1.49%  127.58us         1  127.58us  127.58us  127.58us  [CUDA memcpy DtoH]


==28987== API calls:

Time(%)      Time     Calls       Avg       Min       Max  Name

 46.87%  169.19ms         7  24.170ms  15.572us  168.45ms  cudaFree

 25.98%  93.788ms         1  93.788ms  93.788ms  93.788ms  cudaDeviceReset
```