Report Lab 1

Andres Calderon

October 30, 2015

1 Code

The following code was used to complete the report:

1.1 kernel.cu

```
/******************************************************************************
 *cr
 *cr            (C) Copyright 2010 The Board of Trustees of the
 *cr                        University of Illinois
 *cr                         All Rights Reserved
 *cr
 ******************************************************************************/

#include <stdio.h>

#define TILE_SIZE 16

__global__ void mysgemm(int m, int n, int k, const float *A, const float *B, float *C) {

    /********************************************************************
     *
     * Compute C = A x B
     *   where A is a (m x k) matrix
     *   where B is a (k x n) matrix
     *   where C is a (m x n) matrix
     *
     * Use shared memory for tiling
     *
     ********************************************************************/

    // INSERT KERNEL CODE HERE
    // Declaring the variables in shared memory...
    __shared__ float A_s[TILE_SIZE][TILE_SIZE];
  __shared__ float B_s[TILE_SIZE][TILE_SIZE];

  // Finding the coordinates for the current thread...
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int col = blockIdx.x * blockDim.x + tx;
  int row = blockIdx.y * blockDim.y + ty;

  float sum = 0.0f;

  for(int i = 0; i < ((k - 1) / TILE_SIZE) + 1; ++i){
    // Validation in the case the thread tries to write in share
    // memory of the dimensions of matrix A...
    if(row < m && (i * TILE_SIZE + tx) < k){
      A_s[ty][tx] = A[(row * k) + (i * TILE_SIZE + tx)];
    } else {
      // In that case, just write a 0 which will no affect
      // the computation...
      A_s[ty][tx] = 0.0f;
```

```
48      }
49      // Similar validation for B...
50      if((i * TILE_SIZE + ty) < k && col < n){
51        B_s[ty][tx] = B[((i * TILE_SIZE + ty) * n) + col];
52      } else {
53        B_s[ty][tx] = 0.0f;
54      }
55      // Wait for all the threads to write in share memory
56      __syncthreads();
57
58      // Compute the multiplication on the tile...
59      for(int j = 0; j < TILE_SIZE; ++j){
60        sum += A_s[ty][j] * B_s[j][tx];
61      }
62      // Wait to finish before to go ahead with the next phase...
63      __syncthreads();
64    }
65    // Write the final result in C just if it is inside of the valid
66    // dimensions...
67    if(row < m && col < n){
68      C[row * n + col] = sum;
69    }
70
71  }
72
73  void basicSgemm(char transa, char transb, int m, int n, int k, float alpha, const float *A, int lda, const
      ↪  float *B, int ldb, float beta, float *C, int ldc)
74  {
75      if ((transa != 'N') && (transa != 'n')) {
76      printf("unsupported value of 'transa'\n");
77        return;
78      }
79
80      if ((transb != 'N') && (transb != 'n')) {
81      printf("unsupported value of 'transb'\n");
82      return;
83      }
84
85      if ((alpha - 1.0f > 1e-10) || (alpha - 1.0f < -1e-10)) {
86      printf("unsupported value of alpha\n");
87      return;
88      }
89
90      if ((beta - 0.0f > 1e-10) || (beta - 0.0f < -1e-10)) {
91      printf("unsupported value of beta\n");
92      return;
93      }
94      const unsigned int BLOCK_SIZE = TILE_SIZE;
95
96      // Initialize thread block and kernel grid dimensions
97      const dim3 dim_block(BLOCK_SIZE, BLOCK_SIZE, 1);
98    const dim3 dim_grid(((n - 1) / BLOCK_SIZE) + 1, ((m - 1) / BLOCK_SIZE) + 1, 1);
99
100     // Calling the kernel with the above-mentioned setting...
101   mysgemm<<<dim_grid, dim_block>>>(m, n, k, A, B, C);
102 }
```

2 Answer to Questions

1. In your kernel implementation, how many threads can be simultaneously executing? Assume a GeForce GTX 280 GPU which has 30 streaming multiprocessors.

2. Use nvcc –ptxas-options="-v" to report the resource usage of your implementation your implementation. Note that the compilation will fail but you will still get a report of the relevant information. Experiment with the Nvidia visual profiler, which is part of the CUDA toolkit, and use it to further understand the resource usage. In particular, report your branch divergence behavior and whether your memory accesses are coalesced.

```
NVCC        = nvcc
NVCC_FLAGS  = --ptxas-options="-v" -O3 -I/usr/local/cuda/include
LD_FLAGS    = -lcudart -L/usr/local/cuda/lib64
EXE          = sgemm-tiled
OBJ          = main.o support.o

default: $(EXE)

main.o: main.cu kernel.cu support.h
  $(NVCC) -c -o $@ main.cu $(NVCC_FLAGS)

support.o: support.cu support.h
  $(NVCC) -c -o $@ support.cu $(NVCC_FLAGS)

$(EXE): $(OBJ)
  $(NVCC) $(OBJ) -o $(EXE) $(LD_FLAGS)

clean:
  rm -rf *.o $(EXE)
```

Figure 1: Content of Makefile.

```
storm.ee.ucr.edu /home/tempmaj/classacc2391/PhD/Y1Q1/GPU/lab2 $ make
nvcc -c -o main.o main.cu --ptxas-options="-v" -O3 -I/usr/local/cuda/include
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z7mysgemmiiiPKfS0_Pf' for 'sm_10'
ptxas info    : Used 12 registers, 2104 bytes smem, 12 bytes cmem[1]
nvcc main.o support.o -o sgemm-tiled -lcudart -L/usr/local/cuda/lib64
storm.ee.ucr.edu /home/tempmaj/classacc2391/PhD/Y1Q1/GPU/lab2 $
```

Figure 2: Output of compilation using *–ptxas-options="-v"*.

The *–ptxas-options="-v"* option was included in the Makefile as shows figure 1. The output result can be seen in figure 2. It shows the number of registers allocated (12), the size of share memory (around 2 Kb) and some bytes used for constant memory (12 bytes). It is consistent with the use of two floating-point arrays in share memory for matrices A and B, each of 16x16 (TILE_SIZE). The remaining bytes and size of constant memory could be explained by the use of kernel arguments and internal operations.

Using the Nvidia Visual Profiler (NVVP) it was possible to extract valuable information about the performance of the code. A very gentle introduction to the use of NVVP is available at [3]. NVVP 5.0 (available at storm.ee.ucr.edu) was used to analyze the implementation. For TILE_SIZE=16 the results are shown in figure 3. All the tests were run using the default parameters.

We can see that the figures for *Branch Divergence Overhead*, which measures the instruction issue overhead caused by divergent branches, and *Total Replay Overhead*, the percentage of instruction issues due to memory replays, are relatively small in comparison with the same figure for the no tiled version (figure 4). Similarly, the value of *Global Memory Replay Overhead*, the percentage of instruction issues due to replays for non-coalesced global memory accesses, is consistently smaller in the tiled version. However, as we can see for the *Global Load Efficiency* metric, the use of global memory bandwidth is more efficient for the simple implementation (61.2%) than for the tiled implementation (39.8%).

3. Compare the performance of the The Tiled Matrix multiplication to the simple matrix multiplication as you increase the size of the matrices and for different tile sizes. Explain any trends that you see.

   In order to collect data, the script in figure X was used to run three iterations of the implementation with different values of N (size of a square matrix). The result of the iterations was saved to a log file using the following command line functions:

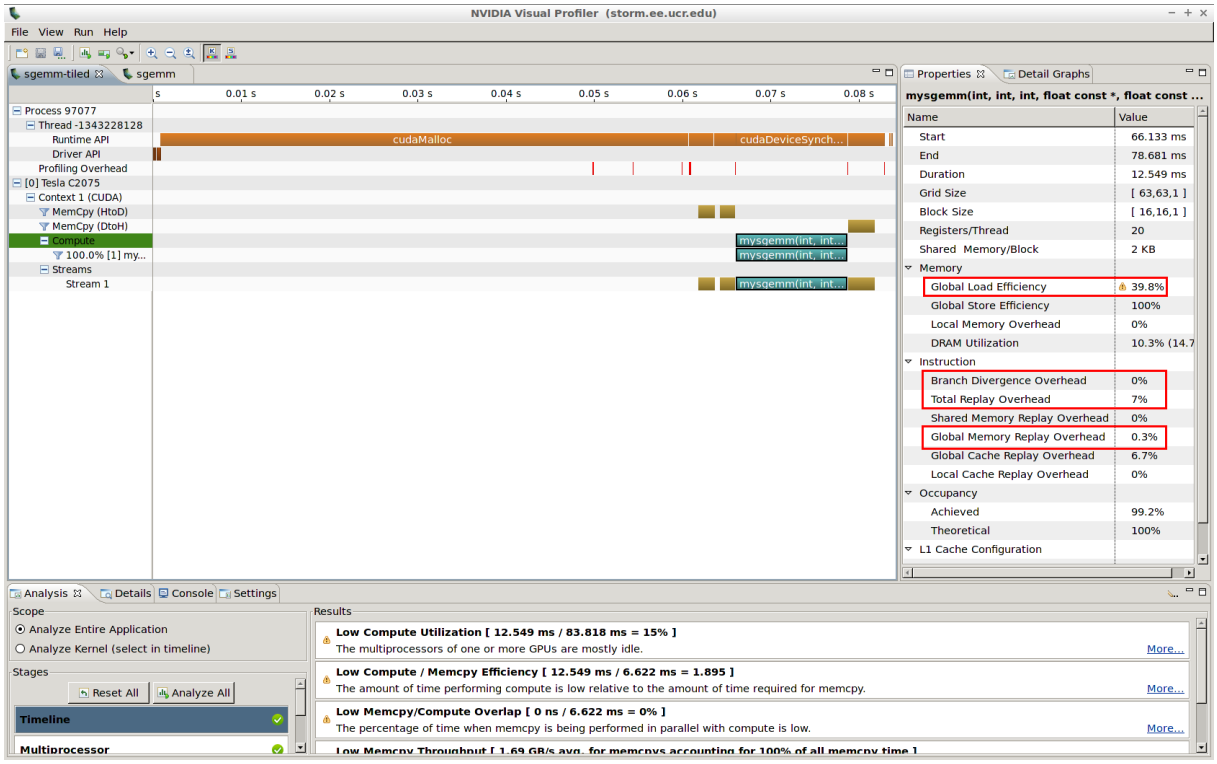

Figure 3: NVVP performance analysis for *sgemm-tiled*.

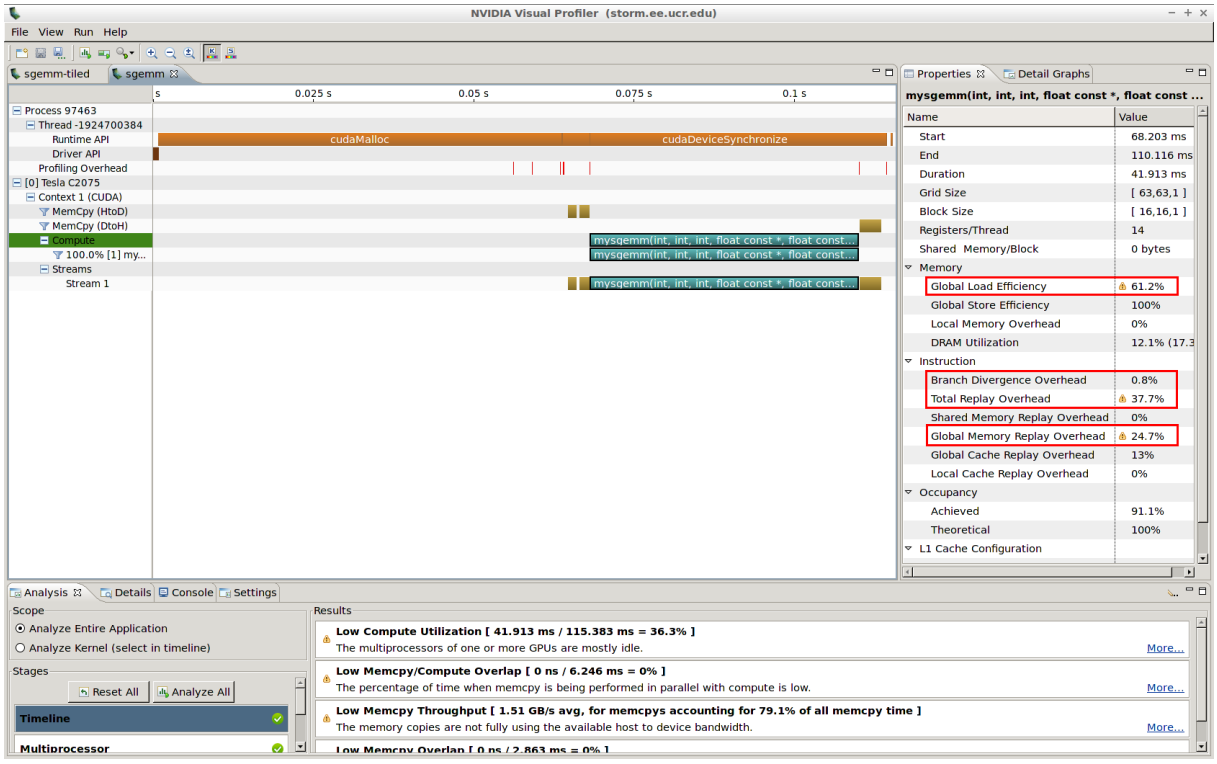

Figure 4: NVVP performance analysis for *sgemm*.

```
#!/bin/bash
date
for i in `seq 50 50 1000`; do
  ./sgemm-tiled $i
  ./sgemm-tiled $i
  ./sgemm-tiled $i
done
STRING="Done!!!"
echo $STRING
date
```

Figure 5: Script to collect test data.

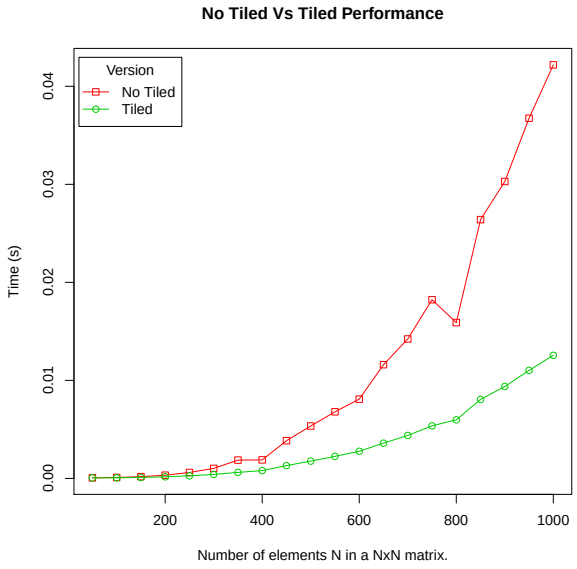

Figure 6: First performance comparisson between tiling and no tiling versions.

```
storm.ee.ucr.edu /home/tempmaj/classacc2391/PhD/Y1Q1/GPU/lab2 $ ./test.sh >
 ↪  testing_tiled_1K-20K_T16.log
```

Then, the values for the time executing of the kernel section is extracted using:

```
storm.ee.ucr.edu /home/tempmaj/classacc2391/PhD/Y1Q1/GPU/lab2 $ more
 ↪  testing_tiled_1K-20K_T16.log | grep 'Launching kernel...' | grep -Po '\d+.\d+' >
 ↪  times_tiled_1K-20K_T16.dat
```

The files were processed in R[1] to get the average of the three iterations and generate some plots. All data, code and figures are available at [1].

Figures 6 and 7 show the performance between both implementations. The tests were set using different values of N for square matrices. Overall, the tiled implementation outperforms the simple one in small and relatively bigger datasets. Although, the tiled implementation shows a lower global load efficiency, it seems that a more appropriate control of branch divergence and coalesced access gives better results.

[1] https://www.r-project.org/

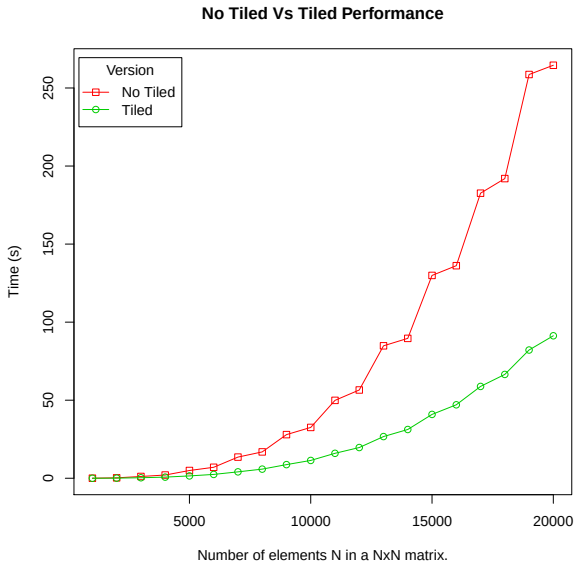

Figure 7: Second performance comparisson between tiling and no tiling versions.

Figure 8 shows the performance for different values of TILE_SIZE. The general trend shows that the larger the size of the tile the better performance. However, there is not significant difference between sizes 16 and 32. Figure 9 shows an additional test with bigger values of N for just these two values. It seems there is not an increase in performance using a size of 32 over 16. That is explained by the fact that a TILE_SIZE equal to 32 allows 1024 threads per block, but the hardware limitation for each Streaming Multiprocessor (SM) is up to 1536 threads [2]. So, under this configuration just one block (1024 threads) is actively executing.

References

[1] Andres Calderon. *GitHub Personal Repository.* `https://github.com/aocalderon/PhD/tree/master/Y1Q1/GPU/lab2`, 2015.

[2] David Kirk and Wen-Mei Hwu. *Programming Massively Parallel Processors: A Hands-On Approach.* Morgan Kaufmann, 2012.

[3] David Luebke, John Owens, Mike Roberts and Cheng-Han Lee. *Using NVVP Part1 and Part 2 - Intro to Parallel Programming.* Udacity Course, 2015. `https://www.youtube.com/watch?v=hyKA5fb5ZJI`.

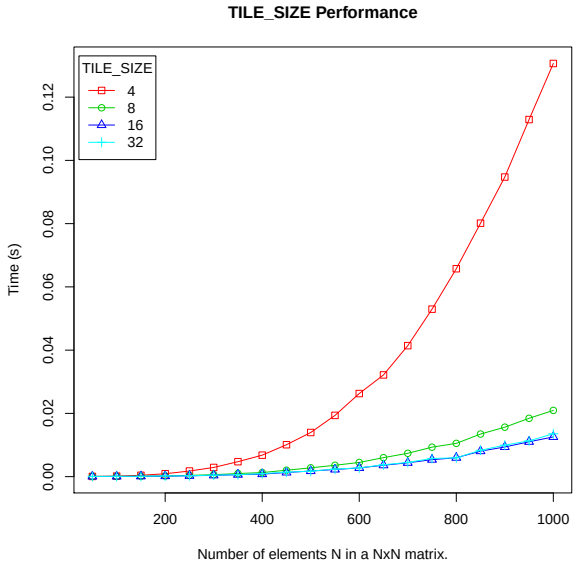

Figure 8: Performance using different values of TILE_SIZE.

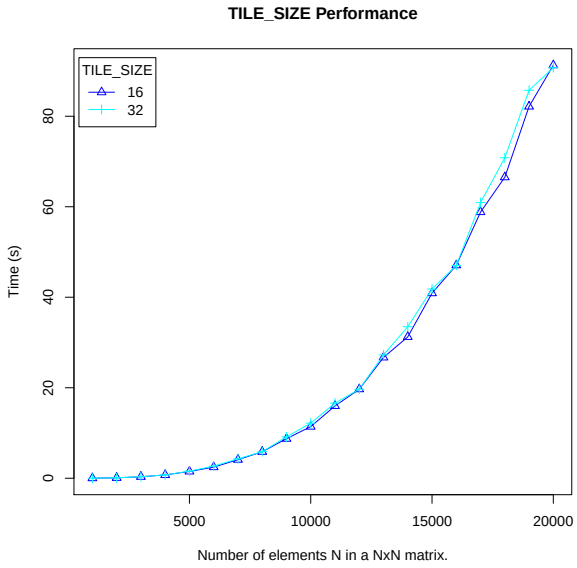

Figure 9: Performance of TILE_SIZE 16 y 32 with more data.