GPU Programming CS6023

Fall 2018

Assignment 2

Report

Author: Varun Sundar

EE16B068

16th September 2018

1 Implementation and Technical Notes

The code was run on a *GTX 1080 Ti* and not the cluster of *K'40s*. This could be a reason for variance in the optimal configuration found for the GPUs. However, the code does not make any hardware assumptions and can therefore be run on any suitable cluster.

CUDA version 9.0 was used to compile the code, with `nvcc` as the device compiler and `gcc` as the host compiler.

2 Question 1 and 2

Matrix Multiplication via thread-wise access to global memory was used.

We implement two matrix multiplication kernels: one for column major access (fastest varying y) and one for row major access (fastest varying x). We utilise *Cuda Events* to time the two runs. The results may be replicated by running,

```
./ee16b068_1.cu
```

2.1 Run Timing and Interpretation

```
Time spent in col maj 28395.593750

Time spent in row maj 46189.101562
```

This can be attributed to the fact that DRAM bursts are more efficiently accessed with simultaneous access to the same row. For parallel access this means each thread should ideally access different rows with time, but the same row spatially. Hence, column major outperforms row major.

For questions 2,3 , we shall consider column major timing only.

2.2 Code Blocks (pertinent only)

```
5  __global__ void MatrixMulKernel_col_maj(double* M, double* N, double* P,
       int Width) {
6      // Calculate the row index of the P element and M
7      int Row = blockIdx.y*blockDim.y+threadIdx.y;
8      // Calculate the column index of P and N
9      int Col = blockIdx.x*blockDim.x+threadIdx.x;
10
11     if ((Row < Width) && (Col < Width)) {
12             float Pvalue = 0;
13         for (int k = 0; k < Width; ++k) {
14             Pvalue += M[Row*Width+k]*N[k*Width+Col];
15         }
```

```
16              P[Row*Width+Col] = Pvalue;
17          }
18      }
19
20  __global__ void MatrixMulKernel_row_maj(double* M, double* N, double* P,
        int Width) {
21      // Calculate the row index of the P element and M
22      int Row = blockIdx.y*blockDim.y+threadIdx.x;
23      // Calculate the column index of P and N
24      int Col = blockIdx.x*blockDim.x+threadIdx.y;
25
26      if ((Row < Width) && (Col < Width)) {
27              float Pvalue = 0;
28          for (int k = 0; k < Width; ++k) {
29              Pvalue += M[Row*Width+k]*N[k*Width+Col];
30          }
31              P[Row*Width+Col] = Pvalue;
32          }
33      }
```

3 Question 2

We now use similar code to question 1, except, varying threadsPerBlock as a parameter, from 4 to 32. (We could use up to 1024 threads a block, and for two dimensional threads, this is 32^2). Nevertheless, we include upto 128, and note any errors.

Corresponding plot in **Figure 1**

The noted performance is:

4 Question 3

We utilise the shared memory, however, not via tiling, but rather just using dependencies of each block. Assuming that a block is of size $TileWidth * TileWidth$, the numbers access directly (no iterartions) are $2 * TileWidth * MatrixWidth$. Note that, here for the sake of simplicity we are working with square matrices. We find the time taken for the same multiplication as in **Question 1** with this new strategy.

In the first part of this questionn, we use a $16 * 16$ matrix, and obtain the following:

```
Time spent in col maj 0.019232
```

In the second case, we use the original size of 8192. Considering the smaller size of shared memory (as opposed to global memory, we expect this to fail, returning a null matrix output).

cudaPeekErrors reveals the lack of sufficient shared memory in the second case.

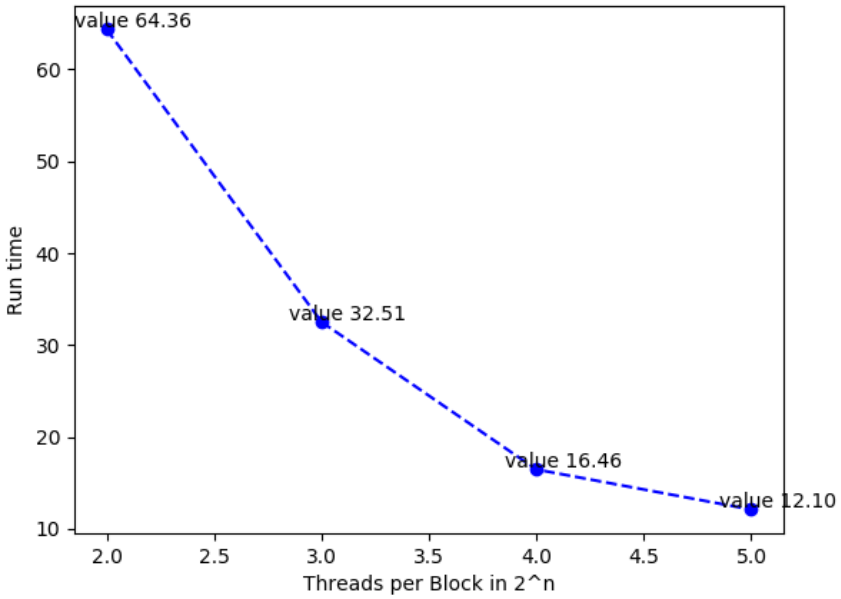

Figure 1: Naive Matrix Multiplication versus Threads per Block for **GTX 1080 Ti**

```
cudaError_t err1 = cudaPeekAtLastError();//To capture last error in
   function call
```

4.1 Code Changes

We use the concept of templates and external shared registers to dynamically declare arrays. Reference : CUDA Documentation.

```
37        __global__ void MatrixMulKernel_col_maj(double* M, double* N,
             double* P, int Width) {
38    extern __shared__ double buffer[];
39
40    double *ds_M = &buffer[0]; // TILE_WIDTH WIDTH
41    double *ds_N = &buffer[TILE_WIDTH*Width]; // WIDTH TILE_WIDTH
42
43    //__shared__ float ds_M[Width][Width];
44    //__shared__ float ds_N[Width][Width];
45    int bx = blockIdx.x;  int by = blockIdx.y;
46    int tx = threadIdx.x; int ty = threadIdx.y;
47    int Row = by * blockDim.y + ty;
48    int Col = bx * blockDim.x + tx;
49
50
51    // Loop over the M and N tiles required to compute the P element
52    for (int p = 0; p < Width/TILE_WIDTH; ++p) {
53    // Collaborative loading of M and N tiles into shared memory
```

```
54      ds_M[ty*Width + tx + p*blockDim.x ] = M[Row*Width + p*TILE_WIDTH+tx
           ];
55      ds_N[ty*TILE_WIDTH + blockDim.y*TILE_WIDTH*p + tx] = N[(p*TILE_WIDTH
           +ty)*Width + Col];
56      __syncthreads();
57      }
58
59      double Pvalue = 0;
60      for (int i = 0; i < TILE_WIDTH; ++i){
61          Pvalue += ds_M[ty*Width + i] * ds_N[i*Width + tx];
62      }
63      __syncthreads();
64      P[Row*Width+Col] = Pvalue;
65  }
```

5 Question 4

Here, we implement tiling for the case of similar square matrices.

We compare the performance boost versus question 1.

```
Time spent in col maj 4015.230225
```

The matrix multiplication takes just 4 seconds, a major improvement over 28 seconds previously. Note that this is due to lower latency in data access and the problem of DRAM locality being efficiently addressed by tiling.

5.1 Code Changes

```
67  __global__ void MatrixMulKernel_col_maj(double* M, double* N, double* Q,
        int Width) {
68      //extern __shared__ double buffer[];
69      //double *ds_M = &buffer[0];
70      //double *ds_N = &buffer[Width*Width];
71
72      __shared__ double ds_M[TILE_WIDTH][TILE_WIDTH];
73      __shared__ double ds_N[TILE_WIDTH][TILE_WIDTH];
74
75      // Generate IDs
76      double Pvalue=0;
77      int bx = blockIdx.x;  int by = blockIdx.y;
78      int tx = threadIdx.x; int ty = threadIdx.y;
79      int Row = by * blockDim.y + ty;
80      int Col = bx * blockDim.x + tx;
81
82
83      // Loop over the M and N tiles required to compute the P element
84      for (int p = 0; p < (Width)/TILE_WIDTH; ++p) {
```

```
85          if ( (Row < Width) && (tx + p*TILE_WIDTH) < Width){
86          // Collaborative loading of M and N tiles into shared memory
87          ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
88          }
89          else{
90              ds_M[ty][tx]=0.0;
91          }
92          if ( (Col < Width) && (ty + p*TILE_WIDTH) < Width){
93          ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
94          }
95          else{
96              ds_N[ty][tx]=0.0;
97          }
98          __syncthreads();
99
100         for (int i = 0; i < TILE_WIDTH; ++i){
101             Pvalue += ds_M[ty][i] * ds_N[i][tx];
102
103         }
104         __syncthreads();
105
106     }
107
108     if ((Row < Width) && (Col < Width)){
109     Q[Row*Width+Col] = Pvalue;
110     }
111 }
```

6 Question 5

We vary the tile size from $4*4$ to $32*32$ to find the most optimal tile size. Considering that the **GTX 1080 Ti** can fit 1024 threadsPerBlock, we expect 32 to be the optimal tilesize.

Note that tilesize being varied independently of the blocksize, ... ie, having a block size as a multiple of the tile size doesnot make a difference as it now effectively acts a bigger tile. Hence the only independent parameter here is the tilesize.

Again, empirically, we observe that $32 * 32$ is the most optimal tile size. This makes sense, since there are greater advantages with a larger tile size, and the only upperbound being the shared memory and the number of threadsPerBlock.

6.1 Run-Times

No major code changes here, besides varying the tile sizes given these optimal *Threads per Block* .

Corresponding plot in **Figure-2**.

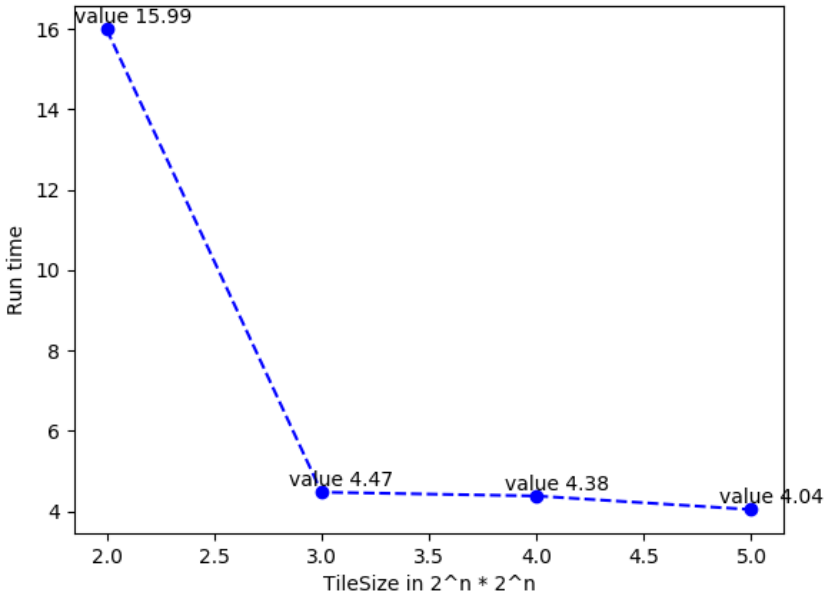

Figure 2: Average Runtime versus Tile Size for **GTX 1080 Ti**

7 Question 6

We generalise tiling for rectangular matrices. The changes made are primarily by matching corresponding row-column sizes as well as the correct boundaries for each tile and index.

This is more evident in the code snippet below. We try this out for a matrix multiplication of $4096 * 8192$ times $8192 * 16384$, and report the following runtime.

Note that this is slightly slower than 1 due to more matrix ops. Further if conditions are put in place to handle the externalities. This therefore can generalise to arbitrary multiplicand sizes.

7.1 Code Changes

```
112 __global__ void MatrixMulKernel_col_maj(double* M, double* N, double* Q,
        int M_r, int N_c, int M_c) {
113     //extern __shared__ double buffer[];
114     //double *ds_M = &buffer[0];
115     //double *ds_N = &buffer[Width*Width];
116
117     __shared__ double ds_M[TILE_WIDTH][TILE_WIDTH];
118     __shared__ double ds_N[TILE_WIDTH][TILE_WIDTH];
119
120     // Generate IDs
121     double Pvalue=0;
122     int bx = blockIdx.x;  int by = blockIdx.y;
123     int tx = threadIdx.x; int ty = threadIdx.y;
124     int Row = by * blockDim.y + ty;
```

```
125     int Col = bx * blockDim.x + tx;
126
127
128     // Loop over the M and N tiles required to compute the P element
129     for (int p = 0; p < (M_c)/TILE_WIDTH; ++p) {
130         if ( (Row < M_r) && (tx + p*TILE_WIDTH) < M_c){
131         // Collaborative loading of M and N tiles into shared memory
132         ds_M[ty][tx] = M[Row*M_c + p*TILE_WIDTH+tx];
133         }
134         else{
135             ds_M[ty][tx]=0.0;
136         }
137         if ( (Col < N_c) && (ty + p*TILE_WIDTH) < M_c){
138         ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*N_c + Col];
139         }
140         else{
141             ds_N[ty][tx]=0.0;
142         }
143         __syncthreads();
144
145         for (int i = 0; i < TILE_WIDTH; ++i){
146             Pvalue += ds_M[ty][i] * ds_N[i][tx];
147
148         }
149         __syncthreads();
150
151     }
152
153     if ((Row < M_r) && (Col < N_c)){
154     Q[Row*N_c+Col] = Pvalue;
155     }
156 }
```

8 Question 7

We use a similar structure as **Question 5** to find the most optimal configuration , here, dictated by the tile size. An array stores the time taken and empirically finds the best configuration.

We find the following configuration:

```
Optimal time is 32806.507812, threads per block is 32 x 32, tile size is
    32 x 32 blocks per grid is 257 x 1025.
```

9 References

- Classroom lectures and Slides.
- CUDA Dev Documentation.