Matrix Multiplication on GPU with CUDA

Homework 4

Rebah Özkoç 29207

CS 406 Parallel Computing

May 23, 2023

Sabancı Üniversitesi

May 23, 2023

Introduction

In this homework there are two programs given namely ”first.cu” and ”second.cu”. The second program contains the improved version matrixMultiplyOptimised function. Except for this difference both of the programs are the same. They are calculating the multiplication of two 2048x2048 matrices.

Code Outputs and Performance Analysis

The output of the first code compiled with:
nvcc first.cu -Xcompiler -fopenmp -O3 -Xcompiler -O3

```
Number of threads: 1024 (32x32)
Number of blocks: 4096 (64x64)

Time to calculate results on CPU: 28006.119141 ms -- result[10] = 24576.000000 -- result[1000] = 2052096.000000
Time to calculate results on CPU_Opt (single thread): 11066.624023 ms  -- result[10] = 24576.000000 -- result[1000] = 2052096.000000
Time to calculate results on CPU_Opt (32 threads): 1524.736572 ms -- result[10] = 24576.000000 -- result[1000] = 2052096.000000

Time to calculate results on GPU: 52.307648 ms
Effective performance: 305.883 GFlop
Effective bandwith: 0.962 GB
result[10] = 24576.000000 -- result[1000] = 2052096.000000

Time to calculate results on GPU (optimised): 94.373245 ms
Effective performance: 169.540 GFlop
Effective bandwith: 0.497 GB
result[10] = 24576.000000 -- result[1000] = 2052096.000000
```

The output of the second code compiled with:
nvcc second.cu -Xcompiler -fopenmp -O3 -Xcompiler -O3

```
Number of threads: 1024 (32x32)
Number of blocks: 4096 (64x64)

Time to calculate results on CPU: 28077.193359 ms -- result[10] = 24576.000000 -- result[1000] = 2052096.000000
Time to calculate results on CPU_Opt (single thread): 11058.838867 ms  -- result[10] = 24576.000000 -- result[1000] = 2052096.000000
Time to calculate results on CPU_Opt (32 threads): 1529.792236 ms -- result[10] = 24576.000000 -- result[1000] = 2052096.000000

Time to calculate results on GPU: 54.302048 ms
Effective performance: 294.648 GFlop
Effective bandwith: 0.927 GB
result[10] = 24576.000000 -- result[1000] = 2052096.000000

Time to calculate results on GPU (optimised): 19.632383 ms
Effective performance: 814.980 GFlop
Effective bandwith: 2.388 GB
```

```
result[10] = 24576.000000 -- result[1000] = 2052096.000000
```

From the results, it can be observed that every function in both versions of the code is calculating matrix multiplication correctly.

Analysis of The CPU Version

This code computes the matrix multiplication on the CPU with a naive algorithm first. Then it improves the algorithm in matrixMultiplyCPU_Opt function. This function takes the transpose of the matrix b and computes the multiplication on the transpose of b. This improvement changes the array access to row-major access from the column-major access and it increases the utilization of cache usage by reducing the unnecessary cache transfers.

It also uses OpenMP to parallelize the outermost loop. Other than these improvements the function uses ___restrict___keyword and loop unrolling to improve the performance.

These improvements yield 2.22 times faster execution on the single thread execution than the naive algorithm. The parallelization with 32 threads yields 16 times faster execution than the naive algorithm.

The second version of the code does not modify the functions that compute the matrix multiplication on the CPU. Thus we have the same performance as the first version.

Analysis of The Naive GPU Version

```
__global__ void matrixMultiplySimple(float *a, float *b, float *c, int width) {
  int col = threadIdx.x + blockIdx.x * blockDim.x;
  int row = threadIdx.y + blockIdx.y * blockDim.y;

  float result = 0;

  if (col < width && row < width) {
    for (int k = 0; k < width; k++) {
      result += a[row * width + k] * b[k * width + col];
    }
    c[row * width + col] = result;
  }
}

int main(){
  ...
  int width = 2048; // Define width of square matrix
  int sqrtThreads = sqrt(THREADS_PER_BLOCK);
  int nBlocks = width / sqrtThreads;
  ...
  dim3 grid(nBlocks, nBlocks, 1);
  dim3 block(sqrtThreads, sqrtThreads, 1); // Max number of threads per block
  ...
  matrixMultiplySimple<<<grid, block>>>(a_d, b_d, c_d, width);
}
```

This code creates a kernel function to calculate the square matrix multiplication. It runs the function on GPU with 1024 threads per block and 4096 blocks. Each thread performs a

dot product of the corresponding row from the first matrix and the column from the second matrix. The result is stored in the c matrix. The matrix a and b are stored in the global memory of the GPU and the result is written back to the global memory. Access to the global memory is slower than the access to the shared memory and this function is open to improvements.

A GPU can parallelize lots of threads on different streaming multiprocessors and the thread transition overhead inside a block is minimal compared to CPUs. Thus, the naive GPU version of the code is 29 times faster than CPU_Opt with 32 threads.

The second version of the code does not modify this function. Thus we have the same performance as the first version.

Analysis of The Optimized GPU Version

```
__global__ void matrixMultiplyOptimised(float *a, float *b, float *c, int width) {
  // Allocate 2D tiles in shared memory
  __shared__ float s_a[TILE_WIDTH][TILE_WIDTH];
  __shared__ float s_b[TILE_WIDTH][TILE_WIDTH];

  // Calculate row and column index of element
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  float result = 0;

  // Loop over tiles of input in phases
  for (int p = 0; p < width / TILE_WIDTH; p++) {
    // Collaboratively load tiles into shared memory
    s_a[threadIdx.x][threadIdx.y] = a[row * width + (p * TILE_WIDTH + threadIdx.x)
    ];
    s_b[threadIdx.x][threadIdx.y] = b[col + width * (p * TILE_WIDTH + threadIdx.y)
    ];

    __syncthreads();     // Wait until all data is loaded before allowing any
    threads in the block to continue

    // Dot product between row of s_a and column of s_b
    for (int i = 0; i < TILE_WIDTH; i++) {
      result += s_a[i][threadIdx.y] * s_b[threadIdx.x][i];
    }

    __syncthreads();    // Wait until all calculations are finished before
    allowing any threads in the block to continue
  }

  // Write result
  c[row * width + col] = result;
}
```

The optimized version of the GPU computation in the first file performs 1.8 times worse than the naive GPU computation. Although it tries to reduce the memory access time by using the shared memory, the memory access pattern causes **bank conflict**, and the performance of the kernel drops significantly.

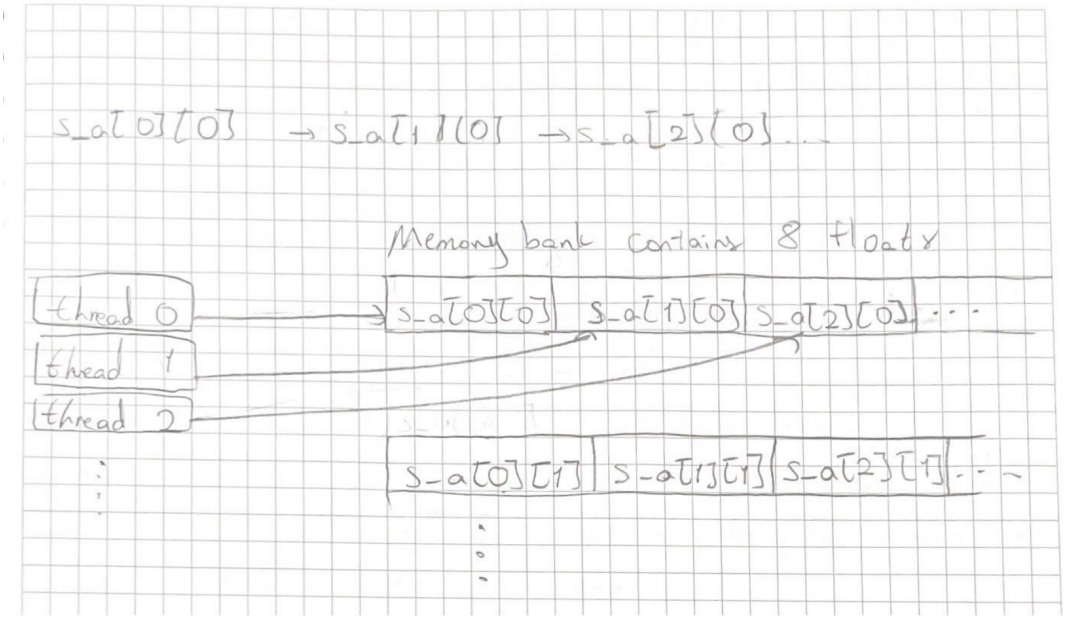

Figure 1: Bank Conflict

In order to obtain high-capacity memory usage during simultaneous operations, shared memory is segmented into uniformly proportioned memory units which are called memory banks. A bank conflict occurs when more than one thread in a WARP tries to access different words in a memory bank. If k threads try to reach different words in a bank this is called **k-way bank conflict**.

There is more than one bank conflict in this code. The first one is on line 15 with `s_a[threadIdx.x][threadIdx.y]`

In this one, the access pattern to s_a goes like this:

Also, the same problem occurs on `s_b[threadIdx.x][threadIdx.y]` on line 16.

These two instructions run width/TILE_WIDTH times in a thread.

There is another bank conflict on `s_b[threadIx.x][i]` on line 22. This is more problematic since it runs 32 (TILE_WIDTH) times more than the previous ones.

There is no bank conflict on `s_aa[i][threadIdx.y]` because threads are accessing the same place within a bank. This is called broadcasting.

In the second version of the code, these problems are solved.

Analysis of The Fixed Optimized GPU Version

```
1  __global__ void matrixMultiplyOptimised(float *a, float *b, float *c, int width) {
2    // Allocate 2D tiles in shared memory
3    __shared__ float s_a[TILE_WIDTH][TILE_WIDTH];
4    __shared__ float s_b[TILE_WIDTH][TILE_WIDTH];
5
6    // Calculate row and column index of element
7    int row = blockIdx.y * blockDim.y + threadIdx.y;
8    int col = blockIdx.x * blockDim.x + threadIdx.x;
9
```

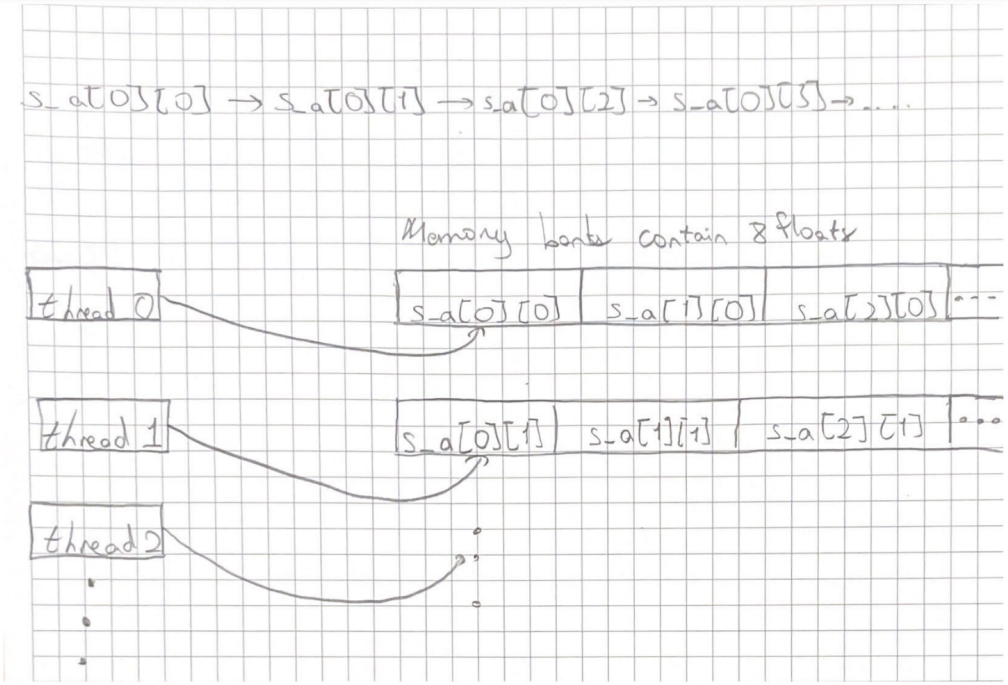

Figure 2: Bank Conflict Solved

```
10  float result = 0;
11
12  for (int p = 0; p < width / TILE_WIDTH; p++) {
13    // Collaboratively load tiles into shared memory
14    s_a[threadIdx.y][threadIdx.x] = a[row * width + (p * TILE_WIDTH + threadIdx.x)
      ];
15    s_b[threadIdx.y][threadIdx.x] = b[col + width * (p * TILE_WIDTH + threadIdx.y)
      ];
16
17    __syncthreads();     // Wait until all data is loaded
18
19    // Dot product between row of s_a and column of s_b
20    for (int i = 0; i < TILE_WIDTH; i++) {
21      result += s_a[threadIdx.y][i] * s_b[i][threadIdx.x];
22    }
23    __syncthreads();    // Wait until all calculations are finished before
      allowing any threads in the block to continue
24  }
25  // Write result
26  c[row * width + col] = result;
27 }
```

In this version, the memory access pattern is changed and it is 2.7 times faster than the naive GPU computation.

There is no bank conflict in this code and it modified the `s_a[threadIdx.x][threadIdx.y]` access to `s_a[threadIdx.y][threadIdx.x]` on line 14.

The memory access pattern to s_a array is like this:

Also, the same problem is solved on line 15 by modifying the `s_b[threadIdx.x][threadIdx.y]` to `s_b[threadIdx.y][threadIdx.x]`.

The last bank conflict on `s_b[threadIx.x][i]` on line 21 is solved by changing the access to `s_b[threadIx.y][i]`.

Thus, all of the bank conflicts are solved in the second version and the implementation is working like expected.