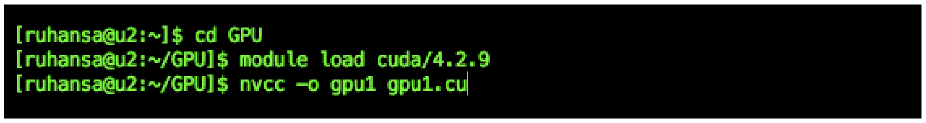
CSE603 Assignment2

Ruhan Sa 50060400

November 9, 2012

Aim: Compute the product of matrix A and matrix B. Result stored into C.
Input: Both A and B are generated using dynamic allocation, using malloc function in C. Size of A and B are determined in the command line as input argument of main function. In normal experiment phases, the element is derived from random float numbers ranges from 0 to 2.
Verification: This is done using matrix with unit elements and the result is verified by checking wether elements of C equals to the matrix size or not. If so the implementation is verified, if not the implementation is wrong.
Maximum matrix size: The maximum matrix size here denotes the maximum matrix size whose time consumption is still observable during the assignment cycle. The walltime of CCR are within 24 hours in all experiments below.
Compilation: All the compilations are done using following script in the CCR front-end.



```
[ruhansa@u2:~]$ cd GPU
[ruhansa@u2:~/GPU]$ module load cuda/4.2.9
[ruhansa@u2:~/GPU]$ nvcc -o gpu1 gpu1.cu
```

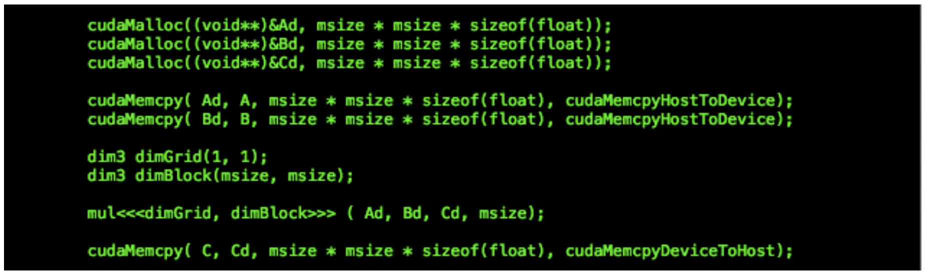
Figure 1: Compilation script

1 One block with one thread per element

1.1 A Glimpse of Source Code

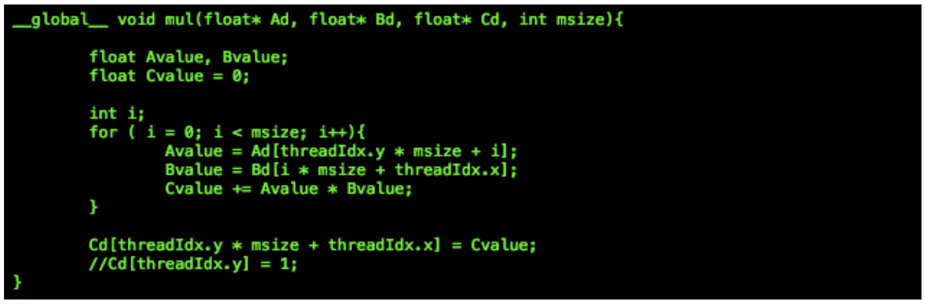
Main part of the source code and the scripts are shown in Figure 2 and Figure 3. In this part, the block size is determined by the matrix size, which is denoted as variable "*msize*" in the code. Threads within one block is scaled in $msize \times msize$ fashion.

The script shows the command line of running verification as well as the experiments on CCR. As Figure 4 shows the matrix size is determined in the command line. This way the code can be more flexible than defining the size in the code.



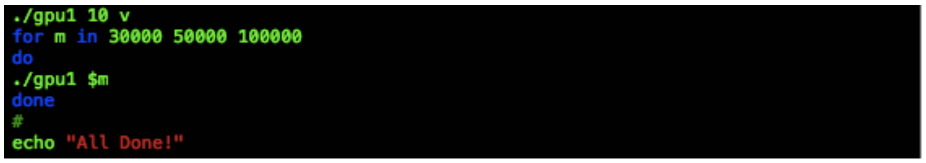
```
        cudaMalloc((void**)&Ad, msize * msize * sizeof(float));
        cudaMalloc((void**)&Bd, msize * msize * sizeof(float));
        cudaMalloc((void**)&Cd, msize * msize * sizeof(float));

        cudaMemcpy( Ad, A, msize * msize * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy( Bd, B, msize * msize * sizeof(float), cudaMemcpyHostToDevice);

        dim3 dimGrid(1, 1);
        dim3 dimBlock(msize, msize);

        mul<<<dimGrid, dimBlock>>> ( Ad, Bd, Cd, msize);

        cudaMemcpy( C, Cd, msize * msize * sizeof(float), cudaMemcpyDeviceToHost);
```

Figure 2: Host side code



```
__global__ void mul(float* Ad, float* Bd, float* Cd, int msize){

        float Avalue, Bvalue;
        float Cvalue = 0;

        int i;
        for ( i = 0; i < msize; i++){
                Avalue = Ad[threadIdx.y * msize + i];
                Bvalue = Bd[i * msize + threadIdx.x];
                Cvalue += Avalue * Bvalue;
        }

        Cd[threadIdx.y * msize + threadIdx.x] = Cvalue;
        //Cd[threadIdx.y] = 1;
}
```

Figure 3: Device side code



```
./gpu1 10 v
for m in 30000 50000 100000
do
./gpu1 $m
done
#
echo "All Done!"
```

Figure 4: partial script

1.2 Result with the increase of matrix size

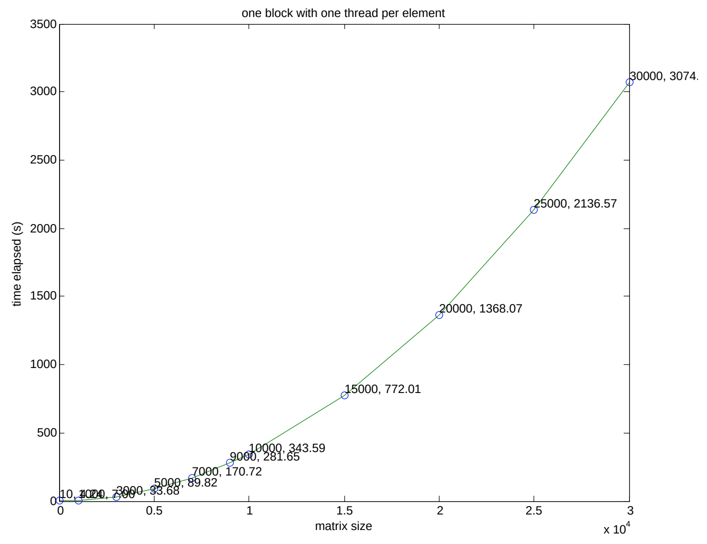


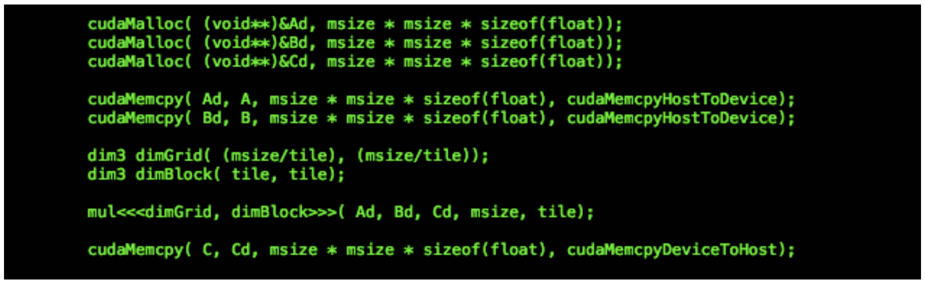
Figure 5: time consumption with the increase of matrix size.

Figure 5 shows the time consumption with the increase of matrix size. The maximum matrix size is 30000×30000.

2 Multiple blocks with one thread per element

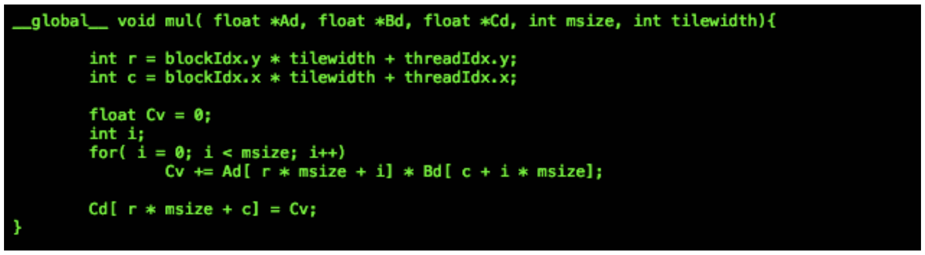
2.1 A Glimpse of Source Code

Main part of the source code and the scripts are shown in Figure 6, Figure 7 and Figure 8. As the code shows, every block solves $tile \times tile$ elements and are layed out uniformly in the Grid. Each element are solved by one thread.



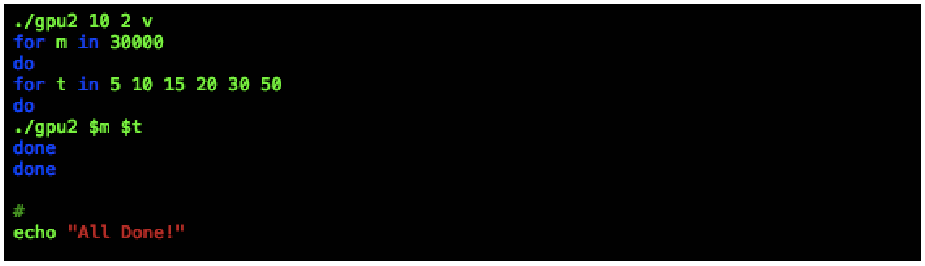
```
        cudaMalloc( (void**)&Ad, msize * msize * sizeof(float));
        cudaMalloc( (void**)&Bd, msize * msize * sizeof(float));
        cudaMalloc( (void**)&Cd, msize * msize * sizeof(float));

        cudaMemcpy( Ad, A, msize * msize * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy( Bd, B, msize * msize * sizeof(float), cudaMemcpyHostToDevice);

        dim3 dimGrid( (msize/tile), (msize/tile));
        dim3 dimBlock( tile, tile);

        mul<<<dimGrid, dimBlock>>>( Ad, Bd, Cd, msize, tile);

        cudaMemcpy( C, Cd, msize * msize * sizeof(float), cudaMemcpyDeviceToHost);
```

Figure 6: Host side code



```
__global__ void mul( float *Ad, float *Bd, float *Cd, int msize, int tilewidth){

        int r = blockIdx.y * tilewidth + threadIdx.y;
        int c = blockIdx.x * tilewidth + threadIdx.x;

        float Cv = 0;
        int i;
        for( i = 0; i < msize; i++)
                Cv += Ad[ r * msize + i] * Bd[ c + i * msize];

        Cd[ r * msize + c] = Cv;
}
```

Figure 7: Device side code



```
./gpu2 10 2 v
for m in 30000
do
for t in 5 10 15 20 30 50
do
./gpu2 $m $t
done
done

#
echo "All Done!"
```

Figure 8: partial script

2.2 Result with the increase of matrix size

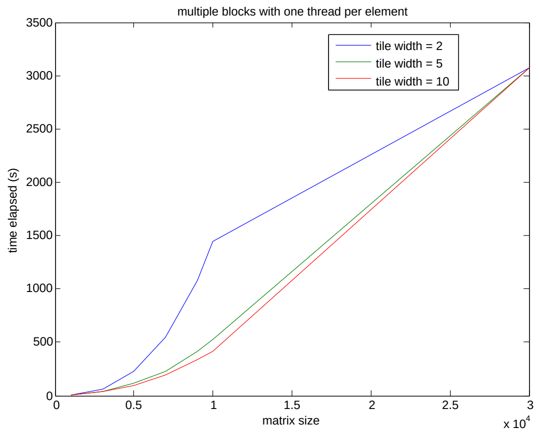
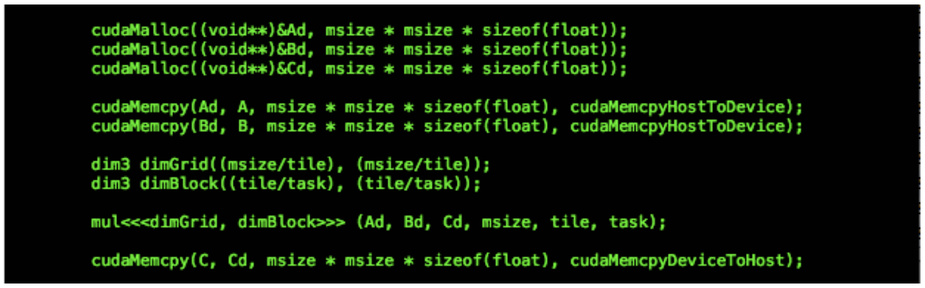


Figure 9: time consumption with the increase of matrix size.

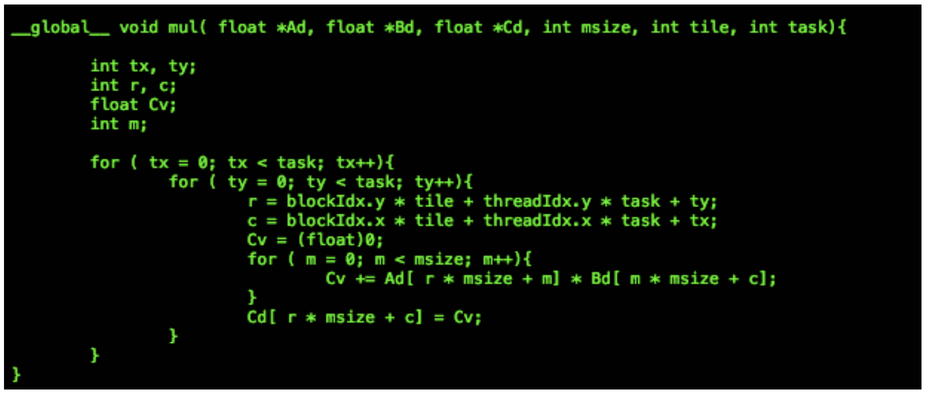
Figure 9 shows the time consumption with the increase of matrix size. The maximum matrix size is 30000×30000. Since, in this part, the time consumption can be influenced by 2 major facts: size of the matrix and the tile width. The experiments are conducted using different "*tile_width*" chosen from $\{2, 5, 10\}$. From the figure we can see that with the increase of tile_width, *i.e.* with the increase of the number of elements solved by per block, the speed goes up. For example when computing 1000×1000 matrix, the time consumption of using 2×2 elements per block is three times larger than using 10×10 elements per block. This is because, one SM can only take 8 blocks, if the granularity is 4 thread/block, the whole SM only reaches $(4 * 8 = 32)$ threads. However if the granularity is 100 thread/block, one SM can hold all blocks, which means that one SM can compute up to 768 (which is the whole capacity according to G80) threads simultaneously.

3 Multiple blocks with one thread multiple element



```
        cudaMalloc((void**)&Ad, msize * msize * sizeof(float));
        cudaMalloc((void**)&Bd, msize * msize * sizeof(float));
        cudaMalloc((void**)&Cd, msize * msize * sizeof(float));

        cudaMemcpy(Ad, A, msize * msize * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(Bd, B, msize * msize * sizeof(float), cudaMemcpyHostToDevice);

        dim3 dimGrid((msize/tile), (msize/tile));
        dim3 dimBlock((tile/task), (tile/task));

        mul<<<dimGrid, dimBlock>>> (Ad, Bd, Cd, msize, tile, task);

        cudaMemcpy(C, Cd, msize * msize * sizeof(float), cudaMemcpyDeviceToHost);
```

Figure 10: Host side code



```
__global__ void mul( float *Ad, float *Bd, float *Cd, int msize, int tile, int task){

        int tx, ty;
        int r, c;
        float Cv;
        int m;

        for ( tx = 0; tx < task; tx++){
                for ( ty = 0; ty < task; ty++){
                        r = blockIdx.y * tile + threadIdx.y * task + ty;
                        c = blockIdx.x * tile + threadIdx.x * task + tx;
                        Cv = (float)0;
                        for ( m = 0; m < msize; m++){
                                Cv += Ad[ r * msize + m] * Bd[ m * msize + c];
                        }
                        Cd[ r * msize + c] = Cv;
                }
        }
}
```

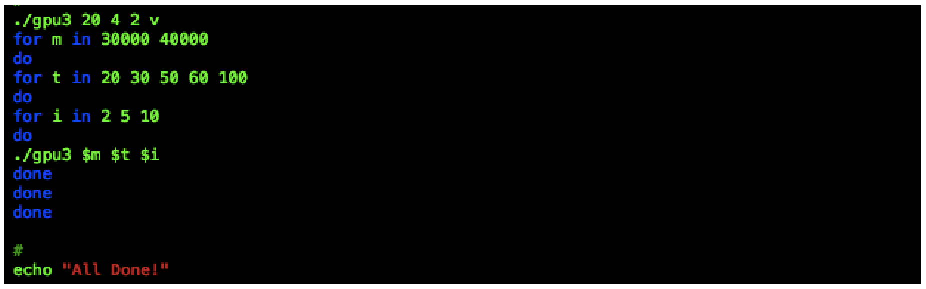
Figure 11: Device side code

3.1 A Glimpse of Source Code

Main part of the source code and the scripts are shown in Figure 10, Figure 11 and Figure 12. As the code shows, block granularity here is determined by $tile \times tile$ and each thread solves $task \times task$ elements.

3.2 Result with the increase of matrix size

In this part the time consumption is determined by three factors, matrix size, the tile_width and the task_width. The experiments tries to cover different



```
./gpu3 20 4 2 v
for m in 30000 40000
do
for t in 20 30 50 60 100
do
for i in 2 5 10
do
./gpu3 $m $t $i
done
done
done

#
echo "All Done!"
```

Figure 12: partial script

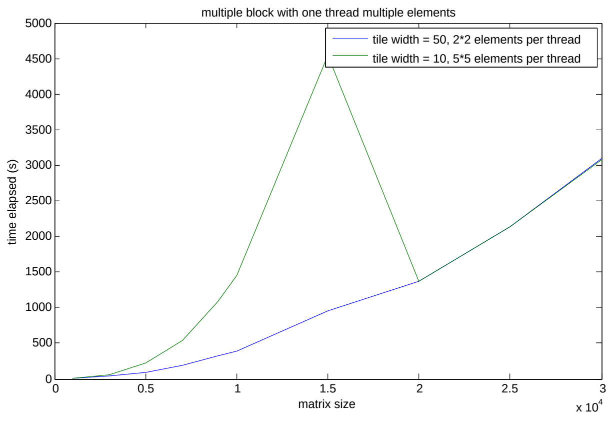
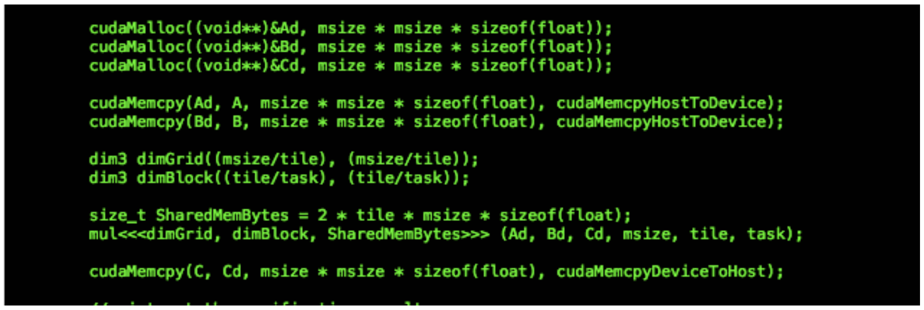


Figure 13: time consumption with the increase of matrix size

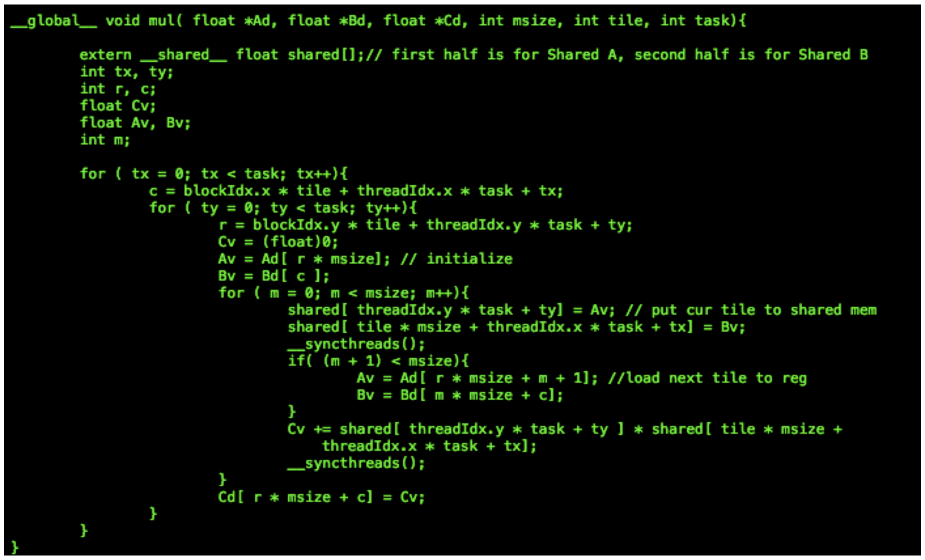
parameters to see the effect. The tile_width is chosen from $\{10, 20, 50\}$ and the task_width is chosen from $\{2, 5, 10\}$. Since the data is too much to plot in one figure, Figure 13 shows only the two extreme cases, the fastest one, which has 50×50 tile and 2×2 elements per thread granularity, and the slowest one, which has 10×10 tile and 5×5 elements per thread granularity. However, when the matrix size exceeds 2000×2000, the time consumptions of all granularity are similar. The maximum matrix size here is 30000×30000.

4 Double Buffering



```
        cudaMalloc((void**)&Ad, msize * msize * sizeof(float));
        cudaMalloc((void**)&Bd, msize * msize * sizeof(float));
        cudaMalloc((void**)&Cd, msize * msize * sizeof(float));

        cudaMemcpy(Ad, A, msize * msize * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(Bd, B, msize * msize * sizeof(float), cudaMemcpyHostToDevice);

        dim3 dimGrid((msize/tile), (msize/tile));
        dim3 dimBlock((tile/task), (tile/task));

        size_t SharedMemBytes = 2 * tile * msize * sizeof(float);
        mul<<<dimGrid, dimBlock, SharedMemBytes>>> (Ad, Bd, Cd, msize, tile, task);

        cudaMemcpy(C, Cd, msize * msize * sizeof(float), cudaMemcpyDeviceToHost);
```

Figure 14: Host side code

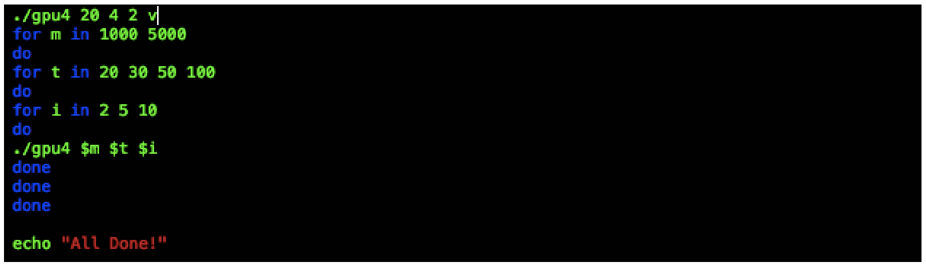


```
__global__ void mul( float *Ad, float *Bd, float *Cd, int msize, int tile, int task){

        extern __shared__ float shared[];// first half is for Shared A, second half is for Shared B
        int tx, ty;
        int r, c;
        float Cv;
        float Av, Bv;
        int m;

        for ( tx = 0; tx < task; tx++){
                c = blockIdx.x * tile + threadIdx.x * task + tx;
                for ( ty = 0; ty < task; ty++){
                        r = blockIdx.y * tile + threadIdx.y * task + ty;
                        Cv = (float)0;
                        Av = Ad[ r * msize]; // initialize
                        Bv = Bd[ c ];
                        for ( m = 0; m < msize; m++){
                                shared[ threadIdx.y * task + ty] = Av; // put cur tile to shared mem
                                shared[ tile * msize + threadIdx.x * task + tx] = Bv;
                                __syncthreads();
                                if( (m + 1) < msize){
                                        Av = Ad[ r * msize + m + 1]; //load next tile to reg
                                        Bv = Bd[ m * msize + c];
                                }
                                Cv += shared[ threadIdx.y * task + ty ] * shared[ tile * msize +
                                    threadIdx.x * task + tx];
                                __syncthreads();
                        }
                        Cd[ r * msize + c] = Cv;
                }
        }
}
```

Figure 15: Device side code

4.1 A Glimpse of Source Code

Main part of the source code and the scripts are shown in Figure 14, Figure 15 and Figure 16. The Device side code shows that the double buffering is accomplished by simultaneously reading and computing, *i.e.* read the next tile and compute the current tile simultaneously.



```
./gpu4 20 4 2 v
for m in 1000 5000
do
for t in 20 30 50 100
do
for i in 2 5 10
do
./gpu4 $m $t $i
done
done
done

echo "All Done!"
```

Figure 16: partial script

4.2 Result with the increase of matrix size

In this part the time consumption is also determined by matrix size, tile_width and task_width. Tile_width is chosen from $\{20, 30, 50, 100\}$ and task_width is chosen from $\{2, 5, 10\}$. Figure 17 shows the fastest among these experiments, where $tile\_width = 30$ and $task\_width = 10$. Here the maximum matrix size is 30000×30000. However, the difference between time consumption of taking different parameters doesn't vary too much. For example, when the matrix size is 5000×5000 the time consumptions of taking different parameters are shown in Figure 18.

5 Comparison with MPI and OpenMP and all the above GPU paradigms

Figure 19 shows the performance comparison with OpenMP and MPI when computing matrix size of 1000×1000 and 5000×5000. (Since the computational time of OpenMP for 5000×5000 matrix is too large to observe, the second group of bars misses one bar.) Here the MPI performance was observed when using 20 processors, OpenMP performance was observed using 20 processors, the $gpu1$ bar indicates the "One block with one thread per element" method, $gpu2$ indicates the "Multiple block with one thread per element" method, $gpu3$ indicates the "Multiple block with one thread multiple element" method and $gpu4$ indicates the "double buffering" method. The performance is chosen when each method performs the best in experiment. For GPU methods, the computational time of their maximum matrix size, which is 30000×30000, are nearly equal with each other.

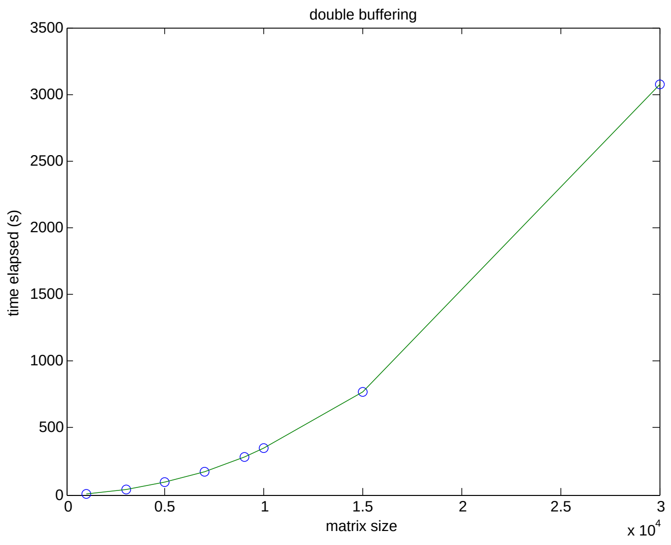


Figure 17: time consumption with the increase of matrix size

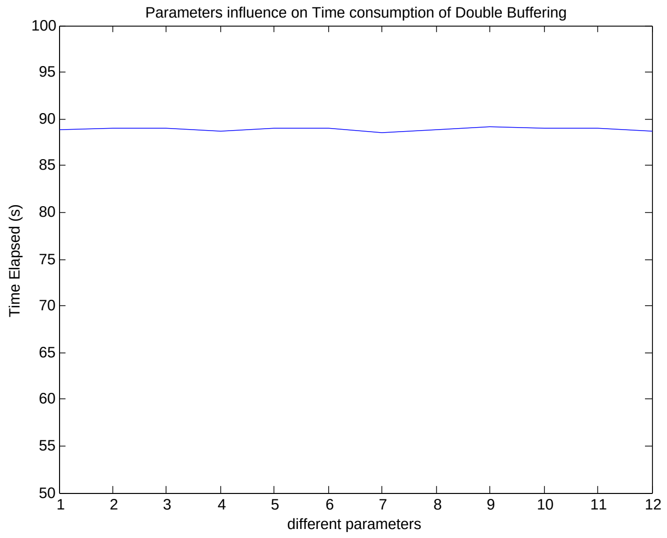


Figure 18: time consumption with the increase of matrix size

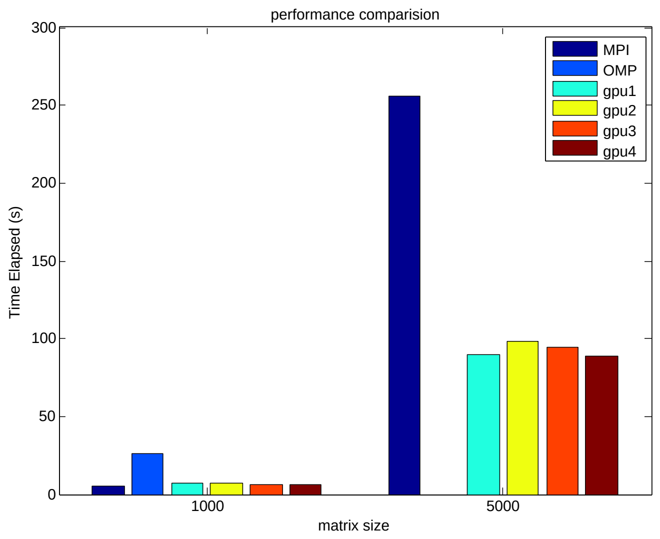


Figure 19: comparison with MPI, OpenMP and all the above methods.