CSC4005 2023 Fall Project 2 Report

Tianci Hou 121090184

1 Efficient Dense Matrix Multiplication

1.1 Set up

Environment: Refer to course GitHub page.

Type	Compilation flags
Naive	`-O2`
Locality	`-O2`
Simd	`-O2 -mavx2`
OpenMP	`-O2 -fopenmp -mavx2`
MPI	`-O2 -fopenmp -mavx2`
CUDA	default
OpenACC	`-acc`

(a) Compilation Flags

```
$ cd path/to/project
$ mkdir build
$ cd build
$ cmake ..
$ make -j4
$ cd ..
# Running on cluster
$ sbatch src/sbach.sh
# Or use src/test_submit.sh
# All outputs will saved test/
```

(b) Build Commands

Figure 1: Compilation and Build Information

1.2 Overall performance

Method	**Matrices 1024*1024**	**Matrices 2048*2048**
Naive	8165 ms	89250 ms
Memory Locality	565 ms	4797 ms
SIMD + Memory Locality	230 ms	2194 ms
OpenMP + SIMD + Memory Locality (32 threads)	41 ms	202 ms
MPI + OpenMP + SIMD + Memory Locality (total 32 threads best)	31 ms	184 ms
MPI + OpenMP + SIMD + Memory Locality (total 64 threads best)	31 ms	171 ms
Bonus[1]: CUDA (Global Memory)	4.43 ms	34.73 ms
Bonus[1]: CUDA (Shared Memory)	3.52 ms	27.53 ms
Bonus[1]: CUDA (Shared + Merge Threads)	1.45 ms	11.01 ms
Bonus[1]: OpenACC	4 ms	35 ms

Table 1: Matrix Multiplication Times

[1] *Computation time: the time of initialization of malloc and memcpy between host and device is ignored.*

1.3 Speed up and Efficiency

Method	Matrices 1024*1024	Matrices 2048*2048
Memory Locality	1345.13%	1761.70%
SIMD + Memory Locality	3450.00%	3967.91%
OpenMP + SIMD + Memory Locality (32 threads)	19814.63%	44083.17%
MPI + OpenMP + SIMD + Memory Locality (total 32 threads best)	26238.71%	48405.43%
MPI + OpenMP + SIMD + Memory Locality (total 64 threads best)	26238.71%	52092.98%
Bonus: CUDA (Global memory)	184211.51%	256882.44%
Bonus: CUDA (Shared memory)	231860.23%	324091.79%
Bonus: CUDA (Shared + Merge Threads)	563003.45%	810526.70%
Bonus: OpenACC	204025.00%	254900.00%

Table 2: Speed Up (speed up factor - 100%) compared to Naive approach

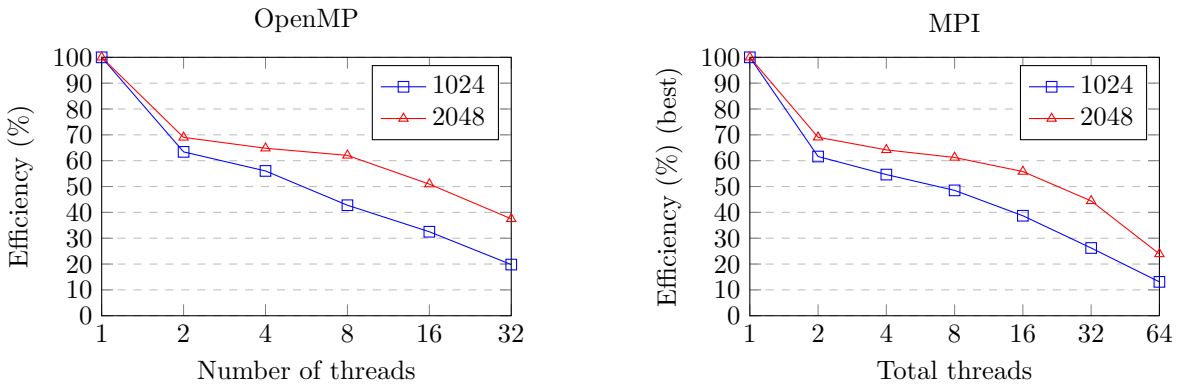

Figure 2: Efficiency

1.4 Correctness

I write a simple shell script to test all of my implementations. Also I generate 30 random test cases (N, M, K are all randomized). The outputs including bonus part are same as the naive approach.

1.5 Task1: Memory Locality

1.5.1 Problems with Naive approach

In the naive approach, it uses the order i, j, k to get the multiplication of Matrix A and Matrix B. As the figure shown below, the number of cache misses will be large since when finding the elements in Matrix B.

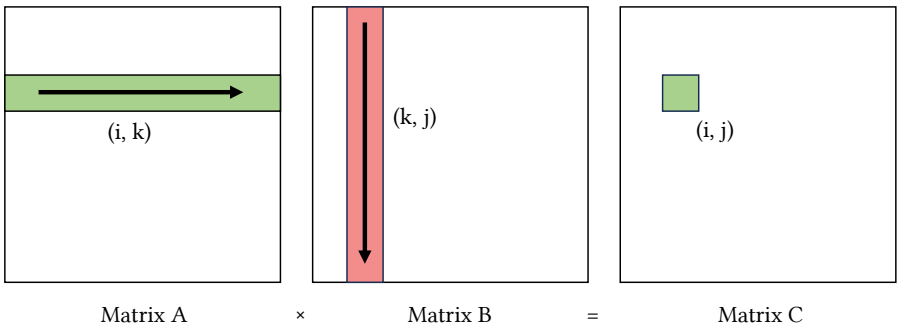

Figure 3: Naive Order (i,j,k)

1.5.2 Problems with Matrix implementation

In the file `matrix.cpp`, we can found the matrix is not organized as the one dimensional array. So the rows between are not contiguous in memory.

1.5.3 Solution 1: Change order without additional memory

To overcome the issues with matrix implementation, we just declare some pointers pointing to the first element of the row we are doing operation. Lets change the order of computation to (k,i,j) or (i,k,j). And as the figure shown below, we can found every elements are visiting continuously in memory.

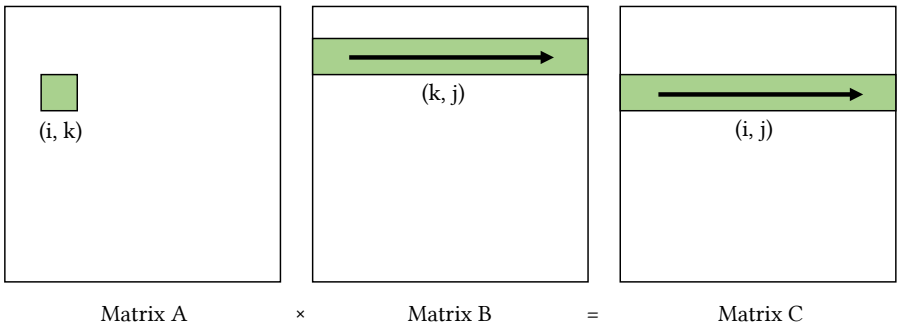

Figure 4: Change Order (i,k,j) or (k,i,j) without additional memory

1.5.4 Solution 2: Change order without additional memory + Tiling

We can divided them into some small blocks. And set the block size, which is just the length of one dimension of the block, is set to the multiple of cache line.

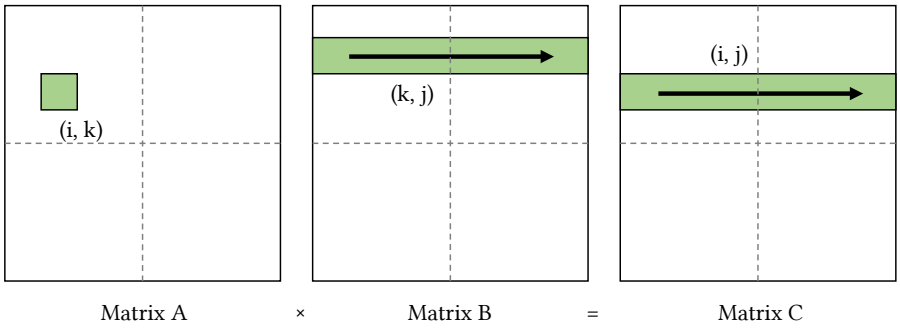

Figure 5: Change Order (i,k,j) or (k,i,j) without additional memory + Tiling

1.5.5 Solution 3: Transpose

We can first get the transpose of Matrix B. Then we just follow the order (i,j,k) but now we access (j,k) in Matrix B^T instead of (k,j) in Matrix B. So we need addition memory to store B^T.

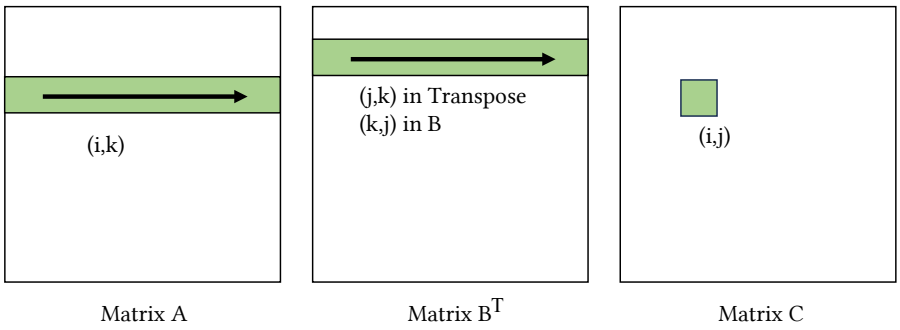

Figure 6: Order (i,j,k) with B^T

1.5.6 Experimental results

We use +C representing Change Order without additional memory, +T representing Tiling. Also, to make it a fairness comparison, for solution of transpose, we use the same implementation in `matrix.cpp`.

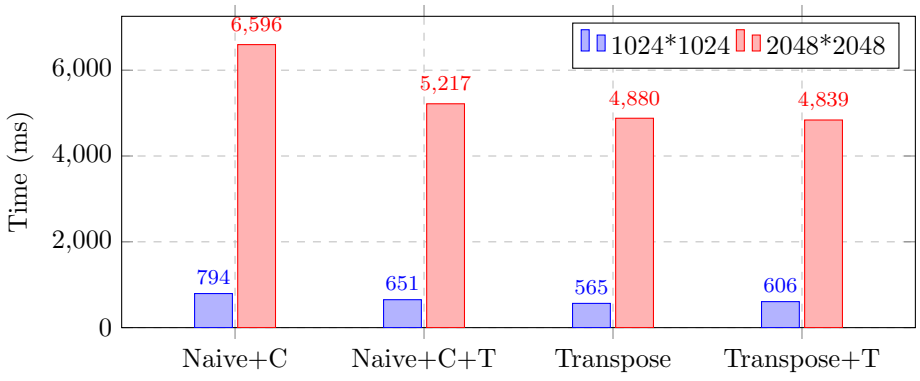

Figure 7: Comparison of Execution Time using different solutions

1.5.7 Profiling results

Let's compare Naive solution and Transpose solution.

Method	CPU-Cycle	Cache Miss	Page Fault	Time (ms)
Navie	34K	2K	113	8165
Transpose	3K	1K	31	565

Table 3: Profiling results of Naive and Transpose (1024*1024)

We can found that we decrease a lot of cache misses and page faults. But the key reason that we run much faster is we decrease CPU-Cycle.

Looking at the detailed results, we can find 24.21% of the CPU-Cycles are wasting on the operation `Matrix::operator[]`. See the detailed implementations of this operation, we can find it use `if` statement to check if our required data is out of the boundary. It will indeed waste a lots of time.

Also not like the one dimension array, it waste a lots time to first find the row and access to the column.

1.6 Task2: Data-Level Parallelism

1.6.1 Solution 1: Upon Naive+C

Extend (i,k) into 8 integers in Matrix A as A256 and set 8 integers (k,[j,j+7]) in Matrix B as B256. Do the multiplication and store back to (i,[j,j+7]) in Matrix C.

For lengths that are not multiples of 8, we directly calculate the remaining values.

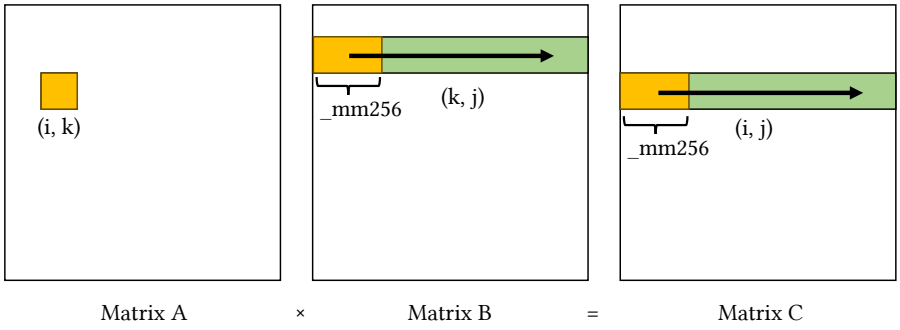

Figure 8: Change Order + SIMD

1.6.2 Solution 2: Upon Transpose

Still, first do the transpose. Now set 8 integers (i,[k,k+8]) in Matrix A as A256, and set 8 integers (j,[k,k+8]) in Matrix B^T as BT256. Do the multiplication and get C256. Get sum of the 8 integers in C256 and store it back to (i,j) in Matrix C.

Here are some tricks on how to get the sum of 8 integers in the vector register. For example using `_mm256_hadd_ps`. I only use `_mm256_extract_epi32` to get values and use simple add operators to sum up them.

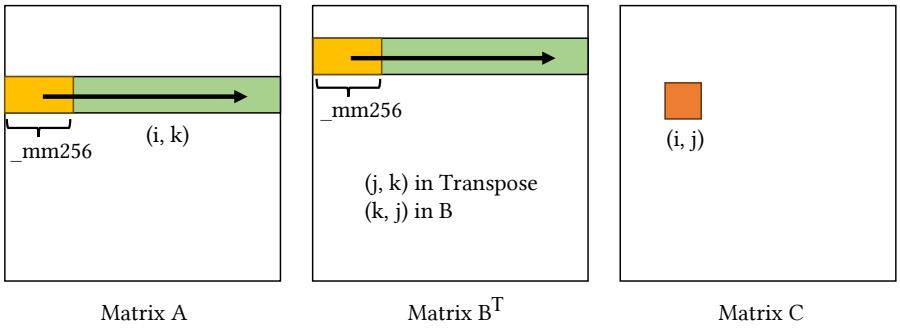

Figure 9: Change Order + SIMD

1.6.3 Experimental results

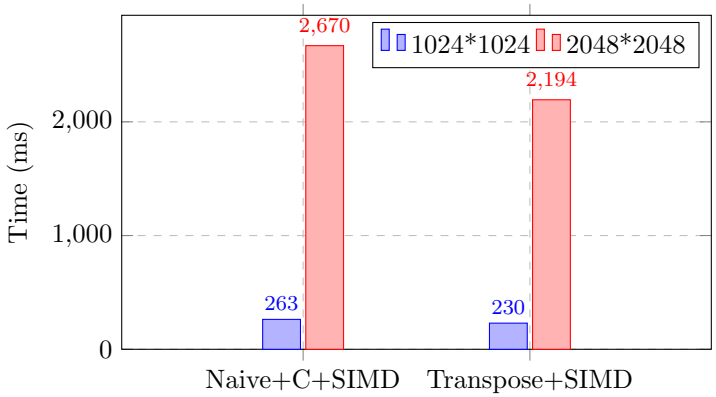

Figure 10: Comparison of Execution Time using different solutions

1.6.4 Profiling results

Method	CPU-Cycle	Cache Miss	Page Fault	Time (ms)
Transpose	3K	1K	31	565
Transpose+SIMD	1K	1K	42	230

Table 4: Profiling results of Transpose and Transpose+SIMD (1024*1024)

We can find that after we use SIMD, the number of CPU-Cycles decreases. It is obvious that when we compress 8 integers into 1 variable and use few cycles for adding them.

1.7 Task3: Thread-Level Parallelism

1.7.1 Solution 1: Upon Naive+C+SIMD

Easy to implement - Just add `#pragma omp parallel for` before the for loops. Since we need to load and store, so we just do the parallel for the outer loop. In other words, I divided matrix A by rows.

1.7.2 Solution 2: Upon Transpose+SIMD

Hard to implement. We need to prepare the `_mm256` arrays. And after my test, just adding `#pragma` shows poor and unstable performance. So we need to split them manually.

1.7.3 Experimental results

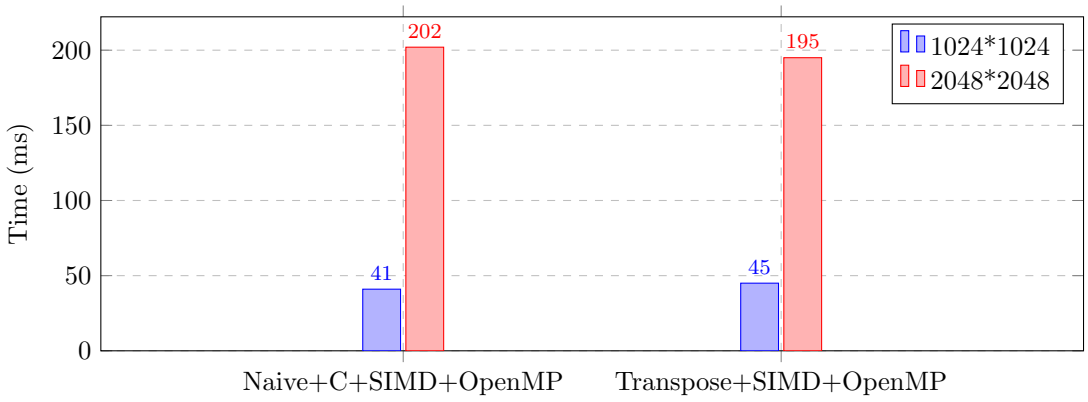

Figure 11: Comparison of Execution Time using different solutions

1.7.4 Profiling results

Threads	CPU-Cycle	Cache Miss	Page Fault	Time (ms)
1	1K	1K	20	260
2	2K	1K	18	205
4	2K	1K	19	116
8	3K	1K	61	76
16	4K	1K	60	50
32	6K	1K	64	41

Table 5: Profiling results of task 3 (1024*1024)

It is hard to use perf to do the profiling. But there are some interesting things. First CPU-Cycle increases. And the largest proportion is Unknown. And self for it is 0%, children for it is about 60%.

1.8 Task4: Process-Level Parallelism

1.8.1 Solution: Upon Naive+C+SIMD+OpenMP

First use the same pre-techniques.

To make it easier, I just implemented this method. Due to the implementation of the matrix, if we do not manually create a buffer, we cannot send all the values in the matrix at once, so we can only send the values of the matrix row by row.

So there are two sending methods, the first is blocking sending, and the second is immediate sending. I have also conducted experiments on both.

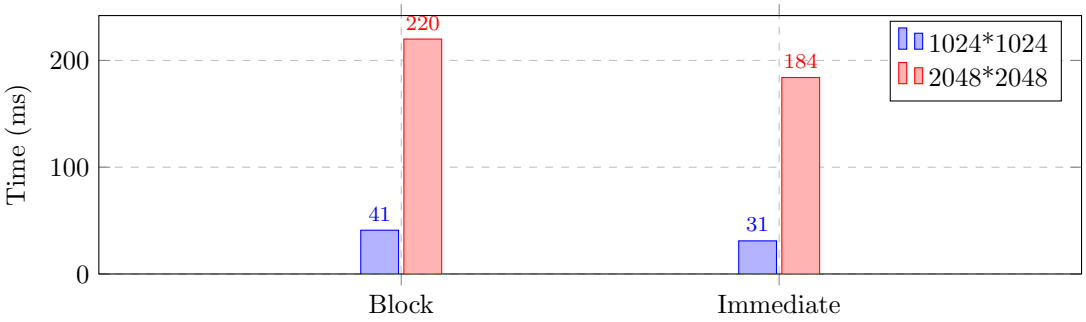

Figure 12: Comparison of best Execution Time using different methods of sending

So I use the immediate sending.

1.8.2 Experimental results

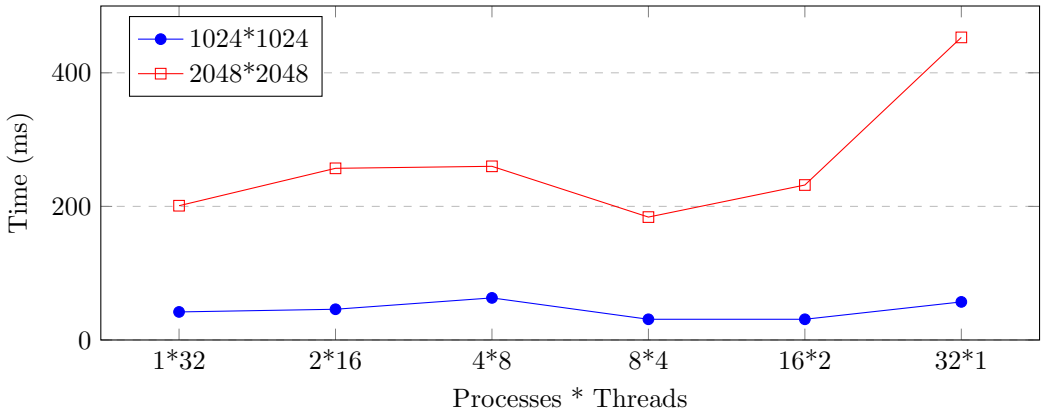

Figure 13: Compassion of Execution Time using processes * threads

Overall, when the number of processes and threads approaches, the performance reaches its best.

1.8.3 Profiling results

This is generated by `perf stat`. So the unit changed to the smallest unit.

Process	average CPU-Cycle	average Cache Miss	average Page Fault	Time (ms)
1	2,848,644,311	389,450	4,977	42
2	1,629,821,383	305,146	5,318	46
4	1,215,671,520	691,119	5,087	63
8	1,071,596,920	1,144,076	4,809	31
16	1,474,778,859	1,844,466	5,315	31
32	1,468,170,706	2,003,002	4,837	57

Table 6: Profiling results of task 4 (1024*1024)

So we can find that average cache miss increase. Things I learnt from it are that perf is not always a good tool, for example for MPI.

1.9 Bonus: CUDA

1.9.1 Solution 1: Global Memory

To make copy more easily, I slightly changed the Matrix implementations to one dimension array.

The kernel function is defined below.

```
__global__ void multiplicateMatrixOnDevice(int *A, int *B, int *C, int M, int K, int N)
{
  int x = threadIdx.x + blockDim.x*blockIdx.x;//row number
  int y = threadIdx.y + blockDim.y*blockIdx.y;//col number
  if (x < N && y < M) {
     int sum = 0;
     for (int k = 0; k < K; ++k) {
        sum += A[y*K + k] * B[k*N + x];
     }
     C[y*N + x] = sum;
   }
}
```

So in the kernel function is aimed to calculated one value in Matrix C.

1.9.2 Solution 2: Shared Memory

Shared memory is much faster than local and global memory.

Shared memory is allocated by thread blocks, so all threads in the block can access the same shared memory. Threads can access data from shared memory loaded from global memory by other threads within the same thread block. As shown in the figure below, each thread block has a shared memory, and the threads in the block obtain data much faster than from global memory. Ref: CUDA Blog

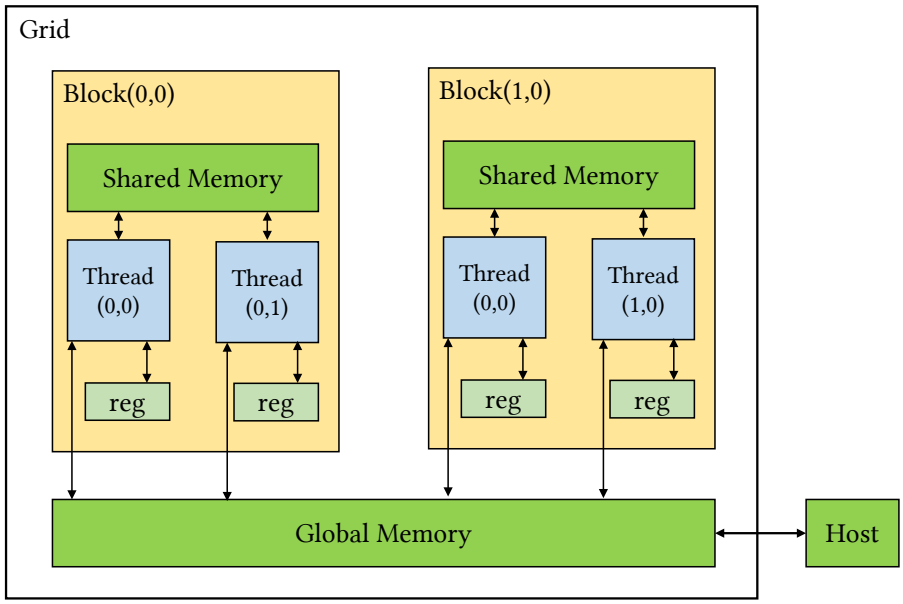

Figure 14: GPU figure

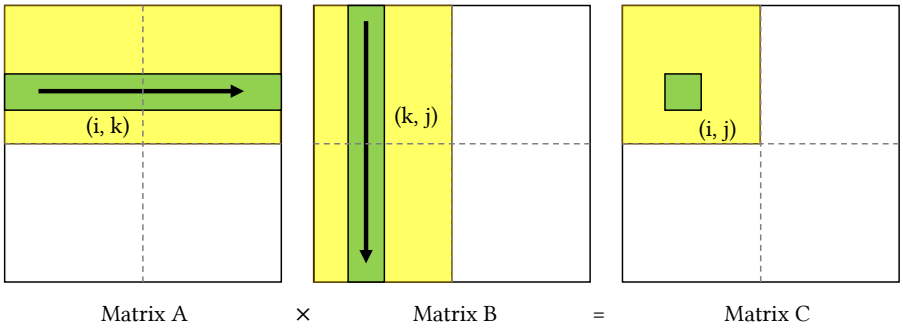

Figure 15: GPU Shared Memory Matrix Multiplication

So we try to managed them in blocks. It is very like the GPU's version of Matrix Tiling.

The kernel function is defined blow using `__syncthreads()` to synchronize threads.

```
__global__ void multiplicateMatrixSharedMemory(int* mA, int* mB, int* mC,
                                                int m, int n, int k)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int res = 0;

    __shared__ int A[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ int B[BLOCK_SIZE][BLOCK_SIZE];

    int nIter = (k + BLOCK_SIZE - 1) / BLOCK_SIZE;
    for(int i = 0; i < nIter; ++i)
    {
        int idxA = row * k + i * BLOCK_SIZE + threadIdx.x;
        int idxB = (i * BLOCK_SIZE + threadIdx.y) * n + col;
        A[threadIdx.y][threadIdx.x] = (idxA < m * k) ? mA[idxA] : 0;
        B[threadIdx.y][threadIdx.x] = (idxB < k * n) ? mB[idxB] : 0;
        __syncthreads();
        for(int j = 0; j < BLOCK_SIZE; ++j)
        {
            res += A[threadIdx.y][j] * B[j][threadIdx.x];
        }
        __syncthreads();
    }
    if(row < m && col < n) mC[row * n + col] = res;
}
```

1.9.3 Solution 3: Merge Threads (up on Solution 2 Shared Memory)

Let's re-think the inner loop in the solution 2 here.

```
for(int j = 0; j < BLOCK_SIZE; ++j)
{
    res += A[threadIdx.y][j] * B[j][threadIdx.x];
}
```

I am wondering if we could use thread merging to disguise a kind of loop unrolling, thus reusing the elements inside, to reduce access to shared memory instead of using it in register files.

The answer is yes. Let's merge two threads first.

```
__global__ void multiplicateMatrixSharedMemoryRegister2(int* mA, int* mB, int* mC,
                                                        int m, int n, int k)
{
    int row = blockIdx.y * blockDim.y * 2 + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    int res[2] = {0,0};

    __shared__ int A[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ int B[BLOCK_SIZE][BLOCK_SIZE];

    int nIter = (k + BLOCK_SIZE - 1) / BLOCK_SIZE;
    for(int i = 0; i < nIter; ++i)
    {
        int idxA = row * k + i * BLOCK_SIZE + threadIdx.x;
        int idxB = (i * BLOCK_SIZE + threadIdx.y) * n + col;
        int a = (idxA < m * k) ? mA[idxA] : 0;
        int b = (idxB < k * n) ? mB[idxB] : 0;
        int a16 = (idxA + 16 * k < m * k) ? mA[idxA + 16 * k] : 0;
        int b16 = (idxB + 16 * n < k * n) ? mB[idxB + 16 * n] : 0;
        A[threadIdx.y][threadIdx.x] = a;
        A[threadIdx.y + 16][threadIdx.x] = a16;
        B[threadIdx.y][threadIdx.x] = b;
        B[threadIdx.y + 16][threadIdx.x] = b16;
        __syncthreads();
        for(int j = 0; j < BLOCK_SIZE; ++j)
        {
            res[0] += A[threadIdx.y][j] * B[j][threadIdx.x];
            res[1] += A[threadIdx.y + 16][j] * B[j][threadIdx.x];
        }
        __syncthreads();
    }
    if(row < m && col < n) mC[row * n + col] = res[0];
    if(row+16 < m && col < n) mC[(row+16) * n + col] = res[1];
}
```

So now, in each thread, we will load A and B twice. But look at the inner loop, we visit the same address of B twice. So we save one visiting operation, although the sum of load operations is same. So we can hide some latency here.

1.9.4 Experimental results

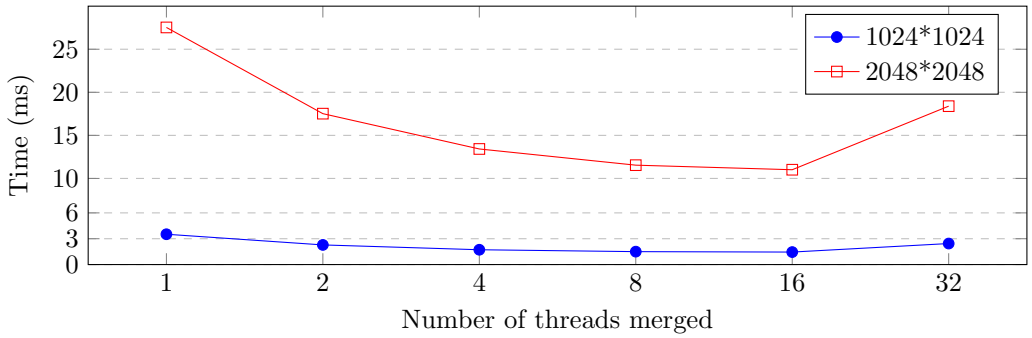

Figure 16: Compassion of Execution Time about the number of merged threads

Number of threads merged	1024*1024 (ms)	2048*2048 (ms)
1	3.52	27.53
2	2.28	17.52
4	1.72	13.42
8	1.50	11.54
16	1.45	11.01
32	2.44	18.39

Table 7: Compassion of Execution Time about the number of merged threads

As the number of threads merged increases, the time decreases and then increases, and the fastest speed is found to be merging 16 threads at 1024*1024 and 2048*2048 on our cluster.

1.9.5 CUDA results

Method	1024*1024 Time (ms)	2048*2048 Time (ms)
CUDA (Global Memory)	4.43	34.73
CUDA (Shared Memory)	3.52	27.53
CUDA (Shared + Merge Threads)	1.45	11.01

Table 8: Compassion of Execution Time

1.9.6 Profiling results

Time(%)	Total Time (ns)	Num Calls	Average	Name
96.1	199,679,832	3	66,559,944.00	cudaMalloc
2.1	4,395,977	1	4,395,977.00	cudaEventSynchronize
1.3	2,717,318	3	905,772.70	cudaMemcpy
0.5	1,000,475	3	333,491.70	cudaFree
0	49,464	1	49,464.00	cudaLaunchKernel
0	23,811	2	11,905.50	cudaEventRecord
0	15,041	2	7,520.50	cudaEventCreate

Table 9: Profiling results of Global Memory (1024*1024)

Time(%)	Total Time (ns)	Num Calls	Average	Name
96.6	205,625,171	3	68,541,723.70	cudaMalloc
1.6	3,483,791	1	3,483,791.00	cudaEventSynchronize
1.3	2,803,713	3	934,571.00	cudaMemcpy
0.4	956,696	3	318,898.70	cudaFree
0	52,502	1	52,502.00	cudaLaunchKernel
0	23,593	2	11,796.50	cudaEventRecord
0	14,627	2	7,313.50	cudaEventCreate
0	6,966	2	3,483.00	cudaEventDestroy

Table 10: Profiling results of Shared Memory (1024*1024)

Time(%)	Total Time (ns)	Num Calls	Average	Name
97.5	204,887,486	3	68,295,828.70	cudaMalloc
1.3	2,734,745	3	911,581.70	cudaMemcpy
0.7	1,445,211	1	1,445,211.00	cudaEventSynchronize
0.5	1,027,578	3	342,526.00	cudaFree
0	51,528	1	51,528.00	cudaLaunchKernel
0	23,992	2	11,996.00	cudaEventRecord
0	14,119	2	7,059.50	cudaEventCreate
0	6,954	2	3,477.00	cudaEventDestroy

Table 11: Profiling results of Shared Memory + Merge 16 Threads (1024*1024)

1.10 Bonus: OpenACC

Use the same solution in CUDA Global Memory. Calculate one value for one thread.

1.10.1 Profiling results

Time(%)	Total Time (ns)	Num Calls	Average	Name
96.1	199,679,832	3	66,559,944.00	cudaMalloc
2.1	4,395,977	1	4,395,977.00	cudaEventSynchronize
1.3	2,717,318	3	905,772.70	cudaMemcpy
0.5	1,000,475	3	333,491.70	cudaFree
0	49,464	1	49,464.00	cudaLaunchKernel
0	23,811	2	11,905.50	cudaEventRecord
0	15,041	2	7,520.50	cudaEventCreate

Table 12: Profiling results of OpenACC (1024*1024)