1 Introduction & Background

1.1 Introduction

1.1.1 Matrix Multiplication

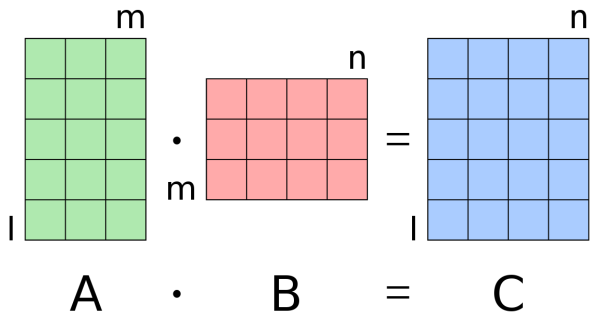

Figure 1: Matrix Multiplication

In this assignment, we'll implement Matrix Multiplication also known as gemm(general matrix multiply) which can be performed in multi-threading. Matrix multiplication is not only excellent exercise in parallel programming, but it also plays a crucial role in reality applications. Many scientific researches perform matrix multiplication, and the rapidly emerging field of AI also excessively performs multiplication on massively large matrices. Many researches have been conducted to optimize the gemm, and there is lots of library supporting optimized gemm such as BLAS, BLIS, etc. However, since parallel gemm strongly depends on the hardware characteristics such as cache configuration, in this assignment, we'll gonna implement our own gemm optimized specially for our hardware.

1.1.2 Spec

AMD EPYC 7452 32-Core Processor			
Core(s) per socket	32		
L1i cache	32 x 32KiB	8-way set associative	
L1d cache	32 x 32KiB	8-way set associative	Write-Back
L2 cache	32 x 512KiB	8-way set associative	Write-Back
L3 cache	8 x 16MiB		
Memory block size	128M		
Total online memory	512G		

Table 1: Hardware Configuration

The workload specification for this assignment is as follows: perform matrix multiplication for matrices A and B, with dimensions n x m and m x n respectively, to compute a

matrix C of size nxn. The size of the matrices is restricted to be within the range of:

$$256 <= n, m <= 2048$$

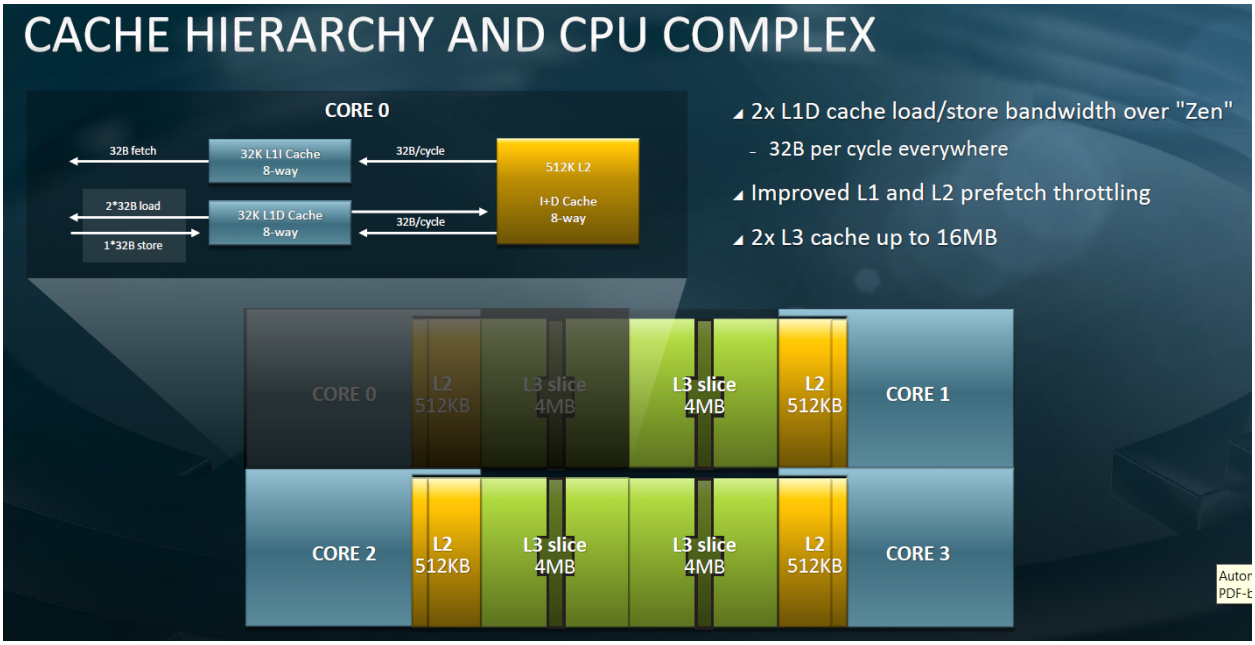

Figure 2: EPYC 7452 - Cache Hierarchy

To optimize specially for our hardware environment, let's briefly take a look at hardware configuration. Table 1. shows hardware configuration of server environment. CPU model name is AMD EPYC 7452 32-Core Processor. We can see that there is 32 physical cores per socket and 1MB L1 I cache, D cache, 32 MB L2 cache, 512 MB L3 cache and 512GB of main memory. L1 is dedicated for each core as a local cache whose size is 32KB each with 8-way set associativity. L2 cache is also dedicated for each core and its size is 512KB each. L3 cache is shared among the cores, which is dedicated for each CPU Complex(CCX). Figure 2. represents the cache hierarchy of each CPU Complex(CCX). CPU Complex(CCX) is basic building blocks of AMD CPU, which consists of several physical cores and shared resources. Each CCX of EPYC 7542 consists of 4 physical cores, L2 cache and L3 cache. Later on, we'll optimize our algorithm to maximize the utilization of these cache composition.

1.2 Background

1.2.1 Memory Access

As mentioned at assignment1, [HW1: 1D Parallel Filtering], one of the most important thing in parallel computing is memory access. Since most of parallel computing application is actually memory bounded, it is important to optimize the memory access. Let's start with how memory access actually occurs in real system.

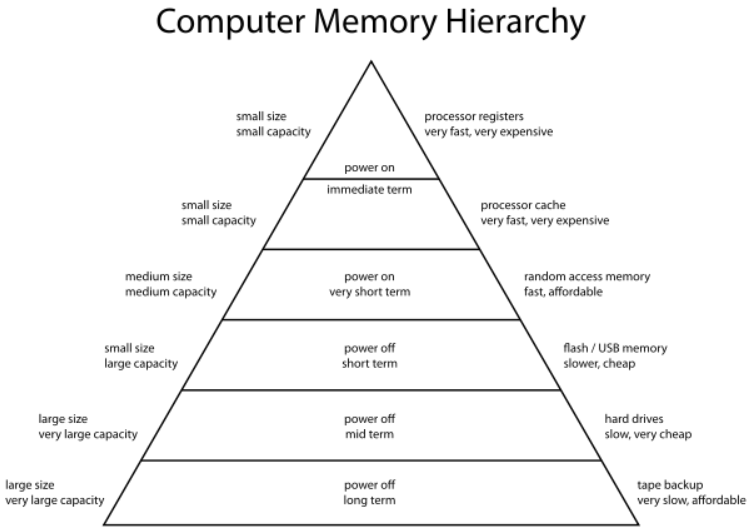

Figure 3: Memory Hierarchy

In the Von Neumann architecture, there are two essential components: the processor and the memory. While the speed of processor gets faster rapidly, speed of memory remains relatively slow. This was a conventional bottleneck of Von Neumann architecture. To address this bottleneck, the concept of a cache was introduced. By positioning a fast, small cache close to the processor and leveraging the locality of data access, it effectively reduces the memory access latency. Cache closer to the processor is faster but small and in this perspective register can be treated as L0 cache. Conversely, further cache is bigger but slower and main memory can be treated as the last level cache.

When processor requests memory I/O, it first accesses to the lowest level of cache(e.g. L1 cache). If there is data that matches to the request, it performs the request(read/write). This is called **cache hit**. On the other hand, if there is no data that matches to the request, it accesses the higher level of cache and so on. This is called **cache miss**. Since lower level of cache is faster, if cache hit occurs for most of the memory access, system can reduce the memory access latency effectively, and might be amortized the overhead of cache hierarchy which potentially leads to performance enhancement.

Figure 4. shows how data is actually transferred between processor, cache and main memory. When data is transferred between different level of cache (or main memory), data

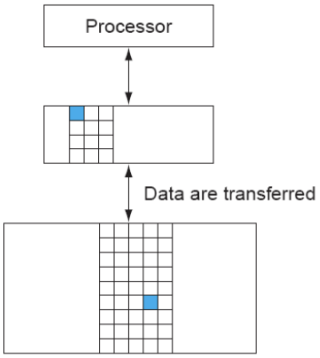

Figure 4: Data Transfer between processor - cache - main memory

adjacent to the requested one is also transferred. This unit of memory accessing is called **cache block** or **cache line**. Therefore, it might be note worthy to align the data along to cache block size when distributing the workload to multiple threads can actually improve the performance. This alignment might improve the cache utilization and also lower the false sharing problem.

Dealing with the memory access latency, **cache miss** must be primarily considered. There is 3 type of cache miss: **Cold miss**, **Capacity miss** and **Conflict miss**. **Cold miss**, also called compulsory miss refers to the cache miss which occurs at the every first access of certain cache block which is inevitable. **Capacity miss** refers to the cache miss due to the size of cache. If data is larger than the cache size, processor needs to access beyond the cache size which occurs eviction of cache block. Therefore, it is very important to adjust the data to fit the cache size. **Conflict miss** refers to the cache miss due to the overlapped cache block address. It can be understood exactly the same to conflict in hash table. Since it leads to cache under utilization where eviction occurs even there is sufficient space, it must be mitigated by reorganizing the data or changing the memory access pattern.

Now, let's look at memory access in matrix multiplication. Assume cache block size is C and multiplying N x N square matrix A and B and each matrix is allocated to memory in row-major order. Matrix A will be accessed in row-major, therefore we can fully utilize the both spatial and temporal locality of data. $\frac{N}{C}$ cache miss will occur for a single row, and there is N rows, $\frac{N^2}{C}$ cache miss will occur. On the other hand, matrix B is accessed in column-major, $\frac{N}{C}$ miss occur for computing a single element of matrix C. Therefore, $\frac{N^3}{C}$ cache miss occurs for matrix B. Also, in the case matrix B doesn't fit into the cache, it might suffer from excessive cache miss due to eviction.

1.2.2 Transpose

Accessing in column major order is problematic. Especially, when column size N is multiple of 2, due to conflict miss, cache utilization lower with factor of its associativity. Consider situation where cache blocks of same column is mapped to same set. Assume the cache is

A-way set associative, first A access will bring the cache block without eviction. However after that, eviction occurs for every C access and it leads to conflict miss where C represents the size of cache block.

To mitigate this, techniques such as padding or transpose can be done. Padding focuses on changing the data organization. By padding, we can adjust the cache blocks from the same column do not mapped to same set. On the other hand, transpose actually focuses on changing the memory access pattern. By transposing the matrix, processor accesses B^T in row major order which mitigates the excessive cache miss due to conflicts. Also it helps improving the cache utilization in the case matrix size is bigger than cache size.

1.2.3 Blocked Matrix Multiplication

Blocked matrix multiplication, also known as tiling is some kind of divide-and-conquer technique. When matrix size is larger than cache size, miss frequency increases due to eviction(conflict miss). To achieve high performance on large matrices, blocked matrix multiplication techniques divdes entire matrix into small sub-matrices and perform matrix multiplication on them.

Since it performs computation repeatedly on sub-matrix, it can fully utilize the temporal locality of the data. Direct approach occurs $\frac{N^2}{C}$ misses on A and $\frac{N^3}{C}$ misses on B, where C represents for the size of cache block. Using blocked matrix multiplication, it reduces to $\frac{N^3}{B \times C}$ both for matrix A and B where B represents for size of block. It's note worthy that cache miss is inversely proportional to the size of block B. Therefore, we can induce that maximizing the block B while maintaining each block fit into the L1 cache is desired.

Blocked matrix multiplication is particularly effective when a single row doesn't fit to L1 cache. If a single row fit into the L1 cache, matrix A occurs $\frac{N^2}{C}$ misses. On the other hand, if it doesn't fit into L1 cache, matrix A occurs $\frac{N^3}{C}$ misses due to eviction. Therefore, by dividing the entire matrix into small, well-fit sub-matrices we can maintain the $\frac{N^3}{B \times C}$ read misses even for the large size of matrix by utilizing the temporal locality.

However, we also must be mindful of the overhead of this technique. First of all, it leads to increased level of for loop. Second, additional computation for remaining elements might be required in the case of matrix size doesn't divided perfectly with block size. We'll analyze about this later.

Also, considering about the layout of sub-matrix can be a good point. We'll start with $B \times B$ sub-matrix first, and apply other layout such as $B_1 \times B_2$ sub-matrix to optimize depending on the input matrices.

2 Implementation

2.1 Direct Approach

```
omp_set_num_threads(NT);
#pragma omp parallel for firstprivate(local_sum)
for (int i = 0; i < n; i++){
  for (int j = 0; j < n; j++){
    for (int k = 0; k < m; k++)
      local_sum += matrixA[i * m + k] * matrixB[k * n + j];
    matrixC[i*n+j] += local_sum;
    local_sum=0;
  }
}
```

Above code represents the most simple way to implement matrix multiplication. It is just the same with the given reference matrix multiplication code except two points. First, parallel for clause in OpenMP directive is used to parallelize for loop. Also, private variable local_sum is used to minimize the memory access. To make local_sum private for each thread, firstprivate clause is used.

2.2 Transpose

```
/******************Transpose*****************/
if(1){
  #pragma omp parallel for num_threads(NT_T) schedule(auto) collapse(2)
  for(int i=0; i<m; i++){
    for(int j=0; j<n; j++)
      matrixB_T[j*m+i] = matrixB[i*n+j];
  }
}

omp_set_num_threads(NT);
/*************Matrix Multiplication**********/
#pragma omp prallel for collapse(2) schedule(auto) firstprivate(
 local_sum)
for(int i=0; i<n; i++){
  for(int j=0; j<n; j++){
    for(int k=0; k<m; k++)
      local_sum += matrixA[i*m+k] * matrixB_T[j*m+k];
    matrixC[i*n+j] = local_sum;
    local_sum = 0;
  }
}
```

Above code represents matrix multiplication with transposing matrix B. Both transposition and matrix multiplication are executed in parallel using OpenMP directive.

2.3 Blocked Matrix Multiplication

```
switch(CASE){

  case 0:
    #pragma omp parallel for schedule(auto) collapse(2) firstprivate(local_sum)
    for(int i=0; i<n; i+=B){
      for(int j=0; j<n; j+=B){
        for(int k=0; k<m; k+=B){

          for(int ii=0; ii<B; ii++){
            for(int jj=0; jj<B; jj++){
              for(int kk=0; kk<B; kk++){
                  local_sum += matrixA[(i+ii)*m + (k+kk)] * matrixB[(j+jj) + (k+kk)*n];
              }
              matrixC[(i+ii)*n + (j+jj)] += local_sum;
              local_sum=0;
            }
          }

        }
      }
    }
    break;

  case 1:
  // For NB x NB submatrix - main
    #pragma omp parallel for schedule(auto) collapse(2) firstprivate(local_sum)
    for(int i=0; i<(n/B)*B; i+=B){
      for(int j=0; j<(n/B)*B; j+=B){
        for(int k=0; k<(m/B)*B; k+=B){

          for(int ii=0; ii<B; ii++){
            for(int jj=0; jj<B; jj++){
              for(int kk=0; kk<B; kk++){
                  local_sum += matrixA[(i+ii)*m + (k+kk)] * matrixB[(j+jj) + (k+kk)*n];
              }
              matrixC[(i+ii)*n + (j+jj)] += local_sum;
              local_sum=0;
            }
          }

        }
      }
    }

    // For NB x NB submatrix - remain
    local_sum = 0;
    #pragma omp parallel for schedule(auto) collapse(2) firstprivate(local_sum)
```

```
48      for(int i=0; i<(n/B)*B; i++){
49        for(int j=0; j<(n/B)*B; j++){
50          for(int k=(m/B)*B; k<m; k++)
51            local_sum += matrixA[i*m+k] * matrixB[j+k*n];
52          matrixC[i*n+j] += local_sum;
53          local_sum = 0;
54        }
55      }
56
57      // For NB x r submatrix
58      local_sum = 0;
59      #pragma omp parallel for schedule(auto) collapse(2) firstprivate(
    local_sum)
60      for(int i=0; i<(n/B)*B; i++){
61        for(int j=(n/B)*B; j<n; j++){
62          for(int k=0; k<m; k++)
63            local_sum += matrixA[i*m+k] * matrixB[j+k*n];
64          matrixC[i*n+j] += local_sum;
65          local_sum = 0;
66        }
67      }
68
69      // For r x NB submatrix
70      local_sum = 0;
71      #pragma omp parallel for schedule(auto) collapse(2) firstprivate(
    local_sum)
72      for(int i=(n/B)*B; i<n; i++){
73        for(int j=0; j<(n/B)*B; j++){
74          for(int k=0; k<m; k++)
75            local_sum += matrixA[i*m+k] * matrixB[j+k*n];
76          matrixC[i*n+j] += local_sum;
77          local_sum = 0;
78        }
79      }
80
81      // For r x r submatrix
82      local_sum = 0;
83      #pragma omp parallel for schedule(auto) collapse(2) firstprivate(
    local_sum)
84      for(int i=(n/B)*B; i<n; i++){
85        for(int j=(n/B)*B; j<n; j++){
86          for(int k=0; k<m; k++)
87            local_sum += matrixA[i*m+k] * matrixB[j+k*n];
88          matrixC[i*n+j] += local_sum;
89          local_sum = 0;
90        }
91      }
92
93      break;
94
95     default:
96      break;
97   }
```

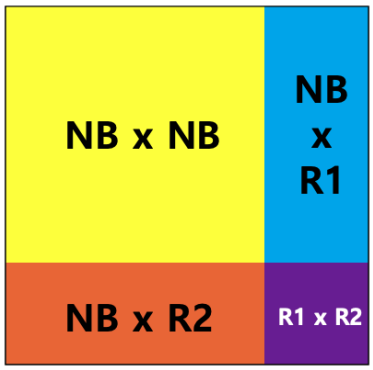

Figure 5: Blocked Matrix Multiplication

Above code shows implementation of blocked matrix multiplication(or tiling). There exists two cases, one for the case where input matrices divided into block and the other case where they don't. Latter case needs additional computation for remaining elements. First for loop computes on NB x NB sub-matrix of matrix C using blocked matrix multiplication techniques. Second for loop also computes on NB x NB sub-matrix of matrix C, but for calculate the elements that doesn't included in prior computation. Third for loop computes on NB x R1 sub-matrix of matrix C. Fourth loop computes on R2 x NB sub-matrix of matrix C. Last loop computes on R1 x R2 sub-matrix of matrix C. As we can see, in the case where input matrices doesn't divided by block, additional computation required to get the correct results. Constructing additional loop might be a significant overhead especially for matrices with small size. Therefore, we need to consider the effect of size and appropriately handle this.

Regarding to which loops should be parallelized, I referred to **Smith, John, and Alice Doe. 2023. "Anatomy of High-Performance Many-Threaded Matrix Multiplication." Journal of Computational Mathematics 29 (2): 101-120**. Briefly introducing this paper, its about BLIS framework extends "GotoBLAS approach" for implementing matrix multiplication. It defines inner product(essentially the matrix multiplication for each single element) as micro kernel. There is 5 for loop around the micro kernel. This paper mainly deals with selecting which for loop should be parallelized to get the optimum performance. In conclusion, since it doesn't provide enough parallelism, and also it shares the data between threads, parallelizing the most three inner loops isn't appropriate. Therefore I parallelized two outermost loops using collapse clouse of OpenMP directives.

2.4 All Together

I implemented matrix multiplication using both transposition and tiling. Since I already explained the implementation of both techniques, I will omit the detailed explanation of this.

3 Evaluation & Optimization

3.1 Evaluation

Technique	Matrix Size	Run Time [sec]
Direct Approach	1024, 1024	0.527784
	2048, 2048	-
Transpose	1024, 1024	0.107273
	2048, 2048	2.49538
Tiling	1024, 1024	0.138311
	2048, 2048	1.85971
All Together	1024, 1024	0.0389178
	2048, 2048	0.20426

Table 2: Evaluation on GEMM

I've done evaluation for each matrix multiplication introduced at [2. Implementation] part. I set 16 threads in the case of size=1024, and 32 threads in the case of size=2048. For Tiling, I set size of blocked matrix B=32. We can observe that applying all together, we can achieve more than 12 times speed up comparing to direct approach. The parameter set in this evaluation is done arbitrarily. It might be determined theoretically and experimentally on [3.2 Optimization] part.

3.2 Optimization

3.2.1 Block Size

Let's start with optimizing the size of sub-matrix. As mentioned earlier, since large B leads to more speed up, we need to choose maximum B while maintaining two sub-matrices of matrix A and B fit into L1 cache. L1 D cache for each core is 32KB, therefore we can simply calculate the optimum $B = \sqrt{\frac{2^{15}}{2^2 \times 2}} = 2^6$.

Block Size (B)	Matrix Size	Run Time [sec]
B=32	1024, 1024	0.0389178
	2048, 2048	0.20426
B=64	1024, 1024	0.0330113
	2048, 2048	0.0919427
B=128	1024, 1024	0.0330148
	2048, 2048	0.237377

Table 3: Evaluation on GEMM w.r.t Block Size **B**

Above table represents the runtime of GEMM with respecting to block size **B**. Therefore we can induce that the optimum block size **B=64**.

3.2.2 Number of Threads - NT

The next parameter is number of threads. As mentioned at assignment 1, large number of threads doesn't always lead to performance improvement. Since creating and distributing the workload for multiple threads have significant overheads especially workload or parallelism isn't sufficient to amortize it, we should adjust the number of thread properly depends on the workloads.

# of Threads (NT)	Matrix Size	Run Time [sec]
NT=8	1024, 1024	0.0469192
	2048, 2048	0.196808
NT=16	1024, 1024	0.0331326
	2048, 2048	0.129705
NT=32	1024, 1024	0.0389415
	2048, 2048	0.0852724

Table 4: Evaluation on GEMM w.r.t NT

We can observe that for 1024 x 1024 matrices, NT=16 is appropriate and for 2048 x 2048 matrices, NT = 32 is appropriate. Since size of input matrices can vary, I conducted additional experiment with various input matrices. In conclusion, I got the following results.

Matrix Size	Run Time [msec]					
	NT=4	NT=8	NT=16	NT=24	NT=32	NT=36
256, 256	2.52885	2.86874	2.55238	3.10523	3.29498	4.448818
512, 512	13.1073	9.40423	7.51635	7.80597	8.00512	7.83803
768, 768	24.0846	23.6799	17.1232	14.6426	18.22	17.0669
1024, 1024	84.4475	39.9654	33.5514	32.2738	29.2203	27.748
1536, 1536	126.876	75.5891	57.7635	57.98	52.6268	46.1095
2048, 2048	289.211	160.722	114.011	101.759	104.664	77.3954

Table 5: Evaluation on GEMM w.r.t NT

Based on this results, I determined the NT depending on input matrices as follows.

Matrix Size	[256, 512)	[512, 1024)	[1024, 2048)
NT	8	24	36

Table 6: NT depending on input matrices

3.2.3 Number of Threads - NT_T

Matrix Size	Run Time [µsec]					
	NT=4	NT=8	NT=16	NT=24	NT=32	NT=36
256	1.67871	2.17986	2.85071	2.91372	15.2056	3.20369
512	6.36122	5.27901	7.23843	5.39373	11.9085	10.1852
768	15.7138	16.4575	15.7371	23.5842	26.704	18.556
1024	31.802	32.6474	28.9627	16.4664	42.9884	19.8939
1536	44.3026	47.7124	49.4893	52.2393	66.262	40.8807
2048	78.6685	140.888	72.4241	53.5816	234.383	88.5893

Table 7: Evaluation on GEMM w.r.t NT_T

Now let's determine the threads for transposing the matrix. Fixing the number of threads determined above, changing the number of threads used for transposition NT_T, I evaluated the performance. Based on this result, I determined NT_T as follows.

Matrix Size	[256, 512)	[512, 1024)	[1024, 2048)
NT_T	4	14	24

Table 8: NT_T depending on input matrices

3.2.4 Number of Threads - NT_R

Now let's determine threads count for computing remained elements. Since computing remaining loops provide less parallelism compared to main computation, we can induce that less thread counts would be optimal for this. However, since it heavily depends on the input matrices, I didn't take additional experiment for this. I think it can be determined by analyzing the number of iteration for each part depending to the dimension of input matrices.

3.3 Final Implementation & Evaluation

```
1  /***************Local Variable***************/
2  int* matrixB_T  =new int[n*m];
3  int local_sum   =0;
4  int CASE        =0;
5  int B           =64;
6  int S           = (n>>6)*(n>>6)*(m>>6);
7  int NT, NT_T, NT_R, NT_r;
8  /*****************Set option*****************/
9  if(S < 4*4*4){
10   NT   = 4;
11   NT_T = 4;
12   NT_R = 4;
13   NT_r = 4;
14 }
15 else if(S < 8*8*8){
```

```
    NT    =8;
    NT_T  =4;
    NT_R  =4;
    NT_r  =4;
  }
  else if(S < 16*16*16){
    NT    =24;
    NT_T  =14;
    NT_R  =14;
    NT_r  =14;
  }
  else if(S <= 32*32*32){
    NT    =36;
    NT_T  =24;
    NT_R  =24;
    NT_r  =24;
  }
```

The above code represents dealing with various input matrices. The parameters are determined by the value $S \propto n^2 \times m$. Because the number of iteration, $N \propto n^2 \times m$ of main loop , thread counts are determined depending to size related value S.

```
      // For r x NB submatrix
      local_sum = 0;
      #pragma omp parallel for num_threads(NT_R) schedule(auto) collapse
   (2) firstprivate(local_sum)
      for(int i=(n/B)*B; i<n; i++){
        for(int j=0; j<n; j++){
          for(int k=0; k<m; k++)
            local_sum += matrixA[i*m+k] * matrixB_T[j*m+k];
          matrixC[i*n+j] += local_sum;
          local_sum = 0;
        }
      }
```

The body of code is just the same implemented above. However I merge the code of computing r x NB sub-matrix and r x r sub-matrix.

Matrix Size	Run Time[µsec]	Matrix Size	Run Time[µsec]	Matrix Size	Run Time[µsec]
442, 1482	9.76126	1806, 951	23.3581	1333, 412	13.407
1466, 767	24.292	1843, 1853	56.8365	1024, 1024	19.8667
1847, 872	33.8715	1441, 1112	23.3056	2048, 2048	57.1385
1312, 767	19.8384	993, 668	10.6116	2048, 256	14.4473
1844, 1980	55.1454	1492, 1112	22.2578	256, 2048	5.94807
1940, 1016	33.9308	601, 834	5.9508		
1144, 1512	37.5192	1172, 712	13.1556		
1304, 939	23.3865	1199, 1892	22.5327		
707, 1097	8.48876	838, 1595	20.0388		
1152, 1500	23.6083	1682, 1250	24.2117		

Table 9: Evaluation on Final Implementation

4 Discussion

4.1 Multi-Level Matrix Multiplication

In our assignment, we've utilized blocked matrix multiplication technique, where divides the input matrix into sub-matrices and perform multiplication on those sub-matrices. It enhances the temporal locality by fitting the sub-matrices into the L1 cache.

As mentioned earlier, our hardware has multiple level of cache, L2 cache and L3 cache. To further exploit the cache utilization including L2 and L3 cache, we can adopt multi-level matrix multiplication. Initially, it divides the input matrices into large sub-matrices with size B_1. Then the large sub-matrices can be divide into sub-matrices with size B_2. Finally, these sub-matrices with size B_2 can be divided into smallest sub-matrices with size B_3. Each sub-matrices should fit into corresponding level of cache to fully utilize the locality. For example, sub-matrices with size B_2 must fit into L2 cache.

Decomposing matrix into smaller sub-matrices, and positioning them in different level of caches enables to utilize the temporal locality and reduce the cache miss frequency, potentially improve the performance. However, as mentioned earlier, it introduce significant overhead since it needs additional loop. In our assignment, size of input matrices range 256 to 2048. It doesn't fully amortize the overheads, leading to performance degradation. However it's noteworthy that multi-level multiplication might work and highly effective in the case of multiplication on massively large matrices.

5 References

[1] Cache Hierarchy: https://www.anandtech.com/show/14694/amd-rome-epyc-2nd-gen/7
[2] HW configuration: https://en.wikichip.org/wiki/amd/epyc/7542
[3] BLIS: Smith, John, and Alice Doe. 2023. "Anatomy of High-Performance Many-Threaded Matrix Multiplication." Journal of Computational Mathematics 29 (2): 101-120