Optimizing Benchmark Performance: Leveraging Architectural Details and Insights To Improve Benchmark Performance

By\ Md Siddiqur Rahman Tanveer
g202417180
COE501 Term 242

1. Introduction

One of the architectural insights that has been considered for this assignment is lower memory latency by better cache utilization. While caches are ubiquitous, different architectures have different cache organization. Ergo, leveraging cache hierarchies is not as universal as aforementioned optimizations. To increase the cache utilization, an implementation has to take into account the usage pattern of the input as well as the outputs. Moreover, the implementation has to account for cache limitation (i.e., capacity, cacheline size, and associativity). The cache locality is set according to the cache parameters; for example, spatial locality in a cache is limited by the cacheline size.

In this assignment, we would like to implement a new opt implementation for the mmult benchmark and add it to the provided educational benchmark suite YABMS [1]. The new opt implementation will leverage blocking to increase cache utilization regardless of the size of the input matrices. Then, we will compare the improvement of performance between theopt implementation and naive implementation.

2. Collaboration

In this assignment at I have used the environment in WSL (Windows Subsystem for Linux) [2] as I learned how to add the new benchmark to the existing YABMS and build and run in previous assignment. I got idea clearly about the block size regarding the performance measurement from Nafisa Tabassum and Sakibul Islam.

3. Naive Implementation

The naïve implementation of the matrix multiplication has been performed according to the Algorithm shown in the manual.

Algorithm 1 Naïve `mmult` Implementation

```
1:  procedure MMULT(A, B)
2:      R ← empty_matrix(M, P)
3:      for i = 1 → M do
4:          for j = 1 → P do
5:              R[i][j] ← 0
6:              for k = 1 → N do
7:                  R[i][j] += A[i][k] * B[k][j]
8:              end for
9:          end for
10:     end for
11:     return R
12: end procedure
```

The code implementation is shown below

```
// Perform matrix multiplication: C = A * B
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < p; j++) {
            for (int k = 0; k < n; k++) {
                matR[i][j] += matA[i][k] * matB[k][j];

            }
        }
    }
```

I built the benchmark and then generated the golden reference which should be IEEE754 [3] floating-point 32-bit single according to the dataset information mentioned in the reading manual as per below.

Table 1: **Datasets Information**–table showing dimensions for input datasets and the golden reference.

Dataset	Matrix A		Matrix B		Matrix R	
	M	N	M	N	M	N
testing	16	12	12	8	16	8
small	121	180	180	115	121	115
medium	550	620	620	480	550	480
large	962	1,012	1,012	1,221	962	1,221
native	2,500	3,000	3,000	2,100	2,500	2,100

I generated the matrices A and B data and stored in text files (*matrix_A_float.txt* and *matrix_B_float.txt*) and multiplication result stored *matrix_C_float.txt* file in row-major-matrix [4] format.

4. Optimized Implementation:

To improve cache utilization, one can change the access patterns for how the output is calculated. To address the issue with low spatial locality in the accessed column of the *B* matrix, we can lower the number of cachelines accessed by enhancing temporal locality. Temporal locality can be improved by changing the access pattern of the implementation. An implementation can improve temporal locality by evaluating elements from different rows in the output rather rather than one complete row of the output. Since the temporal locality of the *B* matrix is improved, the spatial locality of the *B* matrix can be exploited easily. As a result, by exploiting the spatial locality of the *B* matrix, once can improve the temporal locality of the *A* matrix. Even with changes of the compute pattern of the output matrix, the number of access cachelines would overwhelm any reasonable cache configuration and cause a capacity miss. As a result, instead of computing each element to completion, we limit the calculation of each element to a subset of the required computations.

Effectively, we perform partial computation for each output elements utilizing the overlapped elements from matrices A and B. By doing so, we can maximize the temporal and spatial locality for input elements from matrices A and B. By performing partial computations of the output using a sub-set of the input elements, we effectively "block" the input and output into sub-matrices; then, the sub-matrices are multiplied to generate a partial summation of the corresponding output elements.

The optimized Algorithm for block wise matrix multiplication is shown in figure

Algorithm 2 Optimized `mmult` Implementation

```
1: procedure MMULT_OPT(A, B, b)
2:     for i = 1 → M, step b do
3:         for j = 1 → P, step b do
4:             R[i][j] ← 0
5:         end for
6:     end for
7:     for ii = 1 → M, step b do
8:         for jj = 1 → P, step b do                    # For each output sub-matrix
9:             for kk = 1 → N, step b do                # For each input sub-matrix
10:                for i = ii → min(ii + b, M) do       # Perform sub-matrix multiplication
11:                    for j = jj → min(jj + b, P) do
12:                        val ← R[i][j]
13:                        for k = kk → min(kk + b, N) do
14:                            val+ = A[i][k] * B[k][j]
15:                        end for
16:                        R[i][j] ← val
17:                    end for
18:                end for
19:            end for
20:        end for
21:    end for
22:    return R
23: end procedure
```

I have used the opt.c file for the implementation. The corresponding code for this algorithm in this file is shown below

```
    // Initialize result matrix R
    for (int i = 0; i < m; i += b) {
        for (int j = 0; j < p; j += b) {
            for (int k = i; k < i + b && k < m; ++k) {
                for (int l = j; l < j + b && l < m; ++l) {
                    matR[k][l] = 0.0;
                }
            }
        }
    }

//Do the matrix multiplication here */
    // Perform blocked matrix multiplication
    for (int ii = 0; ii < m; ii += b) {
        for (int jj = 0; jj < p; jj += b) {
            for (int kk = 0; kk < n; kk += b) {
                for (int i = ii; i < ii + b && i < m; ++i) {
                    for (int j = jj; j < jj + b && j < p; ++j) {
                        double val = matR[i][j];
                        for (int k = kk; k < kk + b && k < n; ++k) {
                            val += matA[i][k] * matB[k][j];
                        }
                        matR[i][j] = val;
                    }
                }
            }
        }
    }
```

5. Evaluation

I ran the benchmark both *mmultfloat* and *mmultopt* for *num_runs* = 10. I varied the block size as 8 16, 32 and 64 to get the performance of the benchmark. The following tables represent the outcomes from the run. The simulation results have shown below

Table 1: Simulation result for testing dataset

Naïve		Optimized	
Starting statistics run number	1	Starting statistics run number	1
Standard deviation	1742	Standard deviation	779
Average	5005	Average	3627
Number of active elements	10	Number of active elements	10
Number of masked-off	0	Number of masked-off	0
Runtimes (MATCHING)	5005 ns	Runtimes (MATCHING)	3627 ns

Table 2: Simulation result for small dataset

Naïve		Block 8	Block 16	Block 32	Block 64
Starting statistics run number	1	1	1	1	1
Standard deviation	134130	455994	1250597	547788	1232452
Average	3746389	1922211	2503391	1668116	2442041
Number of active elements	10	10	10	10	10
Number of masked-off	0	0	0	0	0
Runtimes (MATCHING) ns	3746389	1922211	2503391	1668116	2442041

Table 3: Simulation result for medium dataset

Naïve		Block 8	Block 16	Block 32	Block 64
Starting statistics run number	1	1	1	1	1
Standard deviation	14331487	6265249	3743581	6915015	5939931
Average	332744622	110088200	108184831	94959162	95240343
Number of active elements	10	10	10	10	10
Number of masked-off	0	0	0	0	0
Runtimes (MATCHING) ns	332744622	110088200	108184831	94959162	95240343

Table 4: Simulation result for large dataset

Naïve		Block 8	Block 16	Block 32	Block 64
Starting statistics run number	-	1	1	1	1
Standard deviation	-	36871179	20189002	58592701	51186574
Average	-	972930059	886746120	833512052	739734752
Number of active elements	-	10	10	10	10
Number of masked-off	-	0	0	0	0
Runtimes (MATCHING) ns	-	972930059	886746120	833512052	739734752

Table 5: Simulation result for native dataset

Naïve		Block 8	Block 16	Block 32	Block 64
Starting statistics run number	-	2	1	1	1
Standard deviation	-	50789073	387872356	1050089235	370943186
Average	-	10010747557	11205131365	10499767472	6921921333
Number of active elements	-	2	10	10	10
Number of masked-off	-	0	0	0	0
Runtimes (MATCHING) ns	-	10010747557	11205131365	10499767472	6921921333

For the large and naïve dataset the *mmultfloat* benchmark showed error but for *mmultopt* it worked properly since the optimized benchmark has utilized the cache properly.

5. Conclusion

After implementing the benchmark *mmultfloat* and *mmultopt* to perform the matrix multiplication for floating point number respectively and matching golden references. I have gather much idea about optimization in benchmarking. I hope that this experience will help us to work on computer architecture in future research.

6. References:

[1] Khalid Al-Hawaj. *YABMS: Yet Another Benchmark Suites*. https://github.com/hawajkm/YABMS

[2] WSL (Windows Subsystem for Linux). https://ubuntu.com/desktop/wsl

[3] IEEE754 floating point number. https://en.wikipedia.org/wiki/IEEE_754

[4] Row-and-column-major order https://en.wikipedia.org/wiki/Row-_and_column-major_order