Cache Utilization - Matrix Multiplication Assignment #2, CSC 746, Fall 2021

Arulselvan Madhavan[*]
San Francisco State University

Abstract

This paper presents the results of studying the impact of cache utilization, and thereby computational intensity(CI), on a well understood computational problem - matrix multiplication. Matrix multiplication was implemented in three different configurations - a basic implementation, an implementation that used copy optimization, and an implementation with blocked/tiling optimization. The implementation that performed most of its operations on the data in the cache(with high CI) - the blocked/tiling version performed the best. A reference implementation from CBLAS was used to validate the correctness of the implementation.

1 Introduction

Cache utilization is the highest when the data that is brought to the cache is utilized as much as possible to calculate the results, thereby reducing the number of calls that make it to the DRAM. Matrix multiplication(MM) has O(n^3) CPU operations and O(n^3) memory accesses in its basic form. The approach taken in this experiment attempts to do as many operations as it can using the data in the cache line by pre-fetching the data to the cache. Two optimizations were implemented - copy optimization(MMCO) and blocking/tiling optimization, to improve cache utilization(and thereby, the computational intensity).

BMMCO had higher computational intensity and performed the best for block size 16. The copy optimization(MMCO) and tiling optimization(BMMCO) performed fairly close, with respect to runtime, with the tiling optimization slightly edging the copy optimization in larger problem sizes. The results are presented §3

2 Implementation

Matrix Multiplication(MM) was implemented in three different ways - Basic/Naive implementation, Basic implementation with copy optimization(MMCO), Blocked Matrix multiplication with tiling optimization(BMMCO). The correctness of the results of the matrix multiplication were verified with CBLAS.

2.1 Basic/Naive Matrix Multiplication (MM)

Basic Matrix multiplication implementation. C = C + A * B, where C,A,B are matrices of size n*n. The arrays were stored in column major order.

2.2 Matrix Multiplication with copy optimization (MMCO)

Matrix Multiplication with copy optimization. For every element in the result matrix, the row and column that is needed to compute the result is brought to the cache ahead of time by using a simple copy operation

[*]email:amadhavan@sfsu.edu

```
for (int row_id = 0; row_id < n; row_id++) {
  for (int col_id = 0; col_id < n; col_id++) {
    int out_idx = vec_idx(row_id, col_id, n);
    for (int k = 0; k < n; k++) {
      int left_idx = vec_idx(row_id, k, n);
      int right_idx = col_iter(col_id, k, n);
      C[out_idx] = C[out_idx] + (A[left_idx] * B
  [right_idx]);
    }
  }
}
```

Listing 1: Basic Matrix Multiplication

```
std::vector<double> row_buf(n);
for (int row_id = 0; row_id < n; row_id++) {
  copy_row(A, row_buf.data(), row_id, n);
  for (int col_id = 0; col_id < n; col_id++) {
    int out_idx = vec_idx(row_id, col_id, n);
    double c_out = C[out_idx];
    for (int k = 0; k < n; k++) {
      c_out = c_out + (row_buf[k] * B[(n *
  col_id) + k]);
    }
    C[out_idx] = c_out;
  }
}
```

Listing 2: Matrix Multiplication with copy optimization

2.3 Matrix Multiplication with blocking/tiling optimization (BMMCO)

Matrix multiplication with tiling optimization brings a block of size b*b into the cache line and calculates the matrix multiplication result for the block. $C_b = C_b + A_b{*}B_b$. Block sizes 2,8,16,32,64 were considered. Total blocks = n/b.

3 Evaluation

The result of running the three different implementations are presented below. The matrix size(n*n) was made configurable and tried for the following values of n - 64, 128, 256, 512, 1024, 2048. The results were compared against CBLAS implementation for the validity of the implmentation.

3.1 Computational platform and Software Environment

- CPU: Intel's Knights Landing Processor
- CPU clock cycle: 1.4GHZ
- Memory Bandwidth: 102 GiB—sec
- L1 caches: 64 KB (32 KiB instruction cache, 32 KB data)
- L2 cache: 1MB
- Compiler: gcc 8.3.0 (O3 optimization)

```
int Nb = n / block_size;

std::vector<double> block_buf(3 * block_size * block_size);
double *AA = block_buf.data() + 0;
double *BB = AA + block_size * block_size;
double *CC = BB + block_size * block_size;

for (int i = 0; i < Nb; i++) {
  for (int j = 0; j < Nb; j++) {

    int cpos = get_start(i, j, block_size, n);
    copy_block(cpos, C, CC, block_size, n);
    for (int k = 0; k < Nb; k++) {
      int apos = get_start(i, k, block_size, n);
      int bpos = get_start(k, j, block_size, n);
      // copy blocks
      copy_block(apos, A, AA, block_size, n);
      copy_block(bpos, B, BB, block_size, n);

      // Basic matrix mul on block
      for (int row_id = 0; row_id < block_size; row_id++) {
        for (int col_id = 0; col_id < block_size; col_id++) {
          int out_idx = vec_idx(row_id, col_id, block_size);
          double c_cout = CC[out_idx];
          for (int m = 0; m < block_size; m++) {
            int left_idx = vec_idx(row_id, m, block_size);
            int right_idx = col_iter(col_id, m, block_size);
            c_cout = c_cout + (AA[left_idx] * BB[right_idx]);
          }
          CC[out_idx] = c_cout;
        }
      }
    }
    // Write result back
    write_block(cpos, CC, C, block_size, n);
  }
}
```

Listing 3: Matrix Multiplication block/tiling optimization

- OS: SUSE Linux Enterprise Server 15

3.2 Methodology

- The experiments were conducted on the KNL node of the Cori supercomputer at NERSC.
- GCC 8.3.0 was used to compile the code with O3 optimization.
- CBLAS(v20.9.1) installed in the KNL nodes were used
- All three implementations were tested on problem sizes - 64,128,256,512,1024,2048.
- The implementations assumed that the matrices were square matrices with equal number of rows and columns.
- The results of the matrix multiplication were compared against the CBLAS implementation. All values matched with the CBLAS calculation within an epsilson(1^{-5}) range.

Problem Size (N)	Basic Mmul (sec)	CBLAS (sec)
64.0	0.0	0.15
128.0	0.01	0.0
256.0	0.22	0.0
512.0	3.31	0.01
1024.0	26.71	0.08
2048.0	213.71	0.63

Table 1: Comparison of runtime between basic matrix multiplication and CBLAS

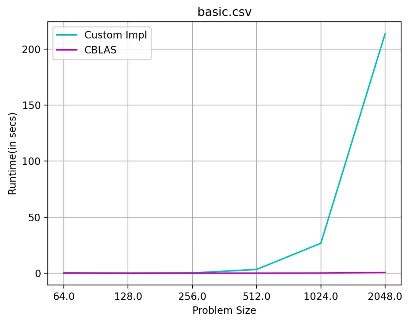

Figure 1: Comparison of MM runtime for different problem sizes

- The evaluation time of the custom matrix multiplication implementation was recorded and so were the CBLAS evaluation times.
- This was the only performance metric that was gathered. It was gathered by instrumenting the code. No external tools were used. This elapsed time was used to derive FLOPS which are presented in 3.6

3.3 Basic Matrix Multiplication(MM)

Basic matrix multiplication(MM) implementation presented in 2 was run on problem sizes - 64,128,256,512,1024,2048. Results are shown in Table:1. Runtime graph is shown in Fig:1. Flops graph is shown in Fig:2

3.4 Matrix Multiplication with copy optimization (MMCO)

Matrix multiplication with copy optimization(MMCO) implementation presented in 2 was run on problem sizes - 64,128,256,512,1024,2048. Results are shown in Table:2. Runtime graph is shown in Fig:3. Flops graph is shown in Fig:4

Problem Size (N)	Basic Mmul - copy optimized (sec)	CBLAS (sec)
64.0	0.0	0.15
128.0	0.01	0.0
256.0	0.08	0.0
512.0	0.62	0.01
1024.0	4.78	0.08
2048.0	37.63	0.63

Table 2: Comparison of runtime between copy optimized basic matrix multiplication and CBLAS

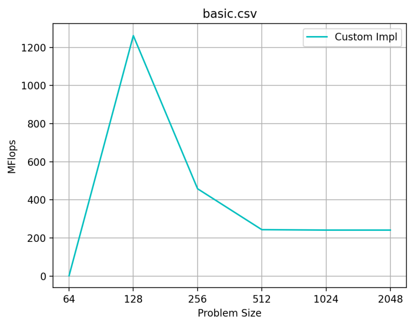

Figure 2: Comparison of MM flops for different problem sizes

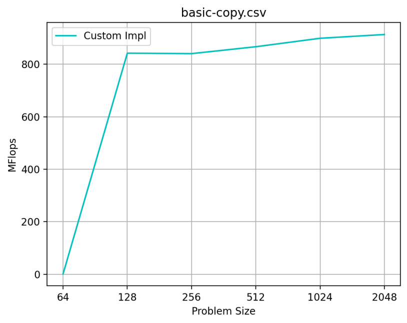

Figure 4: Comparison of MMCO flops for different problem sizes

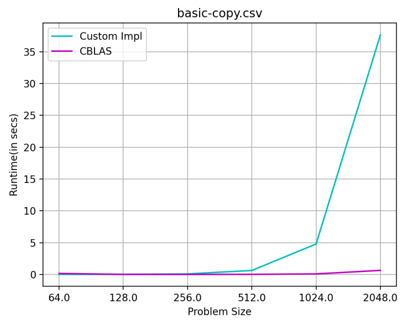

Figure 3: Comparison of MMCO runtime for different problem sizes

3.5 Blocked Matrix Multiplication (BMMCO)

Blocked Matrix Multiplication (BMMCO) implementation presented in 2 was run on problem sizes - 64,128,256,512,1024,2048. Block sizes - 2,8,16,32,64 were tried. Results are shown in Table 3. Runtime graph is shown in Fig:5. Flops graph is shown in Fig:6

3.6 Findings and Discussion

- Basic MM scaled exponentially in its runtime with the increase in problem sizes.
- Computational Intensity(CI) of Basic MM = f/m = $6*n^3+2*n^2/(2*n^3+n^2) \approx 3$
- Best FLOPS for MM was observed for problem size = 128
- MMCO performed better than better than MM. Runtime still scaled exponentially but was close to 10 times better than MM for higher problem sizes
- CI for MMCO = $(2*n^2+4*n^3)/(2*n^2+n^3) \approx 4$
- For problem sizes ¿ 128, MFLOPS averaged around 800 which is four times better than MM

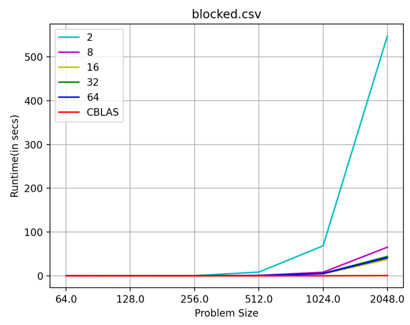

Figure 5: Comparison of BMMCO runtime for different problem sizes

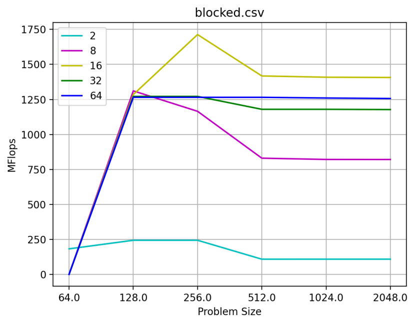

Figure 6: Comparison of BMMCO flops for different problem sizes

- BMMCO performed the best for block size = 16 and worst for block size = 2
- For block sizes less than 16, the cache was under utilized. For block sizes greater than 16, the cache seems to get thrashed which led to slow performance decline for higher block sizes.
- BMMCO achieved MFLOPS close to 1500 for problem sizes 256 and above. CI = f/m = (2*Nb+6n)/2*Nb+2n = Nb+3n/Nb+1 3(n/Nb)
- CI for BMMCO = $(2*N_b+6*n)/(2*N_b+2*n) \approx 3(n/N_b)$
- CI for BMMCO was better than CI for MMCO and CI for MM. This is also evident in Fig:6
- CBLAS performed better than all of these implementations. Even for the highest problem size 2048, it computed the result in less than a second.
- Almost all block size, seem to reach their peak FLOPS and then start to plateau for problem sizes 512 and above.
- Caches being limited in size can only fit so many elements in them. So, higher block sizes weren't necessarily helping.
- One surprising observation was that the runtime of MMCO and BMMCO(block size=16) was very close. I think this shows that BMMCO could be tweaked for more performance.

Problem Size (N)	Mmul(b=2) (sec)	Mmul(b=4) (sec)	Mmul(b=16) (sec)	Mmul(b=32) (sec)	Mmul(b=64) (sec)	CBLAS (sec)
64.0	0.01	0.0	0.0	0.0	0.0	0.14
128.0	0.06	0.01	0.01	0.01	0.01	0.0
256.0	0.48	0.09	0.06	0.08	0.08	0.0
512.0	8.57	1.01	0.58	0.69	0.64	0.01
1024.0	68.51	8.17	4.67	5.52	5.14	0.08
2048.0	547.52	65.39	37.41	44.23	41.24	0.63

Table 3: Comparison of runtime of Blocked Matrix Multiplication where b=n is the block size