Comparative Analysis of the Performance of Matrix Multiplication and Transposition Using Top Down Analysis

Billy Koech
CS246: Advanced Computer Architecture
Electrical Engineering SB
Harvard School of Engineering and Applied Sciences (SEAS)
Cambridge, United States of America

***Abstract*—Top Down analysis is performed on four different programs that implement variations of matrix multiplication and two different programs that implement matrix transposition. For matrix multiplication, the first method is a naïve implementation that involves element by element multiplication. The second version implements tiled multiplication as an optimization. The third, utilizes the optimized matrix library Eigen, and the fourth leverages another optimized library-OpenBLAS. The results show Backend Bound bottlenecks dominant are across all these implementations. In terms of runtime, OpenBLAS performs best followed by tiled then Eigen then naïve. For matrix transposition, a naïve implementation that involves element by element reassignment is compared against a transpose function from the optimized Eigen library. The results show that the naïve implementation performs better than the Eigen function likely due to Eigen's large Frontend Bound bottleneck.**

Keywords—pmu, top down analysis, backend bound, bottleneck

I. Introduction (*Heading 1*)

Top Down analysis is a method developed by Ahmad Yassin [1] to quickly identify computational bottlenecks in out-of-order processors. It reduces hundreds of performance counters to a few abstracted metrics[2] shown in Figure 1. Throughout this paper, these metrics will be referenced directly by the names or by the levels; level 1 refers to the top most part (Frontend Bond, Bad Speculation, Retiring and Backend Bound) and level 4 refers to the bottom part(Scalar, Vector, 3+ports, 2 ports, 1 port, 0 ports, MEM Bandwidth and MEM latency).

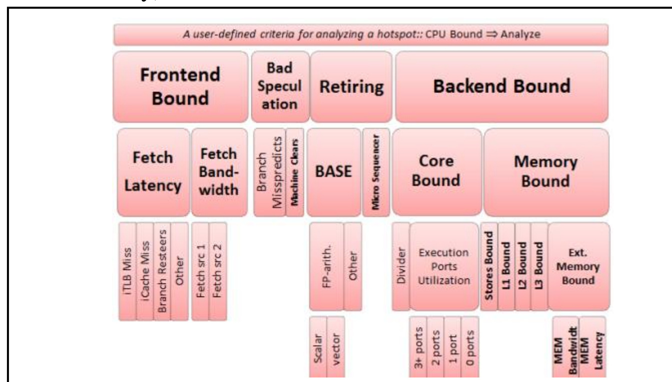


Fig. 1. Top-Down Analysis Hierachy by Ahmad Yasin[1].

This paper leverages a top down analysis python program written as a wrapper of perf known as pmu-tools [3] to collect data from performance counters. Pmu-tools is executed on the following programs

- Naïve Matrix Multiplication
- Tiled matrix multiplication
- Matrix multiplication using the Eigen library
- Matrix multiplication using the OpenBLAS library
- Naïve Matrix transposition
- Matrix transposition using the eigen library

The paper follows the following structure: Section II is dedicated to discussing the designs for matrix multiplication implementations. Section III does the same but for matrix transposition. Section IV discusses the methodology used to calculate runtimes and to perform top down analysis on the implemented programs. Section V, VII, present the performance results in terms of runtime and throughput; section VI, VIII then discus the performance bottlenecks.

II. Matrix Multiplication

A. *Naïve*

The naïve implementation follows an algorithm that calculates each entry as a sum of products. The c implementation is shown of an *out_rows x in_cols* matrix and an *in_cols x out_cols* below:

```
    for (int i = 0; i < out_rows; i++) {
        for (int j = 0; j < out_cols; j++) {
          for (int k = 0; k < in_cols; k++) {
            int a_index = i * out_cols + j;
            int b_index = i * in_cols + k;
            int c_index = k * out_cols + j;

 a[a_index]=a[a_index]+(b[b_index]*c[c_index]);
          }}}
```

Fig. 2. Naïve multiplication in C.

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©2019 IEEE

B. Tiled implementation

Tiled multiplication involves splitting the matrix into smaller blocks(tiles) of equal dimensions and computing the results of tile by tile. This setup takes advantage of spatial locality to prevent stalls due to misses. Appendix 1 shows the c implementation of tiled matrix multiplication where *tile_size* is the one side dimension of the tile, *tile_row* is the number of tiles per row of the matrix, and *in_cols, out_cols* are the dimensions of an *out_rows x in_cols* matrix and an *in_cols x out_col* matrix. This implementation assumes a square matrix so as to decrease the complexity of sweeping values as discussed later in the methodology.

C. Eigen Matrix Multiplicaiton

Eigen is a C++ template library for linear algebra[4]. It provides optimized modules for common matrix and vector operation such as multiplication. The program used in this paper is implemented using Eigen's Matrix class and leverages the multiplication method provided in the library. The Implementation is illustrated in Figure 3 where *dim* is the matrix dimension (assuming a square matrix).

```
#include <iostream>
#include <Eigen/Dense>

MatrixXd output(dim, dim);
MatrixXd m1 = MatrixXd::Random(dim, dim) * 10;
MatrixXd m2 = MatrixXd::Random(dim, dim) * 10;
// multiply and profile
output = m1 * m2;
```

Fig. 3. Matrix multiplication using Eigen in c.

```
cblas_dgemm(CblasColMajor,
  CblasNoTrans,
  CblasTrans,
  MATRIX_ROWS,
  MATRIX_COLS,
  MATRIX_ROWS,
  alpha,
  A,
  MATRIX_ROWS,
  B,
  MATRIX_COLS,
  beta,
  C,
  MATRIX_COLS);
```

Fig. 4. Matrix multiplication using OpenBLAS in c.

D. OpenBLAS Matrix Multiplicaiton

OpenBLAS(Basic Linear Algebra Subprograms) is an optimized library that provides standard interfaces for linear algebra[5]; amongst it methods, it implement high performance matrix multiplication based on a publication on the Anatomy of high performance matrix multiplication[6]. The multiplication program used in this paper uses the routine *cblass_dgemm()* to perform multiplication as shown in Figure 4. *cblass_dgemm()* is an implementation of the equation 1[7].

$$C = \alpha AB + \beta C \quad (1)$$

III. Matrix Transposition

A. Naïve Transpositon

The naïve implementation of transposition involves iterative element by element indexing and replacement as shows in figure 5. The matrix in figure 5 is of dimension *rows x column.*

Fig. 5. Naïve Transposition

```
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
      a[i * cols + j] = b[j * cols + i];
  }
}
```

B. Eigen Transpositon

In addition to multiplication, the Eigen library also has a member function that computes the transpose of a matrix. A snippet of the program is shown in Figure 6 below.

```
#include <iostream>
#include <Eigen/Dense>

MatrixXd output(dim, dim);
MatrixXd m1 = MatrixXd::Random(dim, dim) * 10;
output = m1.transpose();
```

Fig. 6. Eigen Transposition

IV. Methodology

A. Runtime performance measurement

Runtime is calculated using the clock() function of the c time.h library. The clock is recorded in a variable before beginning and after the termination of the target operation. The difference is taken and printed out and the time calculated as per the example in Figure 7.

```
    start_clock = clock();
    output = m1.transpose();
    end_clock = clock();
    cpu_time = ((double)(end_clock - start_clock))
 / CLOCKS_PER_SEC;
```

Fig. 7. Runtime calculation for transpsositon using Eigen's transpose method

B. Top-Down Analysis

Top-Down analysis is done using the `toplev.py` [2]analysis program from pmu-tools[3]. It is invoked on the benchmarks and executed via the scheduling program HTcondor. The processor on which the programs are run is a 3.6-full on Intel(R) Xeon(R) Platinum 8275CL CPU @ 3.00GHz processor.

Each program is drilled down up to four levels deep in the hierarchy .All the benchmarks were written as single-threaded programs therefore the following parameter is used to restrict outputs of `toplev.py` to just a single threaded:`--single-thread`. The programs were also restricted to just a single core using `taskset -c 0` so as to decrease the complexity of data collected by the performance monitoring units across all the multiple cores.

All the implementations for multiplication and transposition are parametrized to take a user defined size for a matrix and from that compute the results for a square matrix of given dimension. The following is the selected sweeping range over which Top-Down analysis is done: 8, 32, 256, 512, 1024, 2048 (that is, 8 refers to an 8 x 8 matrix).

The tiled implementation takes an additional parameter, tile size, which (as the name suggests) defines the size of the tile. The following tile size values are used in the sweeping range when performing top down analysis for just the tiled implementation: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024. Note that if the tile size is larger than the matrix then computation is skipped. When comparing the performance of the tiled implementation to the other implementation, the result with the best performing is used.

For each benchmark the output from `toplev.py` is redirected to a csv file using the `-x, -o filename.csv` parameters; the csv file is then imported into R (a standard analysis package) for analysis.

V. Matrix Multiplication Results

A. Runtime performance

Figure 8 shows the runtime results for all the benchmarks. For small matrix sizes up to about 256 by 256, the difference seems to be marginal as the multiplication is computed within milliseconds for all the implementations. The differences in performance become evident for matrices of size 512 by 512 and greater. The OpenBLAS implementation has the highest performance as it maintains its runtime close to zero even for larger matrices (such as the 2048 by 2048).

As mentioned in the methodology the tiled implementation involved sweeping different tile sizes and selecting the tile size with the lowest runtime to compare against the other implementations. The performance for

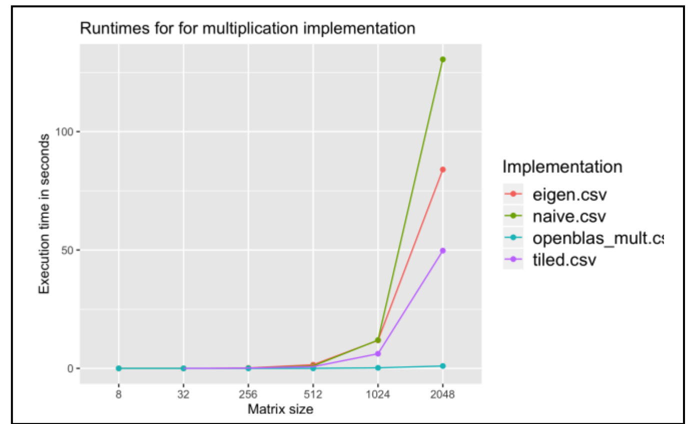


Fig. 8. Runtimes for matrix multiplicaiton implementation

different tile sizes can be seen in Figure 9. Tile size 16 has the lowest computation time and it is used in the comparison in Figure 8.

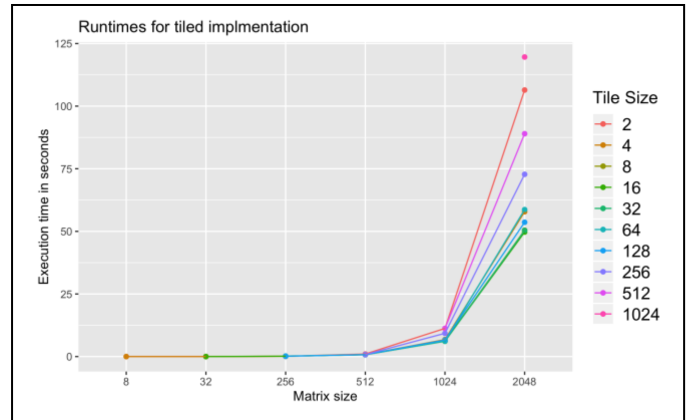


Fig. 9. Runtimes for tiled matrix multiplication

B. Top Down Analysis – Instructions Per Cycle(IPC)

The instructions per cycle (IPC) is measured for each benchmark and shown below in Figure 10:

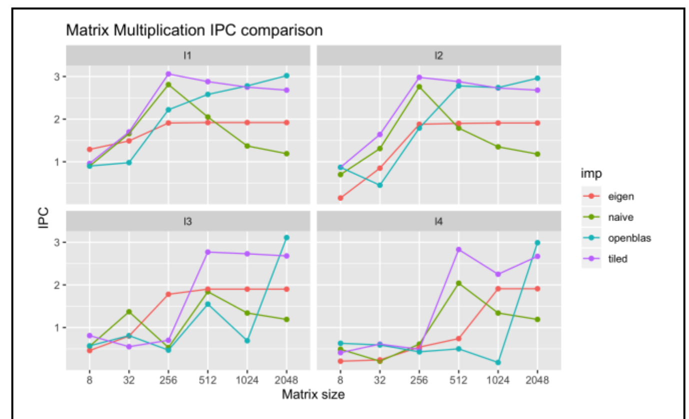


Fig. 10. IPC for the four benchmarks. Each facet represent different levels of Top-Down Analysis. Top left is level 1, top right level 2, bottom left level 3, bottom right level 4.

Interesting enough the tiled implementation exhibits generally high throughput at all four levels. The naïve implementation's throughput is observed to decrease as the matrix size increases. The Eigen implementation exhibits a

steady throughput rate for matrices of sizes between 256 and 2048 in the first three levels. OpenBLAS exhibits a steady increase in throughput in the level 1 and level 2. In levels 3 and 4 the throughput seems to decrease up to 1024 where it spikes up for OpenBLAS.

As with the tile implementation runtimes, the tile implementation IPC values were also measured for tiles of different sizes. This is shown below in Figure 11:

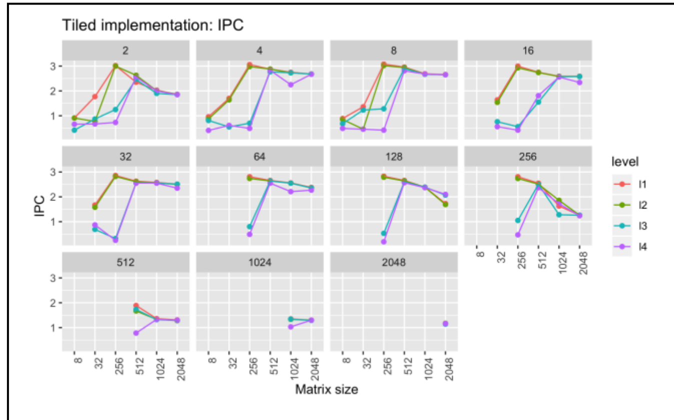


Fig. 11. IPC for tile sizes. Each Facet represents the tile size use to perform the multiplication. Hence the largest tile size 2048 has only one point the largest matrix used in this empirical study is 2048 by 2048. The different colored lines resemble the different Top-Down analysis levels.

VI. Matrix Multiplication Bottlenecks

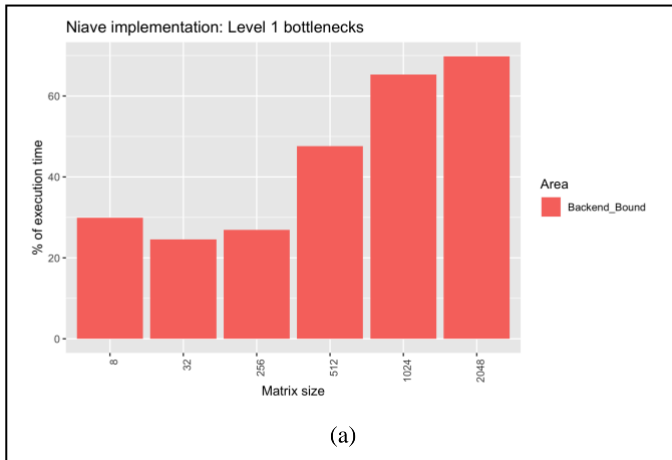
A. *Naïve implementation*

From Figure 12 Backend Bound of Top-Down analysis is flagged in level 1 predominantly across all the different matrix sizes. Furthermore, the percentage of execution time as a result of this bottleneck increases as the matrix increase.

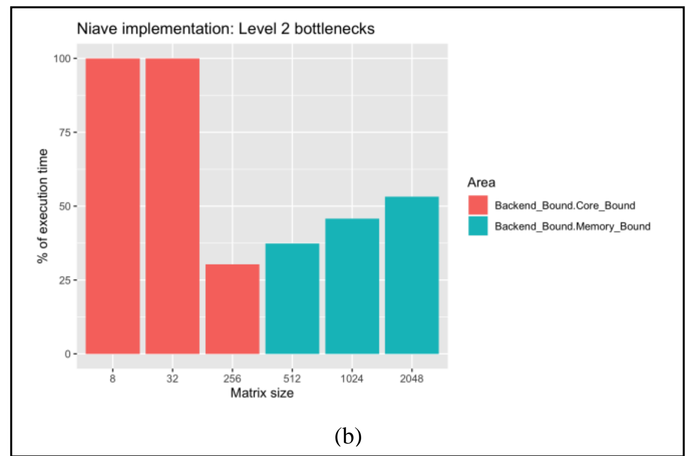
Issues attributed to Backend Bound include data cache misses or stalls due to divider overload[1]. This makes sense for the naïve implementation as we expect more cache misses for larger matrix sizes.

A dive into the next level reveals two dominant bottlenecks: Core Bound and Memory Bound. As the matrix size increase the bottleneck shifts from Core Bound to Memory Bound. This is also expected for naïve given that Memory Bound can be a result of a load missing all caches.

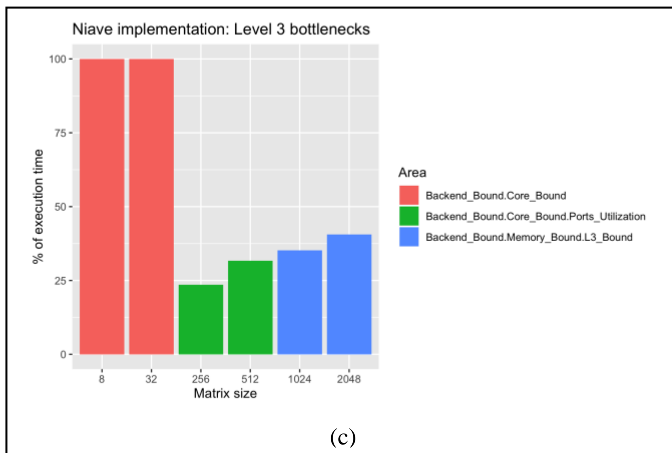
At the third level it is revealed that the for larger matrices the major bottleneck is L3 bound. It also makes sense for larger matrices because naïve does not take advantage of spatial locality therefore when misses occur data has to be fetched from the lowest cache (that is closest to the memory.) This is furthermore corroborated at the fourth level where the major bottleneck for large matrices is L3 Hit latency.



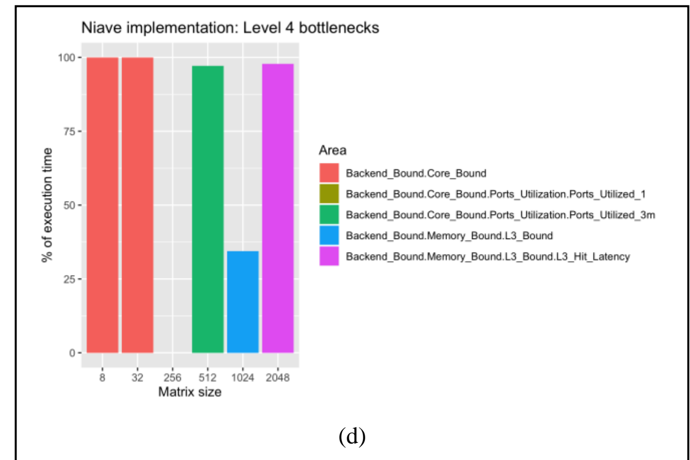
(a)



(b)

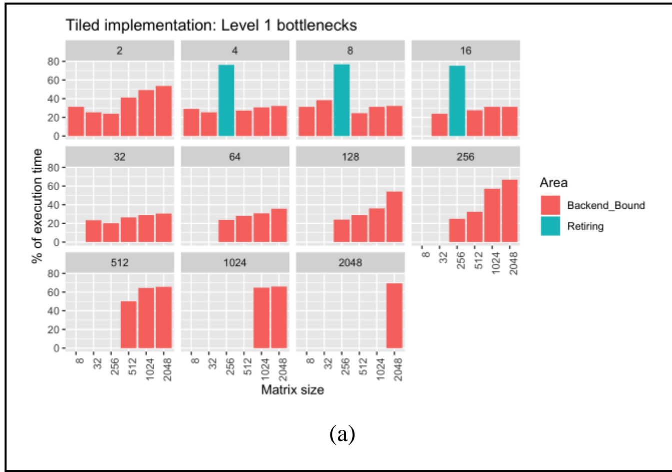


(c)

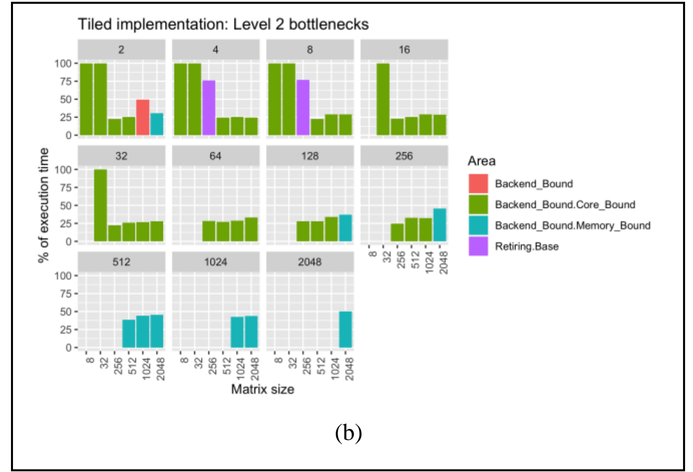


(d)

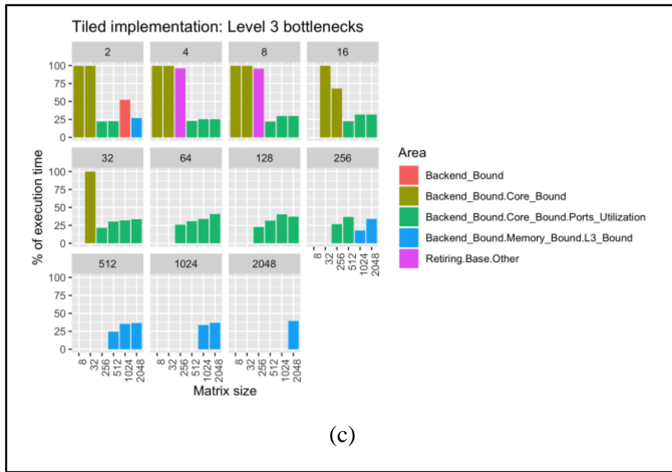
Fig. 12. Top Down Analysis of Naïve Implementation.



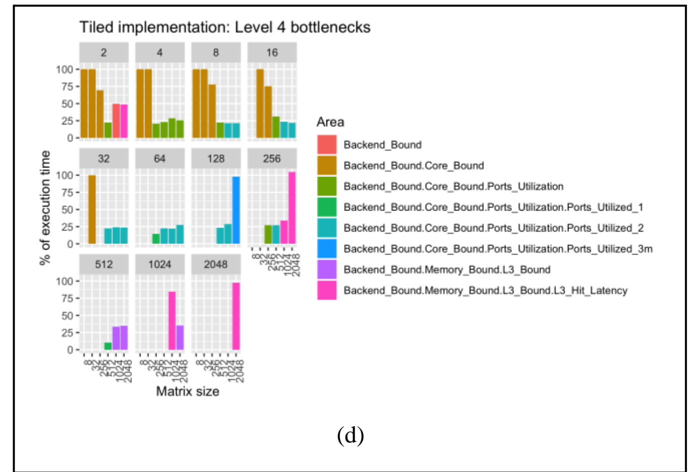
(a)



(b)



(c)



(d)

Fig. 13. Top Down Analysis of Tiled Implementation. The facets represent tiles of different sizes

B. *Tiled implemenation*

Similarly, for the tiled implementation at level 1, the dominant bottleneck is Backend Bound. Three out of the 10 samples (all three of matrix size 256) exhibit Retiring bottlenecks (Figure 13 (a)) . Further examination in level 2 (Figure 13 (b)) shows that for tile sizes between 2 and 28 the bottleneck is Core Bound. Matrices computed with tiles of size 512, 1025 and 2048 exhibit Memory Bound as the major bottleneck. This makes sense because for large tile sizes, the multiplication procedure begins to resemble that of naïve.

At the third level (Figure 13 (c)) it is revealed that the programs that are bottlenecked by Core Bound slots is as a result of sub optimal port utilization[1]. This is corroborated at the 4th level (Figure 13 (d)) where large matrix sizes are limited by sub-optimal port utilization.

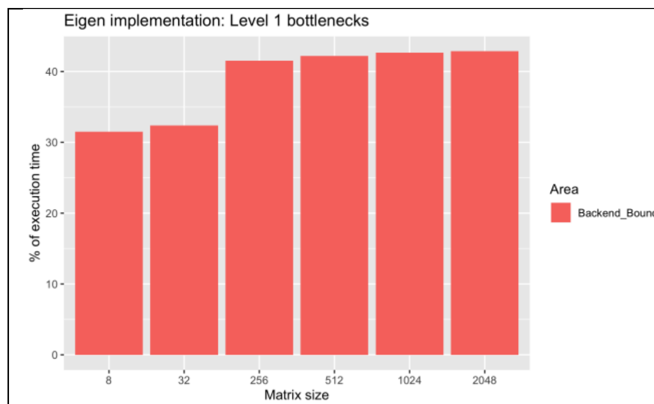
C. *Eigen Library Implementaion*

Eigen Exhibits bottleneck trends similar to that of tiled. The first level is dominated by Backend Bound (Figure 14 (a)) across the different matrix sizes. In the second level (b) it is revealed that the bottleneck is Core Bound. The bottleneck for larger sized matrices seems to be due to sub optimal port utilization as revealed in level 3 and level 4 (Figure 14 (c) and (d)).

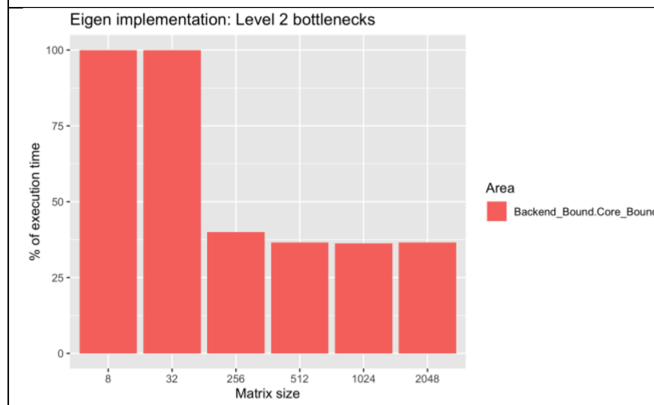
D. *OpenBLAS Library Implementaion*

OpenBLAS differs from the rest of the implementation as it is the only one to exhibit Retiring as a bottleneck for larger matrix sizes such as 2048 (Figure 15 (a)). Investigation of the 4th level shows that bottleneck lies in Vector Floating Point (FP) Arithmetic (Figure 15 (d)). This is an indication that improvement can be made to the code by vectorizing in order to improve performance. Since a high retiring rate corresponds to a high IPC [1]we would expect a significant increase in the IPC if the code's vectorization is improved.

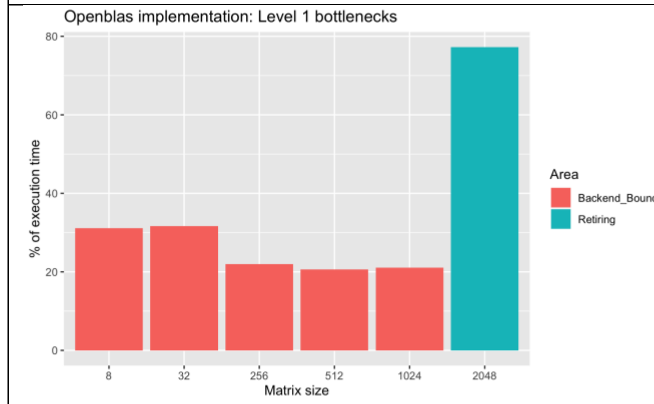
For the smaller sized matrices, the bottleneck (Figure 15 (a), (b) and (c)) is similar to the previous three implementations: it is Core Bound



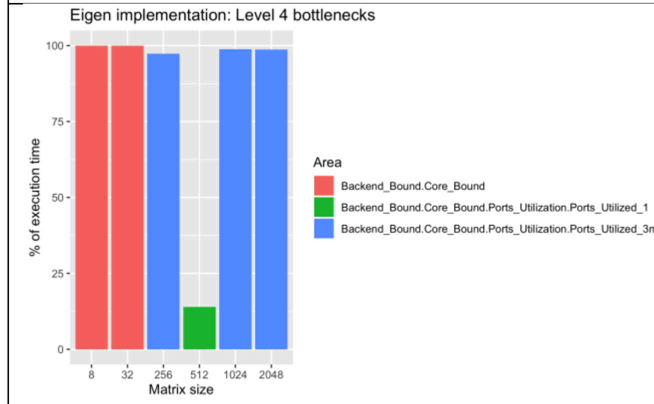
(a)



(b)

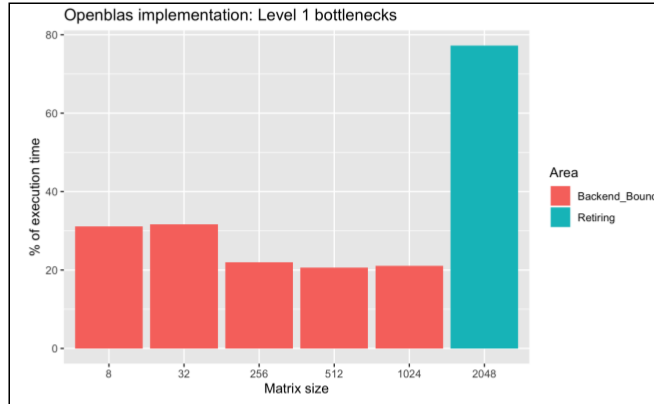


(c)

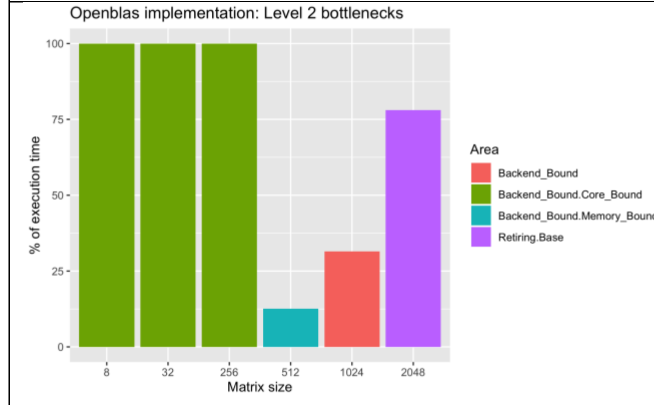


(d)

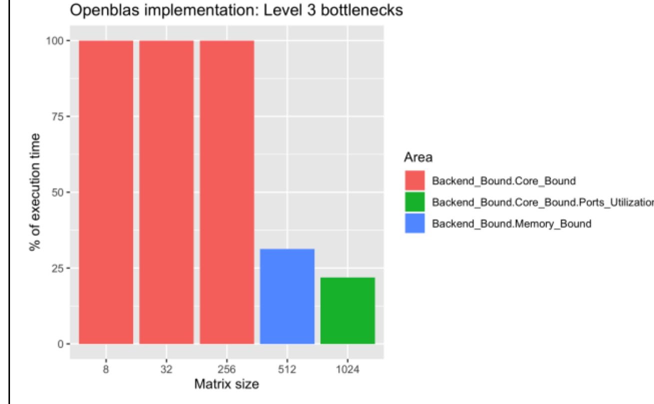
Fig. 14. Top-Down Analysis of Matrix Multiplication using Eigen



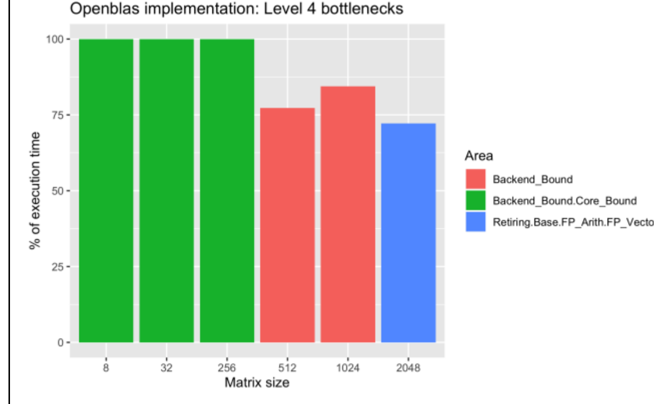
(a)



(b)



(c)



(d)

Fig. 15. Top Down Analysis of Matrix multiplication using OpenBLAS

VII. Matrix Transpostion Results

A. *Runtime performance*

Figure 16 shows the runtimes for the naïve implementation and the eigen implementation of transpose:

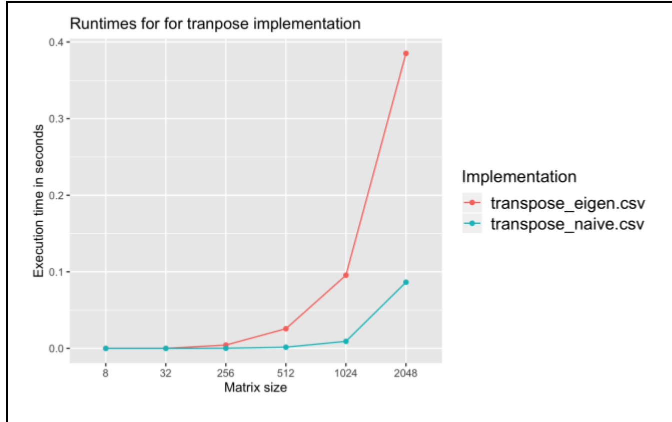


Fig. 16. Runtimes for the naïve implementaiton of transpose and the eigen library transpose function.

Surprisingly, for large size matrices the naïve implementation seems to perform significantly faster than the eigen inbuilt transpose function. The bottlenecks for both implementations are examined in the bottleneck section VIII.

B. *Top-Down Analysis – Instruction Per Cycle(IPC)*

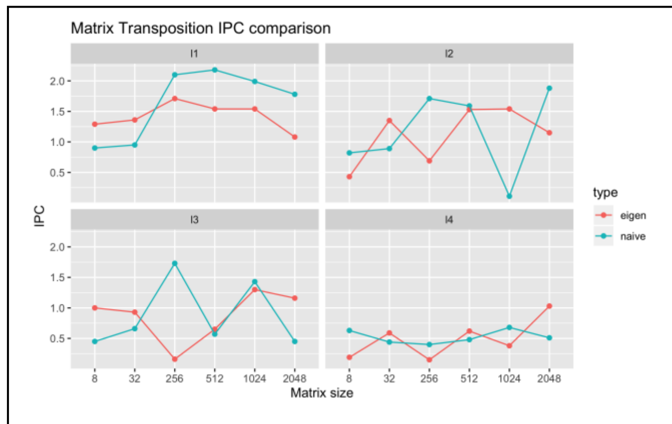


Fig. 17. IPC for the naïve and eigen transpose implementations are collected from Top-Down Analysis. Each facet represent different levels of Top-Down Analysis. Top left is level 1, top right level 2, bottom left level 3, bottom right level 4.

At the first level the naïve implementation seems to have a higher throughput than the eigen method. At the other levels the IPC keeps interchanging between the two implementations.

VIII. Matrix Transpostion Bottlenecks

A. *Naïve implementation*

From Figure18 (a) it can be observed that the dominant bottleneck across the matrix sizes is Backend Bound. The percentage of execution time taken by Backend Bound increases as the matrix size increase from 256 to 2048. In levels 2 3 (Figure 18 (c) and (d)) and the pmu does not specify the port therefore we cannot narrow down on the ports involved in the sub optimal port utilization. However, at level 4 (Figure 18 (d)), for the largest matrix (2048) the pmu specifics port 1 as the bottleneck.

B. *Eigen Library implementation*

From Figure 19 (a) it can be observed that the major bottleneck for larger sized matrices is Frontend Bound. At the level 2 and 3 (Figure 19 (b) and (c)) the major bottleneck is revealed to be Frontend Latency for larger sized matrices.

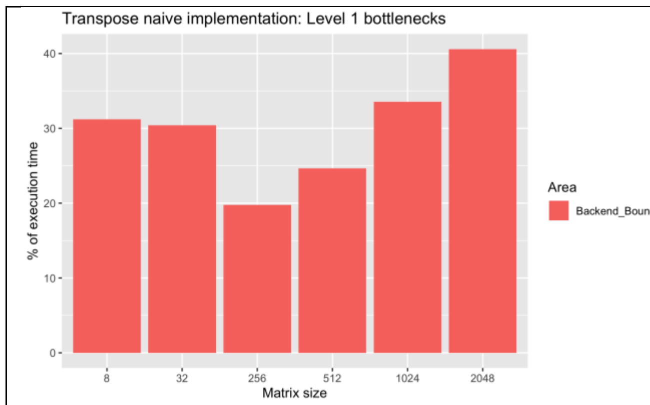
The Frontend Latency is a metric accounting for cases that lead to fetch starvation such as instruction cache misses and Instruction Length Decoding [1]. This is likely to be the reason for the significantly low performance of Eigen library's transpose method in comparison to the naïve implementation.

IX. Conclusion

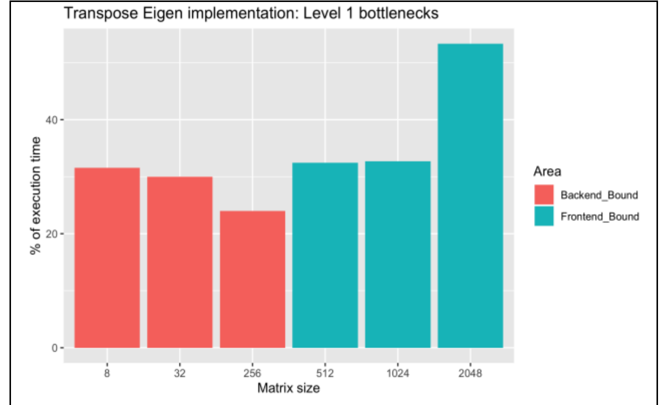
Top-Down analysis is performed on 6 different programs: Naïve matrix multiplication, Tiled matrix multiplication, Eigen library matrix multiplication, OpenBLAS library matrix multiplication, naïve matrix transposition and Eigen library matrix transposition.

The first four programs realize the same functionality- matrix multiplication but the OpenBLAS exhibits the fastest performance while the naïve exhibits the least performance. OpenBlAS's bottleneck lies in the Retiring area(Vector Floating Point Arithmetic) suggesting that vectorizing it's code could further improve it performance. Majority of the other implications are Backend Bound (Core Bound and Memory Bound) which is consistent with the challenge of matrix multiplication – sub optimal port utilization and data cache misses.

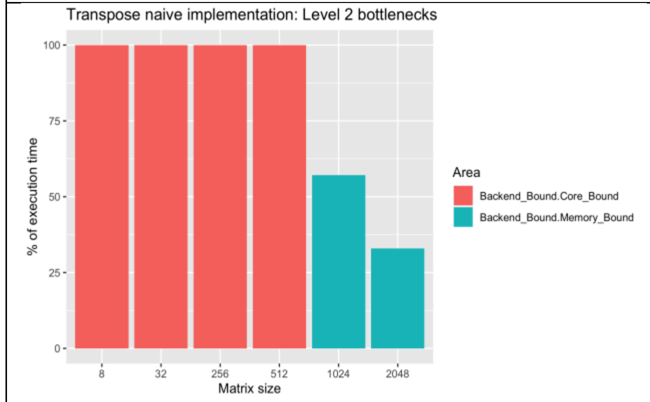
The last two programs perform matrix transposition. The results show that the naïve implementation performs better than the Eigen Library implementation of transpose. We suggest that this low performance is attributed to Frontend Latency due to instruction cache misses as per the Top-Down Analysis results



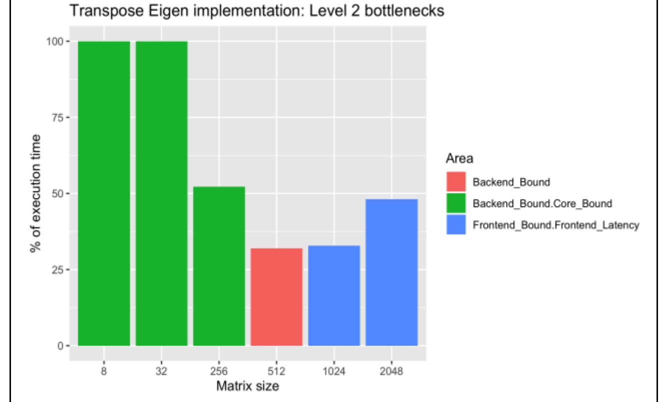
(a)



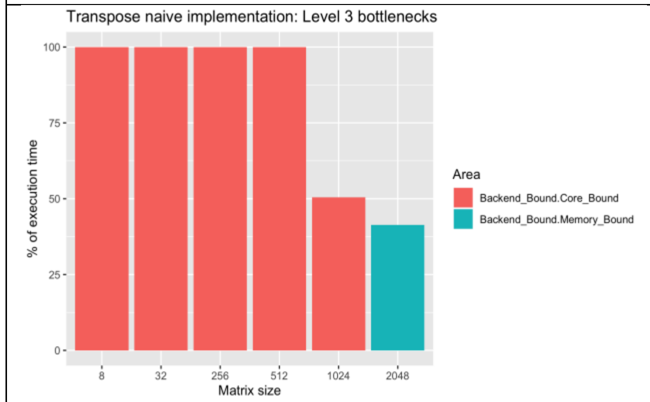
(a)



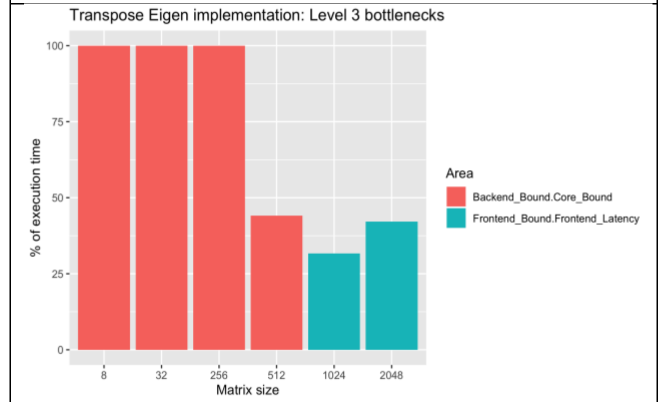
(b)



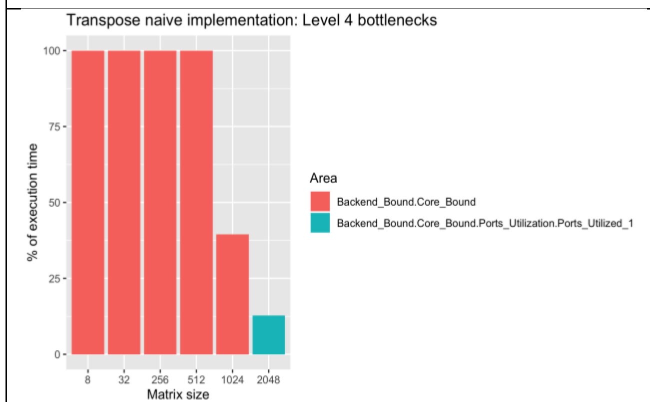
(b)



(c)



(c)



(d)

(d)

Fig. 18. Top-Down Analysis of naïve transpose

Fig. 19. Top Down Analysis of transpose using the Eigen library

REFERENCES

[1] A. Yasin, “A Top-Down method for performance analysis and counters architecture,” in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 35–44.

[2] A. Yasin, “Top-down Microarchitecture Analysis through Linux perf and toplev tools,” p. 28.

[3] A. Kleen, *andikleen/pmu-tools*. 2019.

[4] “Eigen.” [Online]. Available: http://eigen.tuxfamily.org/index.php?title=Main_Page. [Accessed: 22-Nov-2019].

[5] Z. Xianyi, *xianyi/OpenBLAS*. 2019.

[6] J. Huang and R. A. van de Geijn, “BLISlab: A Sandbox for Optimizing GEMM,” *arXiv:1609.00076 [cs]*, Aug. 2016.

[7] admin, “Multiplying Matrices Using dgemm,” 17:38:02 UTC. [Online]. Available: https://software.intel.com/en-us/mkl-tutorial-c-multiplying-matrices-using-dgemm. [Accessed: 10-Dec-2019].

.

APPENDIX I: TILED MATRIX MULTIPLICATION

```
  for (int it = 0; it < tile_rows; it++) {
    for (int jt = 0; jt < tile_rows; jt++) {
      int a_ti = (tile_size * it * in_cols) + (jt * tile_size);
      for (int kt = 0; kt < tile_rows; kt++) {
        int b_ti = (tile_size * it * in_cols) + (kt * tile_size);
        int c_ti = (tile_size * kt * in_cols) + (jt * tile_size);

        for (int im = 0; im < tile_size; im++) {
          for (int jm = 0; jm < tile_size; jm++) {

            int a_mi = (im * in_cols) + jm + a_ti;
            for (int km = 0; km < tile_size; km++) {
              int b_mi = (im * in_cols) + km + b_ti;
              int c_mi = (km * in_cols) + jm + c_ti;

              a[a_mi] = a[a_mi] + (b[b_mi] * c[c_mi]);
            }
          }
        }


      }
    }
}
```