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

By\ Khalid Al-Hawaj

COE403/COE501 Term 242

1 Introduction

In the last programming assignment, we learned about benchmark suites and their importance in studying and understanding differences between various micro-architectures. To this end, to be representative of the full potential of a micro-architecture, a benchmark implementation must utilize and leverage all the features provided by the micro-architecture. The `naïve` implementation acts as a baseline to demonstrate benefits of different architectural details. Compilers produce a low-level (i.e., assembly language) implementation that is functionally equivalent to the high-level implementation of the algorithm. As there are multiple possible low-level implementations that are functionally equivalent to a high-level implementation, compilers have the liberty to choose any low-level implementation applicable while optimizing the execution. The specific micro-architecture is provided to the compiler as an argument. The compiler, then, leverages insights and details of the specified micro-architecture to choose a low-level implementation that is more optimized. However, there are some other optimization that requires transformations, which the compiler is incapable of performing easily. These transformations require algorithmic or data organization changes that must be handled at implementation-level by the programmer.

While some micro-architectures have more purpose-built features giving them unique insights, there are some universal insights that are applicable to all micro-architectures due to the semantics of the instructions more-so than the micro-architectural details. One of these universal insights is the overhead of branch resolution. As instruction fetch is performed at the beginning of the execution cycle while updating the program counter happens towards the end, it is very hard to handle branch resolution without impacting the throughput of any micro-architecture. Moreover, the instructions needed to implement loop semantics (consisting of branching and counter incrementing) constitute an overhead that reduce the performance of the intended functionality (i.e., loop iteration body). As a result, compilers employ universal optimizations such as loop unrolling and static branch analysis.

One of the architectural insights that we will consider 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. As a result, it is important that the implementation employs command-line arguments to accommodate different cache hierarchy.

In this assignment, you are asked to implement a new `opt` implementation for the `mmult` benchmark. The new `opt` implementation shall leverage blocking to increase cache utilization regardless of the size of the input matrices. Then, you should compare the improvement of performance between the `opt` implementation and `naïve` implementation. The remaining sections are as follows: Section 2 explains blocking the `mmult` benchmark; Section 3 discusses a pseudo-code implementation; Section 4 details the deliverables and the structure of the report to be submitted.

2 Increase Utilization Through Blocking

The `naïve` implementation focuses making the output local (i.e., increase utilization) rather than the inputs. Let us consider running the following `naïve` implementation on a hypothetical microarchitecture:

Algorithm 1 Naïve `mmult` Implementation

```
1:  procedure MMULT($A, B$)
2:      $R \leftarrow empty\_matrix(M, P)$
3:      for $i = 1 \rightarrow M$ do
4:          for $j = 1 \rightarrow P$ do
5:              $val \leftarrow 0$
6:              for $k = 1 \rightarrow N$ do
7:                  $val$ += $A[i][k]$ * $B[k][j]$
8:              end for
9:              $R[i][j] \leftarrow val$
10:         end for
11:     end for
12:     return $R$
13: end procedure
```

Figure 1 visualizes the execution of the above `naïve` implementation, assuming: (1) a set-associative cache with four sets and some limited associativity, and (2) a cacheline can hold four elements. The associativity is not assumed as it does not change the execution visualization; however, the associativity impacts the performance as will be discussed later. The output's cacheline is not highlighted in Figure 1 as the compiler can optimize the access to the output element into a register, which will be stored once at the end of the innermost loop.

As can be observed in Figure 1, the `naïve` implementation calculates each output element to conclusion. The implementation maximizes spatial locality for the accessed row from the A matrix; however, as we assume row-major order storage, the spatial locality for matrix B is very minimal. The implementation would have to access 30 elements (i.e., 15 row elements and 15 column elements)–the equivalent of 18 cachelines–before the same cacheline is accessed again to exhibit spatial locality. Assuming a moderate associativity of 2 ways, all the access of the `naïve` implementation would incur a capacity miss penalty. Even if the cache is organized as fully-associative cache, a capacity miss would still happen as the cache can hold eight cachelines in total. As for temporal locality, the `naïve` implementation some temporal locality for the row from A matrix; however, the column from B matrix would only exhibit temporal locality once the algorithm starts evaluating the second row of the output. As a result, temporal locality is extremely low and most cache organizations cannot exploit such sparse temporal locality (*note:* sparse temporal locality refers to the number of unrelated memory access between subsequent memory access to the same address).

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.

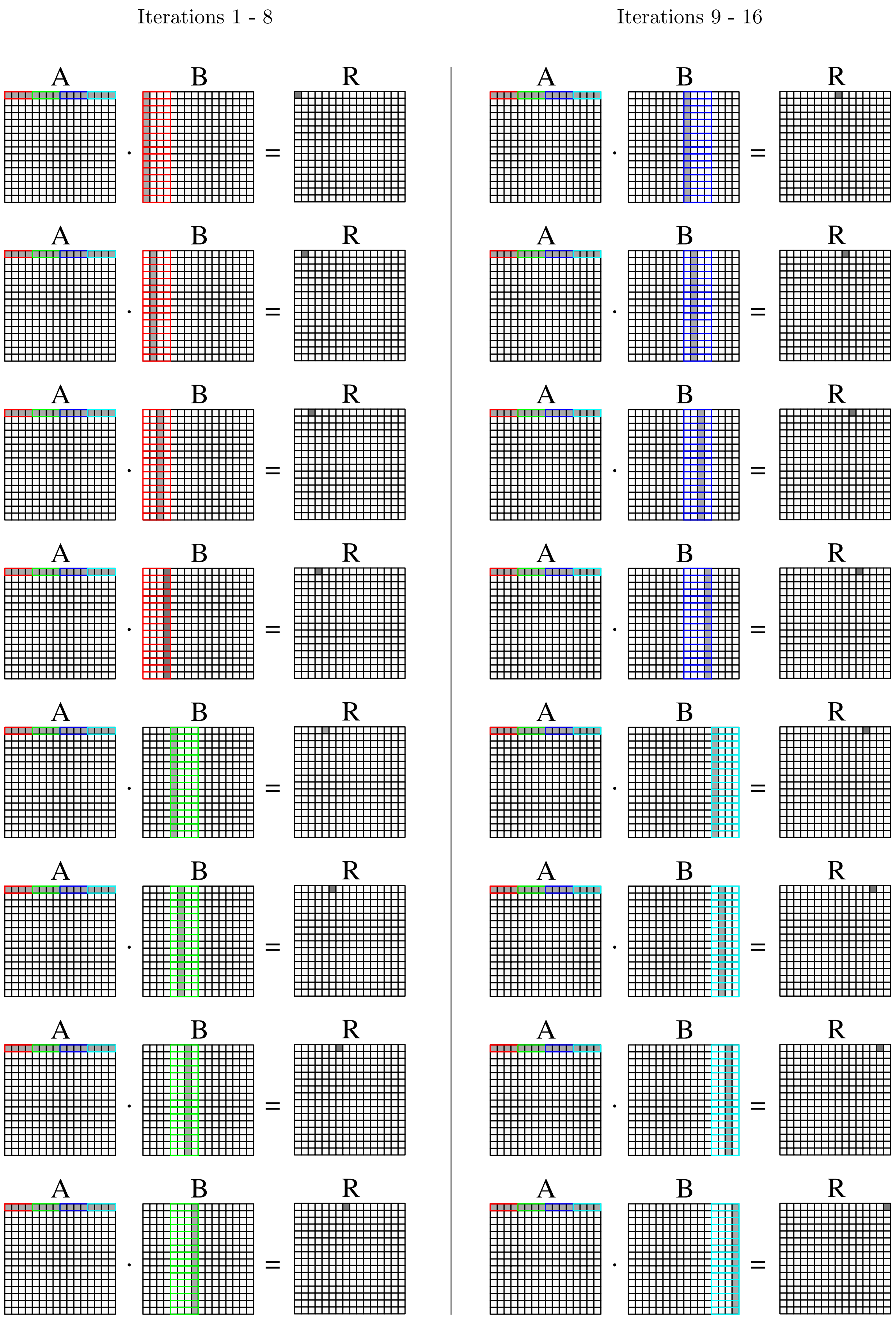

Figure 1: `Naïve` **Implementation Execution**—Elements being accessed (whether read or written) are shown in gray. Elements residing on the same cacheline are outlined with the same color. The coloring of cachelines indicates different set. The cache is assumed to have four sets. The associativity of the cache is not indicated in the figure as it does not affect the presentation of the execution.

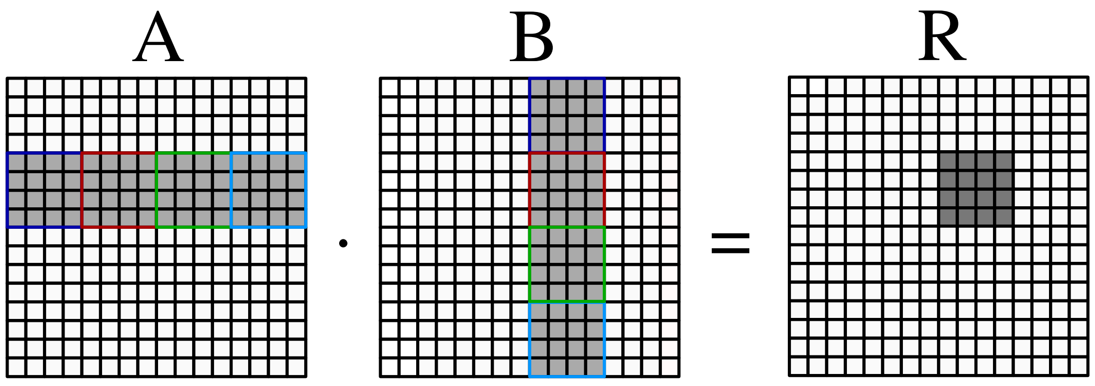

Figure 2: `opt` **Implementation Access Pattern**—The out and input is divided into a $b \times b$ sub-matrices. Then, each $b \times b$ sub-matrix of the output is calculated by accumulating the matrix-multiplication of all $b \times b$ sub-matrices. There are multiple ways to approach the locality depending on which sub-matrix is stationary (i.e., either sub-matrices from A are stationary, sub-matrices from B are stationary, or out-sub-matrices are stationary).

Figure 2 visualizes the execution of the proposed access pattern to increase cache utilization. One can see that the inputs and outputs are blocked into a $b \times b$ sub-matrices, where $b = 4$. As a result, to compute a 4×4 sub-matrix, the rows from the A matrix are blocked into four 4×4 sub-matrices. The columns from the B matrix are, also, blocked into four 4×4 sub-matrices. Then, corresponding sub-matrices from matrix A and matrix B are matrix-matrix multiplied to compute a partial output sub-matrix. When *all* four sub-matrices from matrix A are multiplied with their respective four sub-matrices from matrix B and the output is summed, the complete output sub-matrix. Depending which sub-matrix is stationary, different blocking implementations can yield varying utilization of the cache.

3 Optimize `mmult` Execution By Increasing Cache Utilization

In this assignment, you are tasked with adding a new implementation "`opt`" for the "`mmult`" benchmark, that you added in the previous programming assignment. The new implementation "`opt`" shall increase cache utilization through blocking as its architectural insight to improve the performance when compared to the `naïve` implementation. As discussed extensively in Section 2, there are multiple ways to approach blocking to increase cache utilization. Here is one *possible* implementation:

While the general idea of blocking is similar in all approaches, the decision behind which sub-matrix to make stationary (i.e., the sub-matrix that is enumerated last, thus persisting across multiple iterations) might make an impact on performance. The aforementioned implementation makes the output sub-matrix stationary, while it enumerates through all possible input sub-matrices. It is easy to highlight which sub-matrix is stationary by figuring out the outermost loop. In the case of the above implementation, the outmost loops enumerate through all output sub-matrices (i.e., the i and j indices with b step).

4 Deliverables and Report

The work required for this assignment is divided into two parts: deliverables, and a report.

4.1 Deliverables

There are three deliverables for this assignment. These deliverables and their explanations are as follows:

1. **Naïve Implementation:** A C-code of the `naïve` implementation from the first programming assignment. This implementation *can* be used as as a golden reference, if it was verified correctly; however, it is advised to use datasets that are saved into files and loaded, instead. This `naïve` implementation should operate on IEEE754 single-precision floats, as explained in the first programming assignment.

Algorithm 2 Optimized `mmult` Implementation

```
procedure MMULT_OPT($A, B, b$)
    for $i = 1 \to M$, $step\ b$ do
        for $j = 1 \to P$, $step\ b$ do
            $R[i][j] \leftarrow 0$
        end for
    end for
    for $ii = 1 \to M$, $step\ b$ do
        for $jj = 1 \to P$, $step\ b$ do                          # For each output sub-matrix
            for $kk = 1 \to N, step\ b$ do                        # For each input sub-matrix
                for $i = ii \to min(ii + b, M)$ do                # Perform sub-matrix multiplication
                    for $j = jj \to min(jj + b, P)$ do
                        $val \leftarrow R[i][j]$
                        for $k = kk \to min(kk + b, N)$ do
                            $val += A[i][k] * B[k][j]$
                        end for
                        $R[i][j] \leftarrow val$
                    end for
                end for
            end for
        end for
    end for
    return $R$
end procedure
```

2. **Optimized Implementation:** A C-code of the `opt` implementation described in Section 3. The pseudo-code provided can be used as a starting point. The choice of the block size should be passed by the user through the arguments. This `opt` implementation should operate on IEEE-754 single-precision floats, similar to the naïve implementation.

3. **Test Cases and Results:** Logs and print-outs of testing and verification as performed on all implementations to verify their executions. All implementations should execute correctly on *all* datasets, as detailed in the first programming assignment (i.e., datasets: testing, small, medium, large, native).

4.2 Report

Each student who worked on the assignment is expected to submit a report. This report details the students work on the deliverables and discusses their observations and findings. The bulk of grading will be based on the report, as it reflects their effort; ergo, each student should make sure to allocate enough time to polish their report. At minimum, the report is expected to have the following sections:

1. **Introduction:** A section that introduces the problem and the goal of the report. The section should summarize the student's contributions and key observations. The section should include a paragraph summarizing the student's understanding of the matrix-matrix multiplication operation.

2. **Collaboration:** Since students are allowed to discuss the naïve implementation, and the optimized implementation, the report should include a section that would details the collaborations between the student and other students. The section should mention other students by name and what aspect was discussed with them. **Make sure that collaboration is NOT allowed while implementing any code OR writing the report. Each student is ONLY allowed to collaborate with other students by discussing improvements and implementation aspects, but not collaborate during the actual implementation or writing the report.**

3. **Naïve Implementation:** A section that discusses the naïve implementation of and its algorithm. The student is expected to discuss implementation aspects and details their experience. The section should discuss whether there were bugs and how the student either resolved them or would have resolved them. The section should also discuss **insights** and **observations**.

4. **Optimized Implementation:** A section that discusses the optimized implementation and its algorithm. The section should discuss the student's understanding of the blocked matrix multiplication algorithm. The student is expected to discuss implementation aspects and details their experience. The section should discuss whether there were bugs and how either the student have resolved them or would have resolved them. The section should also discuss **insights** and **observations**.

5. **Evaluation:** A section that discusses the evaluation methodology, results, and insights. The section should start by detailing the evaluation methodology and justify it. The section should continue on by detailing and discussing the evaluation results and present these results in either tables or figures (whatever is appropriate). It is unacceptable for this part of evaluation to be devoid of any tables or figures. The section should conclude with results analysis and takeaways. One possible sweep (i.e., to study the correlated trend) to be performed is the optimal size for the blocks (i.e., b). Also, exploring the optimal selection for the stationary sub-matrix. Most discussions has to be backed-up with figures, graphs, and data. Try your best to perform sweeps to explore the design space. Try to leverage speed-up figures and comparisons with the baseline (i.e., the naïve implementation) to highlight interesting trends or insights. Try to have one figure showing the speedup of the optimized implementation compared to the naïve implementation. You should demonstrate the evaluation for all available datasets. Try to see if there are any interesting insights of the speed-up with varying block size and dataset sizes. Every result you present should be discussed and explained (i.e., the why and how behind the results to explain it). An unexplainable outcome or result is as good as non-existent.

6. **Conclusion:** A section that summarizes the students observations, insights, and findings.

7. **References:** A section with citation to sources and material that helped in the assignment. At minimum, this document and the document for the first assignment should be used as a reference in the assignment. The Git repository for the YABMS [1] should also be a reference for the assignment.

References

[1] Khalid Al-Hawaj. *YABMS: Yet Another Benchmark Suites.* `https://github.com/hawajkm/YABMS`. Accessed: 2025-02-21.