# Tile Size Selection Based on Matrix Dimension Factors and Cache Line Sizes

Larry Wong
*University of Michigan - Ann Arbor*
Ann Arbor, United States
larwong@umich.edu

Yuchen Xia
*University of Michigan - Ann Arbor*
Ann Arbor, United States
stilex@umich.edu

Wynn Kaza
*University of Michigan - Ann Arbor*
Ann Arbor, United States
wynnkaza@umich.edu

***Abstract*—The effectiveness of matrix multiplication is often constrained by memory access patterns and lack of cache awareness. This paper explores cache tiling, a technique aimed at enhancing spatial and temporal locality through developing a cache aware algorithm. We implement a cache tiling pass in LLVM, alongside an algorithm for finding the optimal tile size to reduce cache misses. Furthermore, the research introduces a novel cache tiling technique designed to address instruction overhead with bound checking on current cache tiling algorithms. This optimization helps to reduce the amount of unnecessary instructions within our for-loop by ignoring bound checking. To support this optimization, we designed a new algorithm that finds the optimal tile size, even with this new limitation. Furthermore, we present a theoretical methodology for quantifying the total number of cache misses within the context of a fully-associative LRU cache. Based on this theoretical framework, we present an effective and stable algorithm for tiling size selection.**

***Index Terms*—Loop Tiling, Cache, Matrix Multiplication, LLVM**

**ALGORITHM 1**
Matrix Multiplication

```
$X \leftarrow n \times n$ matrix
$Y \leftarrow n \times n$ matrix
$Z \leftarrow n \times n$ matrix
Function MatrixMult()
for $i = 1$ to $N$ do
    for $k = 1$ to $N$ do
        $r = X[i][k]$
        for $j = 1$ to $N$ do
            $Z[i][j]$ += $r \cdot Y[k][j]$
        end for
    end for
end for
```

## I. Introduction

Due to the gap between processor and memory speed in modern computer architecture, achieving good performance requires optimal cache utilization. Modern compilers have focused on improving locality within caches to achieve good performance. Within this, cache tiling is a popular optimization, focused on improving the reuse of data. Cache tiling enables optimal cache reuse of data, without changing the underlying algorithm or increasing cache misses.

### A. Matrix Multiplication and Loop Tiling

Matrix multiplication is a computationally intensive task with widespread applications in diverse fields, including physics, computer graphics, and most notably machine learning. The pseudo-code for matrix multiplication is shown in **Algorithm 1**.

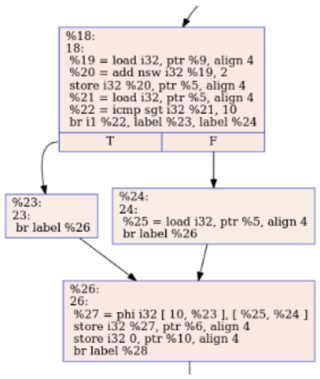
One of the key challenges in optimizing matrix multiplication lies in efficiently utilizing the memory hierarchy, especially the cache subsystem, to enhance data locality and reduce memory access latency. Cache tiling, as a prominent optimization technique, involves dividing the input matrices into smaller blocks or tiles that fit into the cache (refer to **Algorithm 2**). By performing matrix multiplication on these smaller tiles, cache tiling exploits spatial and temporal locality, minimizing cache misses and improving overall computational efficiency. The effectiveness of cache tiling is influenced by factors such as tile size, data layout, and the specific characteristics of the target hardware architecture. Ongoing research in this area aims to develop sophisticated cache tiling strategies that adapt to diverse matrix sizes, optimize for various cache architectures, and integrate seamlessly with parallel computing paradigms. This research is vital for advancing the performance of matrix multiplication algorithms and, by extension, improving the efficiency of a wide range of computational applications. For the sake of this paper, ”tiles” and ”blocks” (and their derivatives) are interchangeable.

**Algorithm 2** is the typical implementation of tilied matrix multiplication. The issue with this implementation is the overhead introduced by array boundary checks i.e., min($kk+T_k-1$, $N$) and min($jj+T_j-1$, $N$). The former occurs $\lceil \frac{N}{T_k} \rceil$ times, and the latter occurs $\lceil \frac{N}{T_j} \rceil$ times. This overhead includes multiple additional arithmetic instructions, comparison instructions, and branching instructions, which can be detrimental to the algorithm's performance with small tiles and large matrices (refer to Fig. 1). We verified the generation of extra instructions by implementing **Algorithm 2** in CPP and compiling using Clang with the `-O0` flag.

We plan to address this issue in our report and provide an implementation of our solution using LLVM.

**ALGORITHM 2**
Tiled Matrix Multiplication

```
$X \leftarrow n \times n$ matrix
$Y \leftarrow n \times n$ matrix
$Z \leftarrow n \times n$ matrix
$T_k \leftarrow$ height of tile
$T_j \leftarrow$ width of tile
**Function** TiledMatrixMult()
**for** $kk = 1$ **to** $N$ by $T_k$ **do**
  **for** $jj = 1$ **to** $N$ by $T_j$ **do**
    **for** $i = 1$ **to** $N$ **do**
      **for** $k = 1$ **to** min($kk + T_k - 1$, $N$) **do**
        $r = X[i][k]$
        **for** $j = 1$ **to** min($jj + T_j - 1$, $N$) **do**
          $Z[i][j]$ += $r \cdot Y[k][j]$
        **end for**
      **end for**
    **end for**
  **end for**
**end for**
```



```
%18:
18:
 %19 = load i32, ptr %9, align 4
 %20 = add nsw i32 %19, 2
 store i32 %20, ptr %5, align 4
 %21 = load i32, ptr %5, align 4
 %22 = icmp sgt i32 %21, 10
 br i1 %22, label %23, label %24
```

```
%23:
23:
 br label %26
```

```
%24:
24:
 %25 = load i32, ptr %5, align 4
 br label %26
```

```
%26:
26:
 %27 = phi i32 [ 10, %23 ], [ %25, %24 ]
 store i32 %27, ptr %6, align 4
 store i32 0, ptr %10, align 4
 br label %28
```

Fig. 1: LLVM IR for min($kk + T_k - 1$, $N$)

## B. Modern Cache Characteristics

In the context of modern cache architecture, there are standard parameters that define size, associativity, and cache line sizes. The cache sizes in modern CPUs can vary significantly depending on the level of the cache and the specific CPU model. The Level 1 (L1) cache, which is closest to the CPU core, is typically smaller - around 32KB to 64KB for data. The Level 2 (L2) cache usually ranges from 256KB to 512KB per pair of cores. The Level 3 (L3) cache, which is shared among all CPU cores, is much larger and can range from 2MB to 30MB, or even up to 60MB in high-end server processors. The associativity of these caches, which determines how many spots in the cache a piece of data can be placed, is common in configurations like 4-way or 8-way set associative. Finally, the cache line sizes, which denote the block of data transferred on a cache miss, are typically 64 bytes in modern systems.

To narrow the scope of this report, our primary focus will be on fully associative L1 data caches. In such a cache memory configuration, every data block has the potential to fit within any cache line. This approach allows us to reasonably match the associativity of modern processors while still simplifying the analysis and understanding of our algorithms and their respective performances. For consistency in our analysis and to match real-world systems, our paper utilizes a cache line size of 64 bytes. We aim to develop another algorithm that will optimize the use of this cache line size.

## C. Measuring Cache Performance using Cachegrind

Cachegrind is a powerful tool within the Valgrind suite of debugging and profiling utilities. Serving the role of a cache profiler, Cachegrind simulates how a target program interacts with a computer's cache architecture. It achieves this through emulating the behavior of the program with respect to an abstract machine that closely replicates primary CPU caches (L1) and last-level caches (L2 or L3, depending on your computer architecture). This allows the simulation of specific cache configurations and their corresponding behavior even without the concrete implementation of such configurations in a physical environment. Therefore, Cachegrind's detailed reporting capability pinpoints inefficient cache usage and potential areas for optimization, enabling developers to adjust code for optimal cache efficiency. We will use Cachegrind to measure the cache performance of various tiling algorithms.

## D. Types of Cache Misses

With regards to cache misses, there are three types of misses that we are concerned with:

- **Compulsory misses** occur when a cache line is referenced for the first time. These are typically unavoidable.
- **Capacity misses** occur when a program's working set size exceeds the cache size, specifically in a fully associative least recently used (LRU) cache, when data is displaced before reuse and causes a miss on the next data reference.
- **Interference misses** occur when a cache line holding reusable data gets replaced due to the cache's replacement policy assigning different data to the same location, even though there is enough space for all reusable data, differentiating them from capacity misses. There are two types of interference misses:
  - *Self-interference misses* result when an element of the same array causes the interference miss.
  - *Cross-interference misses* result when an element of a different array causes the interference miss.

Our algorithms aims to avoid as many capacity and interference misses as possible.

## II. Related Work

### A. The Cache Performance and Optimizations of Blocked Algorithms

Lam *et al.* present data on cache performance for tiled matrix multiplication and propose a model to assess cache interference [1]. Their research finds that selecting a static tile sizes, which consumes a constant fraction of the cache, underperforms when compared to tile sizes specifically adapted to the problem's needs and cache size. They proposed an algorithm that determines the maximum square tile size that minimizes self-interference misses while taking the matrix size into consideration. Nevertheless, their research focused primarily on direct mapped caches, which have different characteristics to caches having a certain degree of associativity. Moreover, square tiles tend to utilize a small portion of the cache due to their dimension restraints.

### *B. Tile Size Selection Using Cache Organization and Data Layout*

Coleman *et al.* put forward a novel model for quantifying cache misses, embracing tile sizes derived from cache size and cache line size, specifically tailored for a direct-mapped cache [2]. Their proposed algorithm targets the elimination of capacity and self-interference misses, while also aiming to lessen cross-interference misses. However, they acknowledge that the attainment of optimal performance is not a constant, particularly at higher set associativity levels.

### *C. A Quantitative Analysis of Tile Size Selection Algorithms*

Hsu *et al.* delves into the efficacy of loop tiling in memory performance optimization, underscoring the critical nature of proper tile size selection [3]. It highlights how tiling algorithms can benefit from the utilization of array padding, presenting it as a vital data layout optimization strategy that significantly boosts the effectiveness and consistency of loop tiling techniques. However, array padding adds significant overhead, especially during matrix multiplication. This overhead comes from the extra computations and iterations that arise due to extraneous array entries.

## III. Tiling Matrix Multiplication Pass and Baseline Algorithms

In our original design, we implemented tiled matrix multiplication to improve cache utilization. The idea behind this algorithm has been explained in **Algorithm 2**. After implementing this algorithm, we discovered that we could reduce the instruction overhead of the original tiling algorithm by introducing a new algorithm which only searches factors of the matrix size, thereby removing the requirement of bound checking. To find the optimal tile size in this scenario, we developed a cost function that factors different information about the cache size to decide upon the most optimal tile size. Our algorithm can produce rectangular tile shapes as well as square tile shapes.

### *A. Tiled Pass Implementations*

To experiment with different tiling algorithms, we implemented a LLVM pass that converts **Algorithm 1** to **Algorithm 2**. We focused our investigation on cache performance by enforcing that the input program adheres to a certain form i.e., our pass assumes we have **Algorithm 1**. We first obtain information about the current for-loops used within matrix multiplication. In the original algorithm, the outer for-loop uses index $i$ and iterates through every element from 1 to $N$. Outside of the index $i$ for-loop, we must introduce two new for-loops. One with index $kk$ and another with index $jj$. Both of these for-loops go through the matrix $N$ as a whole, but are incremented by a value based on the tile block size. This tile block sized is determined by specific algorithms. Within the tiled algorithm, we have similar for-loop for $k$ and $j$. These loops iterate from one end of the tile's width/height to the other end of the tile's width/height up until the edges of the matrix. In addition, we created a second pass that is similar to the first, but removed the bounds checking. We use this pass to implement one of our novel algorithms (Restrictive Tile Size Selection).

### *B. Implementing Baseline Tile Size Selection Algorithms*

We implemented two baseline tile selection algorithms. First, we tested the logical choice of taking the square root of the cache size, and running our tiled matrix multiplication against this. This is the naive tile size as it makes the logical assumption that a block of square root of cache size $\times$ square root of cache size will be able to effectively fit within the cache. Second, we looked at the algorithm from Lam *et al.*, which found a optimal tile size factoring in the interference misses and how to find them [1]. The idea behind this algorithm is to check to see if any of the addresses within the current block size will result in a conflict miss. In the case that a self-interference miss will occur, this algorithm will stop the blocking size, and choose that tile size as the optimal tiled size.

## IV. New Tiling Matrix Multiplication

We present two novel tile selection algorithms. One algorithm is based on reducing instruction overhead, and the other is based on cache line sizes and L1 miss latency.

### *A. Factor Based Tiled Size Selection*

We found that the original tiled matrix multiplication increases instruction overhead by doing an array boundary check as mentioned in the *Introduction*. However, we can remove this bound checking by forcing the tile size to always be a factor of the matrix size. By adding this restriction, we are able to remove the bound checking completely, while still maintaining correctness in our code. Therefore, we implemented a novel tile selection algorithm that generates tiles that fit within our matrices. This algorithm will make use of the second pass mentioned in *III.A*. Appendices A.1 and A.2 demonstrate how our new algorithm eliminates the need for bound-checking instructions, thereby reducing the instruction overhead associated with tiled matrix multiplication.

Taking into account this new tiling restriction, we can recognize our baseline algorithms do not work. This is because to reduce instruction overhead, we must remove bound checking. Selecting a tile size equal to the square root of the cache size can result in a segmentation fault due to out-of-bounds indexing. The tile size chosen by the Lam *et al.* algorithm also has the possibility to throw a similar segmentation fault. Due to these restrictions, we developed our own algorithm to generate appropriate tile sizes. Our algorithm is described in **Algorithm 3**.

**ALGORITHM 3**
Restrictive Tile Size Selection

```
$N \leftarrow$ Matrix Size
$F \leftarrow$ factors of $N$
$CLS \leftarrow$ Cache Line Size
$CS \leftarrow$ Cache Size
$DTS \leftarrow$ Data Type Size
**Function** TileSelection()
Initialize the empty vector $tiles$
**for** $i = 1$ **to** $len(F)$ **do**
    **for** $i = 1$ **to** $len(F)$ **do**
        $cost = \lceil \frac{DTS \cdot (F[i] \cdot F[j] + F[i])}{CLS} \rceil \cdot CLS + CLS$
        **if** $cost < CS$ **then**
            $rowSize$ = $F[i]$
            $colSize$ = $F[j]$
            $CIR = \frac{2 \cdot F[i] + F[j]}{F[i] \cdot F[j]}$
            $Penalty = \lceil \frac{F[i] \cdot DTS}{CLS} \rceil \cdot CLS - (F[i] \cdot DTS)$
            tiles.ADD($\{rowSize, colSize, CIR, Penalty\}$)
        **end if**
    **end for**
**end for**
**reverse sort** $tiles$ by penalty first, then by CIR on ties
**return** $rowSize$ and $colSize$ of greatest element in $tiles$
```

In this algorithm, our primary focus is on utilizing factors of the original matrix size. This constraint ensures that the tiled matrix multiplication stays within bounds while iterating through a tile. Next, we create an array which holds a combination of all factor sizes for rows and columns. (i.e. if the factor array includes (1, 2, 4) then we test the following combinations (1, 1), (1, 2), (1, 4), (2, 1), (2, 2), (2, 4), (4, 1), (4, 2), (4, 4)). For each of these combinations, we calculate three key metrics: the cost, CIR (Cross Interference Rate), and the penalty.

The cost function represents the sizing of the block size and is inspired by the Coleman *et al.*'s cache model [2]. Suppose we have 3 $n \times n$ matrices called X, Y, Z ($XY = Z$), and a tile of size $F[i] \times F[j]$. As shown in **Algorithm 3**, the equation to compute cost is:

$$cost = \lceil \frac{DTS \cdot (F[i] \cdot F[j] + F[i])}{CLS} \rceil \cdot CLS + CLS \qquad (1)$$

We get the function from the following reasoning: for each iteration of the outermost loop in tiled matrix multiplication, the program access $F[i]$ elements of Z and $F[i] \cdot F[j]$ elements of Y. Each element accessed takes up $DTS$ bytes, so get the total number of bytes the program accessed by multiplying the total number of accesses by $DTS$. The equation rounds this up to the nearest cache line size. This is because when the CPU brings data into the cache, it will always bring in a number of bytes corresponding to its cache line size i.e., the CPU will sometimes bring in unnecessary data. Next, although the loop allocates a register to store elements from X, the CPU still needs to access this data and bring it into the cache. This corresponds to another cache line that we append to the formula.

After calculating the cost, we compare it to the cache size. Any cost that is greater than the cache size means that it won't fit within the cache, therefore leading to the probability of a expensive and ineffective tile size. All such tile sizes are discarded.

For any cost that is under the cache size, we will calculate two other values: the cross interference rate ($CIR$) and the penalty. We utilize the equation for cross interference rate from Coleman *et al.*:

$$CIR = \frac{2 \cdot F[i] + F[j]}{F[i] \cdot F[j]} \qquad (2)$$

This equation first calculates the worse-case cross-interference misses: $2 \cdot F[i] + F[j]$. This equation naturally follows from the following line of reasoning. When the cache is entirely occupied by array Y, every subsequent request for an element from array Z forces a previously stored element from Y out of the cache cross-interference miss. When the system needs the same element of Z again, it will cause another cross-interference miss because the cache is likely filled with Y's elements again. There are potentially $2 \cdot F[i]$ cross-interferences between Z and Y. Furthermore, since accessing X can potentially evict elements of Y from the cache, we have to consider another $F[j]$ cross interference misses. Finally, to calculate the number of cross-interference misses per element of Y in a single tile, we divide by $F[i] \cdot F[j]$, providing us the cross-interfernece miss rate.

Understanding the relationship between data size and cache line size is crucial when considering how data fits within the cache. The larger the disparity from the cache line size, the more lines will need to be loaded into the processor, leading to inefficient use of cache space. Consequently, it's vital to consider the tile size's correlation with the cache line size as an approach towards minimizing cache wastage. By employing a penalty function, we can assess the efficiency with which a row (with a length defined as $rowSize$) fits within a cache line. The algorithm first calculates how much data the CPU brings into the cache via cache lines. We determine the number of bytes used by rounding the amount of space a tile row uses (data type size $\times$ number of elements) to the nearest multiple of the cache line size. Then, we can subtract the current tile row size in bytes from the total space used to find the penalty of using this cache size. The larger then penalty, the "worse" the tile is to choose. Thus, we have this equation:

$$Penalty = \lceil \frac{F[i] \cdot DTS}{CLS} \rceil \cdot CLS - (F[i] \cdot DTS) \qquad (3)$$

Finally, our algorithm reverse sorts (greatest to least) the information based firstly on the penalty cost, and then in the case of ties, reverse sort on the cross-interference rate. We sorted on penalty first and not on cross-interference rate so that our top tile sizes would best adhere to our cache line size. If we initially sorted on cross interference misses, extremely small tiles would be strictly prioritized over other better options, which would not utilize an optimal amount of cache space. Therefore, we found that the optimal tiling size comes from sorting by the penalty initially, then by the cross-interference rate.

### *B. LRU-based Tiled Size Selection*

We will shift our focus to a different algorithm. In this section, instead of looking at fitting our tiles into our matrices, our primary focus revolves around accesses to fully associative L1 data caches employing the LRU eviction strategy. The assumption of fully associativity implies that cache lines are only evicted when there is no available entry for a new cache line, simplifying our analysis and enabling us to devise a LRU-based approach to quantify the total number of cache misses and determine the optimal tiling size.

This section introduces a novel algorithm for selecting the LRU-based tiling size, outlined in **Algorithm 4**. The key concept behind this algorithm is to calculate the number of cache access misses for three matrices (X, Y, and Z) involved in the multiplication process and minimize this count. Consider a matrix multiplication scenario with three matrices (X, Y, Z) denoted as $XY = Z$. As outlined by Lam *et al.* [1], the reuse factors for matrices Y, Z, and X in the I, K, and J loops are denoted as N, Tk, and TJ, respectively. The objective is to minimize the number of cache misses by ensuring optimal reuse of these matrices. The algorithm begins by eliminating self-interference and cross-interference on the access to the tiled block of matrix Y in each I loop, resulting in a reuse factor of N for matrix Y.

The algorithm comprises two nested loops. The first loop iterates over all possible tiling block heights (TK), while the second loop traverses different tiling block widths (TJ). The width is selected as a multiple of the cache line size to facilitate clear calculations and reduce compile time.

We initiate the process by determining the necessary cache entries to accommodate the tiling block of matrix Y (TK * TJ) and the required elements from matrices X (TK) and Z (TJ) for each iteration of the I loop. The count is given by the expression $\lceil \frac{TK}{EIL} \rceil + \frac{TJ}{EIL} + \frac{TJ}{EIL} \cdot TK$. To facilitate the reuse of the tiling block of matrix Y in the subsequent I loop iteration, it is imperative to ensure that the block remains in the cache and is not displaced by newly read cache lines from matrices X and Z. Subsequently, we compute the additional cache entries needed to retain the tiling block in the cache. In each following I loop iteration, only TK elements from matrix X and TJ elements from matrix Z are read from memory, resulting in $\frac{TJ}{EIL} + \frac{TJ}{EIL} \cdot EIL$ cache line reads. The term $\frac{TJ}{EIL}$ corresponds to the access to matrix Z, while $\frac{TJ}{EIL} \cdot EIL$ pertains to matrix X. Given the utilization of the LRU eviction strategy, the first cache line to be evicted originates from matrix Y, which is undesirable. Moreover, considering the reuse of the tiling block of matrix Y in each iteration, we allocate additional entries as a buffer to store newly added lines and await the reuse of elements from matrix Y. Specifically, $\frac{TJ}{EIL} \cdot EIL$ represents the number of entries needed to designate the first cache lines of matrix X used in the last iteration as the LRU one. By combining the entries required to store all elements essential for a single I loop iteration and the entries serving as a buffer, we arrive at the required minimum number of entries:

**ALGORITHM 4**
LRU-based Tile Size Selection

```
$N \leftarrow$ Matrix Size
$CS \leftarrow$ Cache Size (Number of Cache Lines)
$DTS \leftarrow$ Data Type Size
$CLS \leftarrow$ Cache Line Size
$EIL \leftarrow$ Element Number In One Cache Line
$A \leftarrow$ Tile Height
$B \leftarrow$ Tile Width
$m \leftarrow$ Misaligned Parameter
**Function** LRUTileSelection()
miss = 1000000
EIL = $\frac{CLS}{DTS}$
**for** $a = 1$ **to** $N$ **do**
    **for** $b = EIL$ **to** $N$ by $EIL$ **do**
        $cost\_iloop = \lceil \frac{a}{EIL} \rceil + \frac{b}{EIL} + \frac{b}{EIL} \cdot a + \frac{b}{EIL} + b$
        **if** $cost\_iloop \cdot m < CS$ **then**
            $\text{miss\_x} = N \cdot \lceil \frac{N}{a} \rceil \cdot \lceil \frac{a}{EIL} \rceil \cdot \lceil \frac{N}{b} \rceil$
            $\text{miss\_z} = N \cdot \lceil \frac{N}{b} \rceil \cdot \lceil \frac{b}{EIL} \rceil \cdot \lceil \frac{N}{a} \rceil$
            **if** miss $>$ (miss_x $+$ miss_z) **then**
                miss $=$ miss_x $+$ miss_z
                $A = a$
                $B = b$
            **end if**
        **end if**
    **end for**
**end for**
**return** $A, B$
```

$$cost\_iloop = \lceil \frac{TK}{EIL} \rceil + \frac{TJ}{EIL} + \frac{TJ}{EIL} \cdot TK + \frac{TJ}{EIL} + TJ \quad (4)$$

This value needs to be constrained by our cache size. The first if statement serves as a check to ensure that it is smaller than the cache size. Additionally, we introduce a misalignment parameter (m) in this context, recognizing that we cannot guarantee perfect alignment of all matrix reads to cache lines. Inevitably, there may be misalignment leading to increased utilization of cache space. The misalignment parameter represents a trade-off; a smaller m (e.g., m = 1) provides optimal cache utilization but may result in poorer performance in the presence of memory misalignment. On the other hand, as m increases, the overall cache utilization rate tends to decrease, but it introduces redundancy to account for misalignment issues. In the final step, we compute the total number of memory reads throughout the entire matrix multiplication process. Given that we have ensured the maximum reuse factor of N for matrix Y, we exclude further consideration of matrix Y to streamline the compile-time calculations. Referring to the analysis of the tiled matrix multiplication function, we readily determine the total number of cache line reads for matrix X ($N \cdot \lceil \frac{N}{TK} \rceil \cdot \lceil \frac{TK}{EIL} \rceil \cdot \lceil \frac{N}{TJ} \rceil$) and matrix Z ($N \cdot \lceil \frac{N}{TJ} \rceil \cdot \lceil \frac{TJ}{EIL} \rceil \cdot \lceil \frac{N}{TK} \rceil$). Within the innermost if statement, we compare the previous minimum number of memory reads with the current one, ultimately selecting the tile height and width that minimizes cache line misses. The miss function reaffirms the findings from prior research, indicating that larger tile widths(TJ) and heights(TK) are essential for reducing the number of cache misses on matrix X and matrix Z respectively. In general, the optimal choice involves selecting the maximized tile block size under the cache size constraint, ensuring the attainment of the maximum reuse factor for matrix Y.

## V. Evaluation

Since we are simulating our cache using cachegrind and running our program on top of that, we cannot rely on program execution times nor the builtin C `clock()` function to measure performance. In general, cachegrind execution times are much longer compared to native executions. Instead, we will be using L1 data cache misses provided by cachegrind to measure performance. We run each algorithm and compare the number of cache misses we have. We ran all algorithms over a search space of N = 25 to 500. The two tile passes, test cases, and analysis scripts can be found at https://github.com/LarWong/EECS583-FinalProject.

## VI. Results and Analysis

In this section, the results of our experimentation and analysis are described.

### *A. Benchmarks*

All algorithms were compared against the original matrix multiplication algorithm. A Python script was created to generate test cases for any matrix size, on which we could apply our pass. For our results, we swept a data range of values for $N = 25$ to $N = 500$. Then we plotted L1 data cache misses against the original matrix multiplication algorithm. By sweeping a large range of $N$ values, we hoped to be able to cover a wide range of test cases, and obtain results that accurately describe the effectiveness of our algorithm

### *B. Results*

Fig. 2 shows the difference in cache misses between the untiled matrix multiplication and two tiling sizes when the cache is sized at 32KB. Choosing a tile size of the square root of the cache size gives subpar performance, even performing worse than the untiled algorithm at higher values of $N$. Furthermore, we calculated the average in cache misses. By first applying:

$$\frac{\text{Untiled Misses} - \text{Algorithm Misses}}{\text{Untiled Misses}} \cdot 100\% \tag{5}$$

to each $N$ and taking the average across all N. On average, this algorithm has 3.13% more cache misses compared to normal matrix multiplication using 32KB caches. However, on the other hand, Lam *et al.*'s algorithm [1] performs substantially better than the untiled algorithm, and also significantly outperforms choosing the square root of the cache size. On average, this tile size has 75.7% fewer cache misses compared to normal matrix multiplication using 32KB caches. In Fig. 3 the 3 algorithms ran on a cache size of 64KB. We can recognize that the untiled matrix multiplication and the square root strategy both perform about equally again. However, the gap between Lam *et al.* [1] has increased substantially. Fig. 2 demonstrates the Lam *et al.* optimization for finding a optimal tile size significantly outperforms simply taking the square root, and that a larger cache size widens this gap. On average, the square root strategy has 0.52% more cache misses and Lam *et al.* has 76.52% fewer cache misses using 64KB caches.

Fig. 5 compares cache misses of **Algorithm 3** and untiled matrix multiplication on a 32KB cache size. Due to the nature of **Algorithm 3**, and only choosing factors of $N$ (except $N$ itself) for tiling size, any prime number will use a tile size of 1. This leads to massive increase in cache misses for any matrix size that is a prime number. However, as prime numbers are not commonly used for matrix multiplication sizes, we can remove those values from the sweep. After removing prime numbers, Fig. 4 shows the reduction in cache misses. In this figure, we can clearly see the cache miss reduction of **Algorithm 3**. We achieved 58.56% fewer in cache misses compared to untiled matrix multiplication. In the case of a cache size of 64KB, as demonstrated in Fig. 6, this increased to 61.08% fewer cache misses. Overall, our algorithm has successfully been able to decrease the number of cache misses and reduce the instruction overhead of our code.

In our experiments with **Algorithm 4**, we varied the cache sizes (32KB and 64KB) while keeping the misalignment parameter at 1.3. The results are depicted in Fig. 7 and Fig. 8 respectively. The evaluation compares the number of L1 data cache misses between tiled matrix multiplication using **Algorithm 4** and untiled matrix multiplication.

We observed a notable reduction in cache misses with Algorithm 4, achieving an average reduction of 84.37% and 80.75% for cache sizes of 32KB and 64KB, respectively. Tiled multiplication tends to perform less optimally when N is small, as the entire matrix can be easily accommodated in the cache. **Algorithm 4** exhibits improved performance with larger matrix sizes (N), showcasing a potential reduction in data cache misses of up to 96%. The reduction rate tends to increase with a larger cache size, but we observe a better reduction rate with a 32KB cache size. This phenomenon is attributed to the fact that there are more scenarios where the 64KB cache size can accommodate the entire matrix. Furthermore, excluding small matrix sizes (N smaller than 90), **Algorithm 4** performs better with a 64KB cache size, achieving an average reduction rate exceeding 90% for both cache sizes. In summary, **Algorithm 4** demonstrates efficiency and stability in reducing cache misses, making it a promising approach for enhancing performance in matrix multiplication.

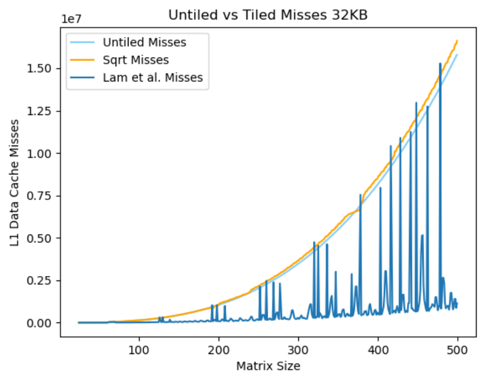


Fig. 2: Untiled vs Tiled matrix multiplication Cache Misses on a 32KB Cache Size

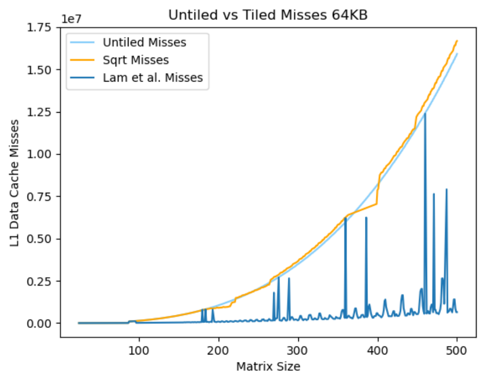


Fig. 3: Untiled vs Tiled matrix multiplication Cache Misses on a 64KB Cache Size

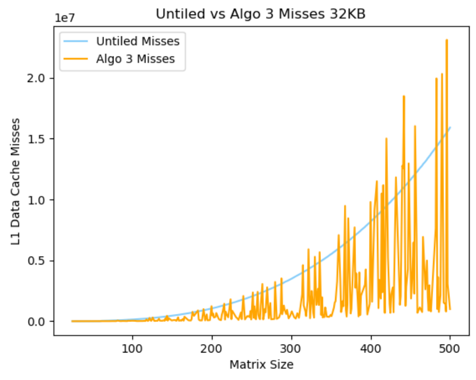


Fig. 4: Untiled vs Algorithm 3 matrix multiplication cache misses on a 32KB Cache Size ignoring prime numbers

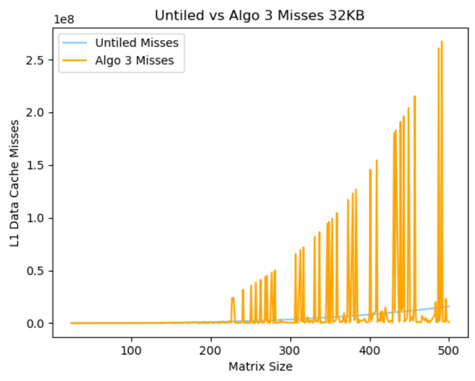


Fig. 5: Untiled vs Algorithm 3 matrix multiplication cache misses on a 32KB Cache Size

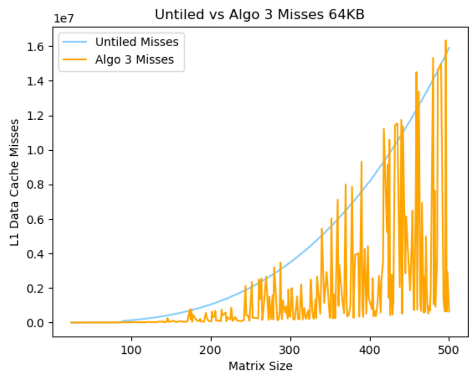


Fig. 6: Untiled vs Algorithm 3 matrix multiplication cache misses on a 64KB Cache Size ignoring prime numbers

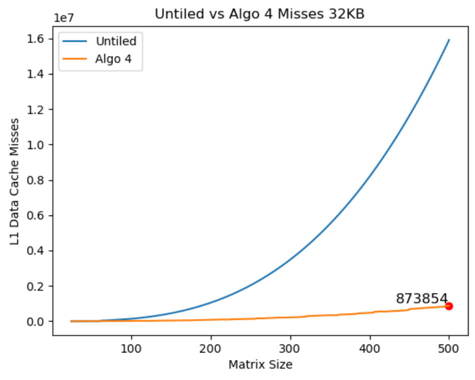


Fig. 7: Untiled vs Algorithm 4 matrix multiplication cache misses on a 32KB Cache Size

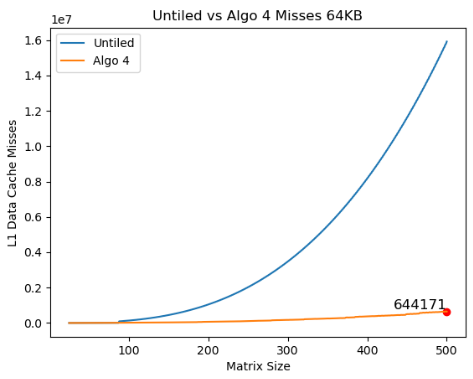


Fig. 8: Untiled vs Algorithm 4 matrix multiplication cache misses on a 64KB Cache Size

## VII. Conclusion and Future Work

Our proposed Tiled Matrix Multiplication conversion has been effectively integrated into LLVM. In addition, we have also successfully implemented four distinct algorithms - one naive (using taking the square root of cache size), one from a previous paper, and two unique ones. The accuracy of the pass was tested by comparing the results of matrix multiplication pre- and post-algorithm implementation. Notable improvements were seen after reviewing cachegrind metrics and comparing L1 data cache misses. Our findings suggest that our novel tile sizes provide a significant boost over the original, non-tiled matrix multiplication.

In our forthcoming research and experimentation, we intend to broaden the scope of these algorithms to be applicable for various cache configurations. This means we will adapt them to suit different sizes, associativity aspects, and line sizes. An additional aspect of our future work involves the development of a detection pass. The function of this tool would be to pinpoint specific locations within arbitrary pieces of code where we could possibly implement tiling. This approach will further enhance the performance and versatility of our algorithms in a multitude of applications.

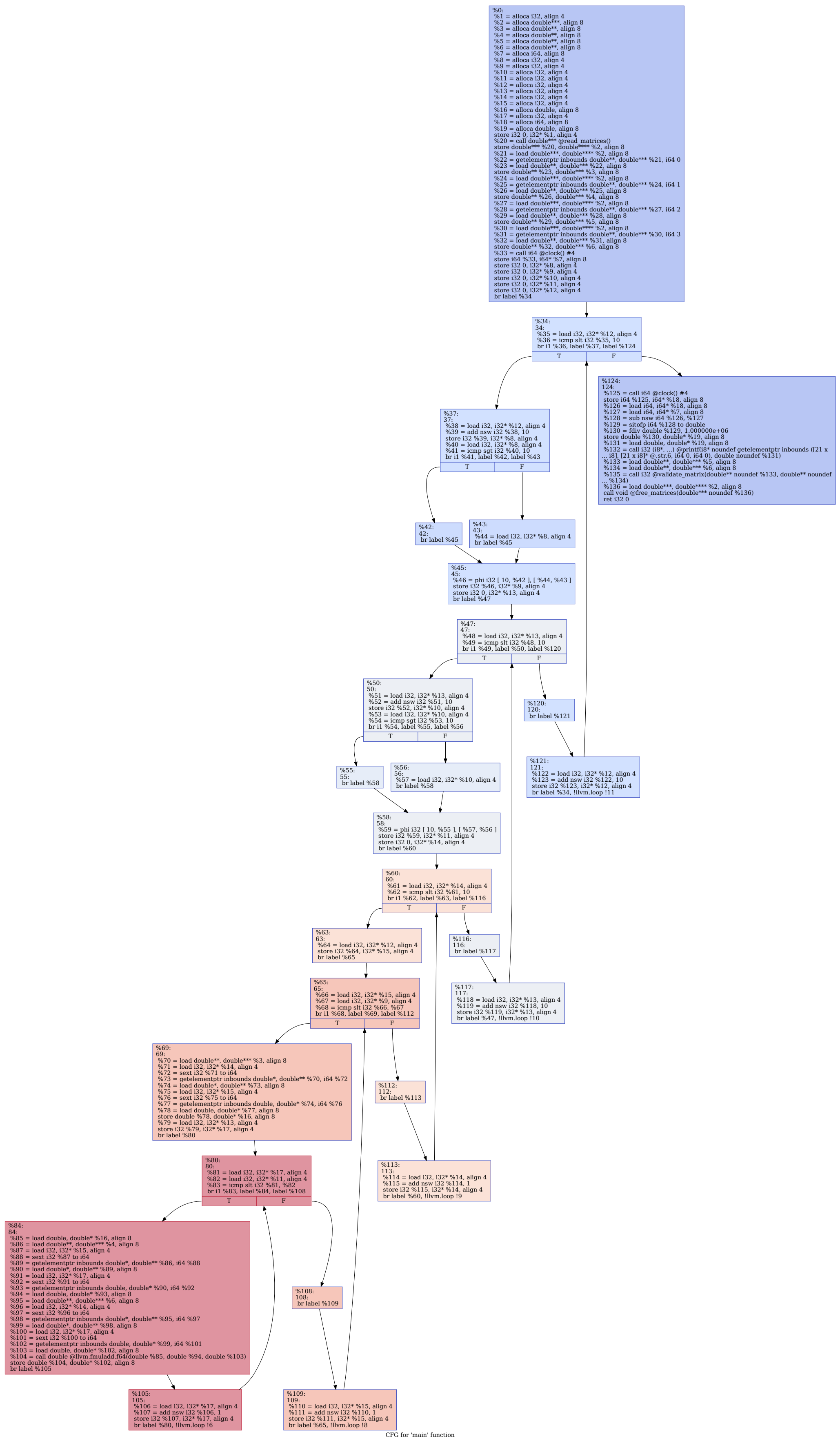
## References

[1] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. 1991. The cache performance and optimizations of blocked algorithms. SIGPLAN Not. 26, 4 (Apr. 1991), 63–74. https://doi.org/10.1145/106973.106981

[2] Stephanie Coleman and Kathryn S. McKinley. 1995. Tile size selection using cache organization and data layout. In Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation (PLDI '95). Association for Computing Machinery, New York, NY, USA, 279–290. https://doi.org/10.1145/207110.207162

[3] Chung-hsing Hsu and Ulrich Kremer. 2004. A Quantitative Analysis of Tile Size Selection Algorithms. J. Supercomput. 27, 3 (March 2004), 279–294. https://doi.org/10.1023/B:SUPE.0000011388.54204.8e

## Appendix A.1



```
%0:
 %1 = alloca i32, align 4
 %2 = alloca double***, align 8
 %3 = alloca double**, align 8
 %4 = alloca double**, align 8
 %5 = alloca double**, align 8
 %6 = alloca double**, align 8
 %7 = alloca i64, align 8
 %8 = alloca i32, align 4
 %9 = alloca i32, align 4
 %10 = alloca i32, align 4
 %11 = alloca i32, align 4
 %12 = alloca i32, align 4
 %13 = alloca i32, align 4
 %14 = alloca i32, align 4
 %15 = alloca i32, align 4
 %16 = alloca double, align 8
 %17 = alloca i32, align 4
 %18 = alloca i64, align 8
 %19 = alloca double, align 8
 store i32 0, i32* %1, align 4
 %20 = call double*** @read_matrices()
 store double*** %20, double**** %2, align 8
 %21 = load double***, double**** %2, align 8
 %22 = getelementptr inbounds double**, double*** %21, i64 0
 %23 = load double**, double*** %22, align 8
 store double** %23, double*** %3, align 8
 %24 = load double***, double**** %2, align 8
 %25 = getelementptr inbounds double**, double*** %24, i64 1
 %26 = load double**, double*** %25, align 8
 store double** %26, double*** %4, align 8
 %27 = load double***, double**** %2, align 8
 %28 = getelementptr inbounds double**, double*** %27, i64 2
 %29 = load double**, double*** %28, align 8
 store double** %29, double*** %5, align 8
 %30 = load double***, double**** %2, align 8
 %31 = getelementptr inbounds double**, double*** %30, i64 3
 %32 = load double**, double*** %31, align 8
 store double** %32, double*** %6, align 8
 %33 = call i64 @clock() #4
 store i64 %33, i64* %7, align 8
 store i32 0, i32* %8, align 4
 store i32 0, i32* %9, align 4
 store i32 0, i32* %10, align 4
 store i32 0, i32* %11, align 4
 store i32 0, i32* %12, align 4
 br label %34
```

```
%34:
34:
 %35 = load i32, i32* %12, align 4
 %36 = icmp slt i32 %35, 10
 br i1 %36, label %37, label %124
```

```
%124:
124:
 %125 = call i64 @clock() #4
 store i64 %125, i64* %18, align 8
 %126 = load i64, i64* %18, align 8
 %127 = load i64, i64* %7, align 8
 %128 = sub nsw i64 %126, %127
 %129 = sitofp i64 %128 to double
 %130 = fdiv double %129, 1.000000e+06
 store double %130, double* %19, align 8
 %131 = load double, double* %19, align 8
 %132 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([21 x
... i8], [21 x i8]* @.str.6, i64 0, i64 0), double noundef %131)
 %133 = load double**, double*** %5, align 8
 %134 = load double**, double*** %6, align 8
 %135 = call i32 @validate_matrix(double** noundef %133, double** noundef
... %134)
 %136 = load double***, double**** %2, align 8
 call void @free_matrices(double*** noundef %136)
 ret i32 0
```

```
%37:
37:
 %38 = load i32, i32* %12, align 4
 %39 = add nsw i32 %38, 10
 store i32 %39, i32* %8, align 4
 %40 = load i32, i32* %8, align 4
 %41 = icmp sgt i32 %40, 10
 br i1 %41, label %42, label %43
```

```
%42:
42:
 br label %45
```

```
%43:
43:
 %44 = load i32, i32* %8, align 4
 br label %45
```

```
%45:
45:
 %46 = phi i32 [ 10, %42 ], [ %44, %43 ]
 store i32 %46, i32* %9, align 4
 store i32 0, i32* %13, align 4
 br label %47
```

```
%47:
47:
 %48 = load i32, i32* %13, align 4
 %49 = icmp slt i32 %48, 10
 br i1 %49, label %50, label %120
```

```
%50:
50:
 %51 = load i32, i32* %13, align 4
 %52 = add nsw i32 %51, 10
 store i32 %52, i32* %10, align 4
 %53 = load i32, i32* %10, align 4
 %54 = icmp sgt i32 %53, 10
 br i1 %54, label %55, label %56
```

```
%120:
120:
 br label %121
```

```
%121:
121:
 %122 = load i32, i32* %12, align 4
 %123 = add nsw i32 %122, 10
 store i32 %123, i32* %12, align 4
 br label %34, !llvm.loop !11
```

```
%55:
55:
 br label %58
```

```
%56:
56:
 %57 = load i32, i32* %10, align 4
 br label %58
```

```
%58:
58:
 %59 = phi i32 [ 10, %55 ], [ %57, %56 ]
 store i32 %59, i32* %11, align 4
 store i32 0, i32* %14, align 4
 br label %60
```

```
%60:
60:
 %61 = load i32, i32* %14, align 4
 %62 = icmp slt i32 %61, 10
 br i1 %62, label %63, label %116
```

```
%63:
63:
 %64 = load i32, i32* %12, align 4
 store i32 %64, i32* %15, align 4
 br label %65
```

```
%116:
116:
 br label %117
```

```
%65:
65:
 %66 = load i32, i32* %15, align 4
 %67 = load i32, i32* %9, align 4
 %68 = icmp slt i32 %66, %67
 br i1 %68, label %69, label %112
```

```
%117:
117:
 %118 = load i32, i32* %13, align 4
 %119 = add nsw i32 %118, 10
 store i32 %119, i32* %13, align 4
 br label %47, !llvm.loop !10
```

```
%69:
69:
 %70 = load double**, double*** %3, align 8
 %71 = load i32, i32* %14, align 4
 %72 = sext i32 %71 to i64
 %73 = getelementptr inbounds double*, double** %70, i64 %72
 %74 = load double*, double** %73, align 8
 %75 = load i32, i32* %15, align 4
 %76 = sext i32 %75 to i64
 %77 = getelementptr inbounds double, double* %74, i64 %76
 %78 = load double, double* %77, align 8
 store double %78, double* %16, align 8
 %79 = load i32, i32* %13, align 4
 store i32 %79, i32* %17, align 4
 br label %80
```

```
%112:
112:
 br label %113
```

```
%80:
80:
 %81 = load i32, i32* %17, align 4
 %82 = load i32, i32* %11, align 4
 %83 = icmp slt i32 %81, %82
 br i1 %83, label %84, label %108
```

```
%113:
113:
 %114 = load i32, i32* %14, align 4
 %115 = add nsw i32 %114, 1
 store i32 %115, i32* %14, align 4
 br label %60, !llvm.loop !9
```

```
%84:
84:
 %85 = load double, double* %16, align 8
 %86 = load double**, double*** %4, align 8
 %87 = load i32, i32* %15, align 4
 %88 = sext i32 %87 to i64
 %89 = getelementptr inbounds double*, double** %86, i64 %88
 %90 = load double*, double** %89, align 8
 %91 = load i32, i32* %17, align 4
 %92 = sext i32 %91 to i64
 %93 = getelementptr inbounds double, double* %90, i64 %92
 %94 = load double, double* %93, align 8
 %95 = load double**, double*** %6, align 8
 %96 = load i32, i32* %14, align 4
 %97 = sext i32 %96 to i64
 %98 = getelementptr inbounds double*, double** %95, i64 %97
 %99 = load double*, double** %98, align 8
 %100 = load i32, i32* %17, align 4
 %101 = sext i32 %100 to i64
 %102 = getelementptr inbounds double, double* %99, i64 %101
 %103 = load double, double* %102, align 8
 %104 = call double @llvm.fmuladd.f64(double %85, double %94, double %103)
 store double %104, double* %102, align 8
 br label %105
```

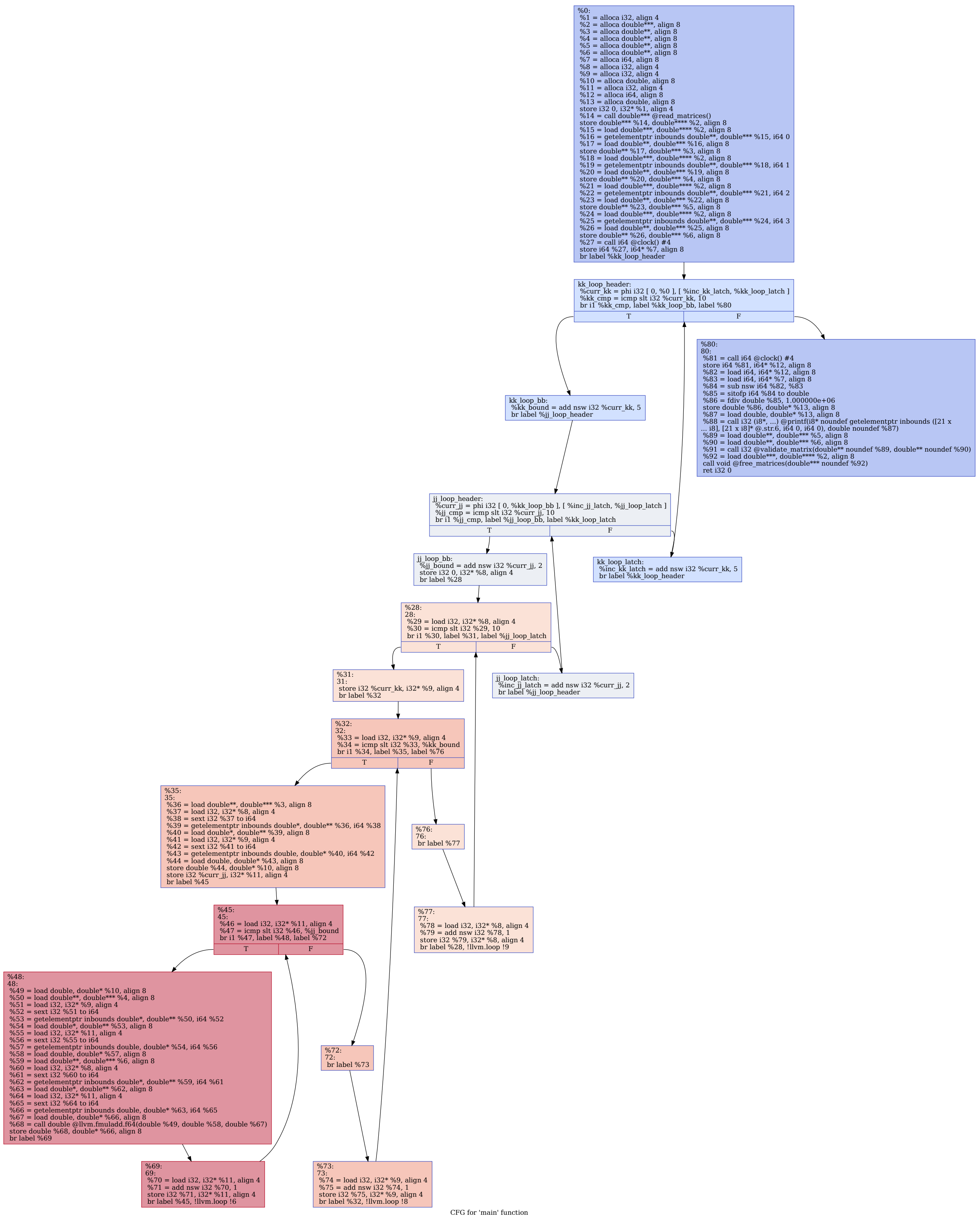
```
%108:
108:
 br label %109
```

```
%105:
105:
 %106 = load i32, i32* %17, align 4
 %107 = add nsw i32 %106, 1
 store i32 %107, i32* %17, align 4
 br label %80, !llvm.loop !6
```

```
%109:
109:
 %110 = load i32, i32* %15, align 4
 %111 = add nsw i32 %110, 1
 store i32 %111, i32* %15, align 4
 br label %65, !llvm.loop !8
```

Fig. 9: CFG illustrating a typical implementation of tiled matrix multiplication compiled with Clang. We employ instructions for bounds checking in BBs (%37, %42, %43, %45) and in BBs (%50, %55, %56, %58).

## Appendix A.2



```
%0:
 %1 = alloca i32, align 4
 %2 = alloca double***, align 8
 %3 = alloca double**, align 8
 %4 = alloca double**, align 8
 %5 = alloca double**, align 8
 %6 = alloca double**, align 8
 %7 = alloca i64, align 8
 %8 = alloca i32, align 4
 %9 = alloca i32, align 4
 %10 = alloca double, align 8
 %11 = alloca i32, align 4
 %12 = alloca i64, align 8
 %13 = alloca double, align 8
 store i32 0, i32* %1, align 4
 %14 = call double*** @read_matrices()
 store double*** %14, double**** %2, align 8
 %15 = load double***, double**** %2, align 8
 %16 = getelementptr inbounds double**, double*** %15, i64 0
 %17 = load double**, double*** %16, align 8
 store double** %17, double*** %3, align 8
 %18 = load double***, double**** %2, align 8
 %19 = getelementptr inbounds double**, double*** %18, i64 1
 %20 = load double**, double*** %19, align 8
 store double** %20, double*** %4, align 8
 %21 = load double***, double**** %2, align 8
 %22 = getelementptr inbounds double**, double*** %21, i64 2
 %23 = load double**, double*** %22, align 8
 store double** %23, double*** %5, align 8
 %24 = load double***, double**** %2, align 8
 %25 = getelementptr inbounds double**, double*** %24, i64 3
 %26 = load double**, double*** %25, align 8
 store double** %26, double*** %6, align 8
 %27 = call i64 @clock() #4
 store i64 %27, i64* %7, align 8
 br label %kk_loop_header
```

```
kk_loop_header:
 %curr_kk = phi i32 [ 0, %0 ], [ %inc_kk_latch, %kk_loop_latch ]
 %kk_cmp = icmp slt i32 %curr_kk, 10
 br i1 %kk_cmp, label %kk_loop_bb, label %80
```

```
%80:
80:
 %81 = call i64 @clock() #4
 store i64 %81, i64* %12, align 8
 %82 = load i64, i64* %12, align 8
 %83 = load i64, i64* %7, align 8
 %84 = sub nsw i64 %82, %83
 %85 = sitofp i64 %84 to double
 %86 = fdiv double %85, 1.000000e+06
 store double %86, double* %13, align 8
 %87 = load double, double* %13, align 8
 %88 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([21 x
... i8], [21 x i8]* @.str.6, i64 0, i64 0), double noundef %87)
 %89 = load double**, double*** %5, align 8
 %90 = load double**, double*** %6, align 8
 %91 = call i32 @validate_matrix(double** noundef %89, double** noundef %90)
 %92 = load double***, double**** %2, align 8
 call void @free_matrices(double*** noundef %92)
 ret i32 0
```

```
kk_loop_bb:
 %kk_bound = add nsw i32 %curr_kk, 5
 br label %jj_loop_header
```

```
jj_loop_header:
 %curr_jj = phi i32 [ 0, %kk_loop_bb ], [ %inc_jj_latch, %jj_loop_latch ]
 %jj_cmp = icmp slt i32 %curr_jj, 10
 br i1 %jj_cmp, label %jj_loop_bb, label %kk_loop_latch
```

```
jj_loop_bb:
 %jj_bound = add nsw i32 %curr_jj, 2
 store i32 0, i32* %8, align 4
 br label %28
```

```
kk_loop_latch:
 %inc_kk_latch = add nsw i32 %curr_kk, 5
 br label %kk_loop_header
```

```
%28:
28:
 %29 = load i32, i32* %8, align 4
 %30 = icmp slt i32 %29, 10
 br i1 %30, label %31, label %jj_loop_latch
```

```
%31:
31:
 store i32 %curr_kk, i32* %9, align 4
 br label %32
```

```
jj_loop_latch:
 %inc_jj_latch = add nsw i32 %curr_jj, 2
 br label %jj_loop_header
```

```
%32:
32:
 %33 = load i32, i32* %9, align 4
 %34 = icmp slt i32 %33, %kk_bound
 br i1 %34, label %35, label %76
```

```
%35:
35:
 %36 = load double**, double*** %3, align 8
 %37 = load i32, i32* %8, align 4
 %38 = sext i32 %37 to i64
 %39 = getelementptr inbounds double*, double** %36, i64 %38
 %40 = load double*, double** %39, align 8
 %41 = load i32, i32* %9, align 4
 %42 = sext i32 %41 to i64
 %43 = getelementptr inbounds double, double* %40, i64 %42
 %44 = load double, double* %43, align 8
 store double %44, double* %10, align 8
 store i32 %curr_jj, i32* %11, align 4
 br label %45
```

```
%76:
76:
 br label %77
```

```
%45:
45:
 %46 = load i32, i32* %11, align 4
 %47 = icmp slt i32 %46, %jj_bound
 br i1 %47, label %48, label %72
```

```
%77:
77:
 %78 = load i32, i32* %8, align 4
 %79 = add nsw i32 %78, 1
 store i32 %79, i32* %8, align 4
 br label %28, !llvm.loop !9
```

```
%48:
48:
 %49 = load double, double* %10, align 8
 %50 = load double**, double*** %4, align 8
 %51 = load i32, i32* %9, align 4
 %52 = sext i32 %51 to i64
 %53 = getelementptr inbounds double*, double** %50, i64 %52
 %54 = load double*, double** %53, align 8
 %55 = load i32, i32* %11, align 4
 %56 = sext i32 %55 to i64
 %57 = getelementptr inbounds double, double* %54, i64 %56
 %58 = load double, double* %57, align 8
 %59 = load double**, double*** %6, align 8
 %60 = load i32, i32* %8, align 4
 %61 = sext i32 %60 to i64
 %62 = getelementptr inbounds double*, double** %59, i64 %61
 %63 = load double*, double** %62, align 8
 %64 = load i32, i32* %11, align 4
 %65 = sext i32 %64 to i64
 %66 = getelementptr inbounds double, double* %63, i64 %65
 %67 = load double, double* %66, align 8
 %68 = call double @llvm.fmuladd.f64(double %49, double %58, double %67)
 store double %68, double* %66, align 8
 br label %69
```

```
%72:
72:
 br label %73
```

```
%69:
69:
 %70 = load i32, i32* %11, align 4
 %71 = add nsw i32 %70, 1
 store i32 %71, i32* %11, align 4
 br label %45, !llvm.loop !6
```

```
%73:
73:
 %74 = load i32, i32* %9, align 4
 %75 = add nsw i32 %74, 1
 store i32 %75, i32* %9, align 4
 br label %32, !llvm.loop !8
```

Fig. 10: CFG illustrating our implementation of tiled matrix multiplication. We no longer have instructions for bounds checking.