CS5220 HW1 Report: DGEMM Optimizations

Group 8: Ji Kim (jyk46)

1. Introduction

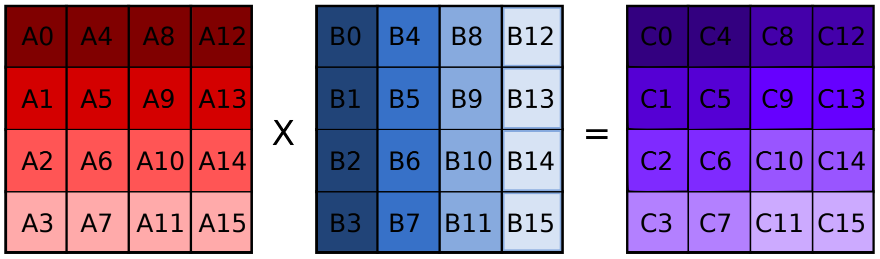


Figure 1: Double-Precision General Matrix-Matrix Multiply – For simplicity, 4x4 matrices are shown with elements labeled in the column-major order. Rows of input matrix `A` is multiplied with columns of input matrix `B` and accumulated to generate the output matrix `C`.

In this report, we explore several optimizations for improving the performance of double-precision general matrix-matrix multiply (DGEMM). The DGEMM algorithm, as shown in Figure 1, is a standard matrix multiplication between two input matrices with double-precision floating point elements to generate an output matrix. The serial algorithm computes elements of the output matrix `C` from input matrices `A` and `B` as follows:

$$C_{ij} = \sum_{k=0}^{N} A_{ik} * B_{kj}$$

All optimizations are implemented on top of the provided blocking algorithm. We compare the performance of the optimizations against this baseline. The blocking algorithm splits the matrices into and operates on smaller blocks that are more likely to fit in the cache to better exploit temporal and spatial locality. The high-level goals of the optimizations explored in this assignment are to maximize hardware resource utilization, produce more desirable memory access patterns, ameliorate negative cache effects, and improve static analysis in the compiler.

Specifically, the following optimizations are explored in this report:

- Vectorizing with SIMD extensions to maximize hardware resource utilization;
- Changing the loop ordering to produce more desirable memory access patterns;
- Copying blocks into a scratchpad in contiguous memory to avoid conflict misses in the cache;
- Experimenting with C keywords and attributes to better express the intent of the code to the compiler;
- Utilizing profile-guided optimization to maximize the benefits of static analysis in the compiler;

The proprietary Intel cross-compiler (i.e., icc) was used for all experiments as the generated code outperforms gcc by a significant factor for DGEMM.

2. Vectorizing with SIMD Extensions

2.1. Reason for Optimization

Many modern processors utilize a separate SIMD pipeline capable of operating on wide (e.g., 256b, 512b) vector registers to achieve parallel computation of multiple data elements. Special vector instructions in the ISA can be used to schedule work onto these SIMD pipelines. In the case of the Intel Xeon E5 2600-series installed on the totient compute nodes, these vector instructions are a part of the Advanced Vector eXtensions (AVX) of the x86 ISA.

One reason why effectively utilizing the SIMD pipeline is so important is because it allows us to exploit data-level parallelism (DLP: same task on different data elements) for workloads with regular control flow and memory access patterns, such as the matrix multiplication in this assignment. Explicitly, SIMD in this context allows us to:

- amortize the overhead of fetching, decoding, and issuing the instruction across multiple data elements;
- apply computation to multiple data elements in parallel;
- and coalesce multiple scalar memory accesses into a single vector memory access.

Another reason is that most of the resources for high-throughput floating-point computation is in the SIMD pipeline and leaving these resources idle would mean we are already limiting our ideal performance to a fraction of the system's potential. For example, each core in the Xeon E5 has a 256b SIMD pipeline capable of supporting 8 floating-point operations in parallel (assuming fused multiply-adds), whereas the scalar pipeline only has 2 FPU units each capable of supporting a single floating-point operation.

Although modern compilers have optimization passes to identify and generate vector instructions automatically, this capability can be limited compared to manually manually inserting vector *intrinsics*.

2.2. Details of Optimization

As such, the goal of this optimization is to effectively utilize the SIMD pipeline with AVX instructions. In order to do this, the first step is to design a vectorization strategy for matrix multiplication. Figure 2 outlines the vectorization strategy used for this optimization.

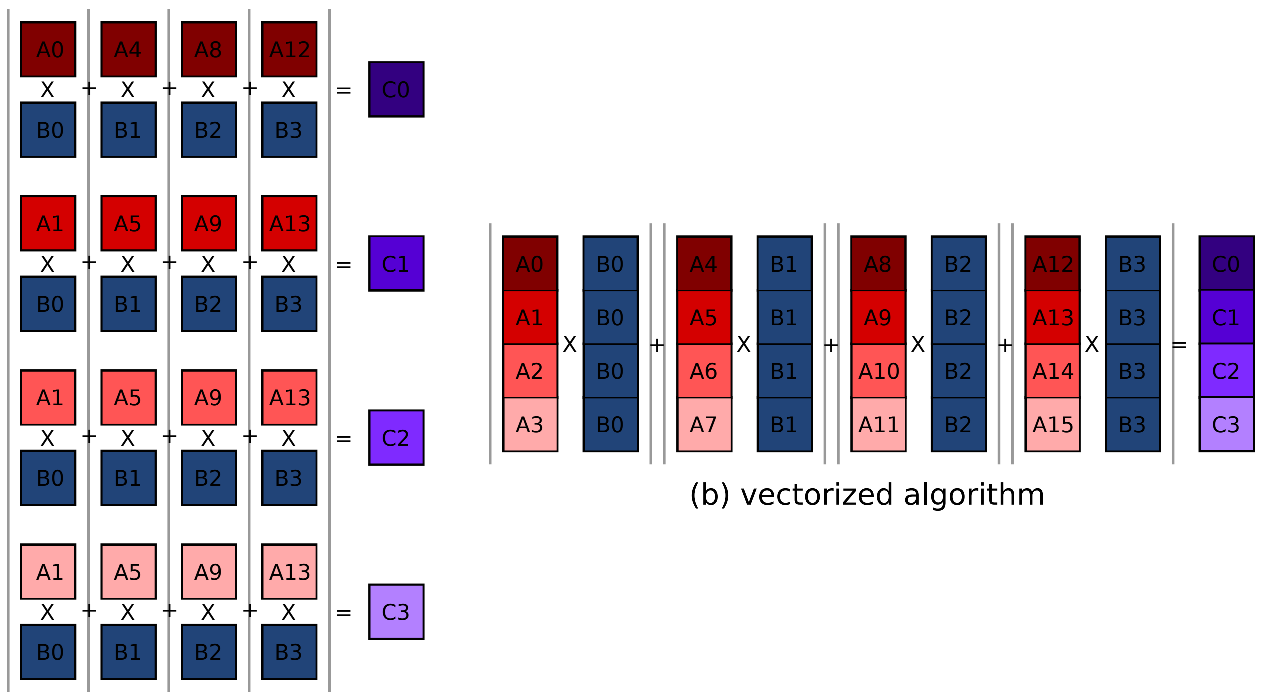
If we consider the computations required to generate column `j` of matrix `C`, we can see that all elements in column `k` of matrix `A` need to be multiplied by B_{kj}. With this insight, we can write the equation for computing a vector of elements in column `j` of matrix `C` as follows:

$$\{C_{0j}, ..., C_{Nj}\} = \sum_{k=0}^{N} \{A_{0k}, ..., A_{Nk}\} * B_{kj}$$

By using vector registers to represent multiple elements of a given column, we are able to vectorize computation across multiple elements within a column of the output matrix.

The pseudo-assembly for computing one column of a block in the output matrix is shown below:

```
    load.v  vr1, 0(c_addr) // vector load {C_0j,...,C_Nj} (partial product)
_loop:
    load.v  vr2, 0(a_addr) // vector load {A_0k,...,A_3k}
    load    r2,  0(b_addr) // load B_kj
    set.v   vr3, r2        // broadcast B_kj to vector register
    mul.v   vr4, vr2, vr3
    add.v   vr1, vr4, vr1
    addi    a_addr, inc    // pointer bump for A
    addi    b_addr, inc    // pointer bump for B
    br      _loop          // loop for N iterations
    store.v vr1, 0(c_addr) // vector store {C_0j,...,C_Nj}
```


A0 X B0 + A4 X B1 + A8 X B2 + A12 X B3 = C0
A1 X B0 + A5 X B1 + A9 X B2 + A13 X B3 = C1
A1 X B0 + A5 X B1 + A9 X B2 + A13 X B3 = C2
A1 X B0 + A5 X B1 + A9 X B2 + A13 X B3 = C3
A0 A1 A2 A3 X B0 B0 B0 B0 + A4 A5 A6 A7 X B1 B1 B1 B1 + A8 A9 A10 A11 X B2 B2 B2 B2 + A12 A13 A14 A15 X B3 B3 B3 B3 = C0 C1 C2 C3

(a) serial algorithm

(b) vectorized algorithm

Figure 2: Overview of Vectorization Strategy – For simplicity, 4x4 matrices are shown with elements labeled in the column-major order. The computation required to generate the elements of a single column of the output matrix can be re-arranged to group multiple elements of the input matrix into a vector register. This effectively allows us to compute multiple elements of the output matrix in parallel (i.e., 4 doubles in parallel with 256b SIMD pipeline).

An alternative strategy would be to vectorize computation across multiple elements within a *row* of the output matrix. The challenge with this approach is that since we would need to load multiple elements within a row of matrices `B` and `C`, and since the matrices are column-major in memory, we would have a stride of `N` instead of unit-stride as before. Not only would this be undesirable for cache locality, it means we would be unable to utilize efficient vector loads, which can only load elements that are consecutive in memory. However, in order to evaluate the impact of regular vs. irregular memory accesses, we have submitted both strategies for this assignment. These implemementations are named `dgemm_avx_regular.c` and `dgemm_avx_irregular.c`, respectively.

In addition to vectorizing matrix multiplication, AVX also has support for fused floating-point multiply-adds (FMAs). Instead of having separate instructions for the multiply and add in the pseudo-assembly above, FMAs allow us to execute these operations as a single instruction. This essentially doubles the throughput of floating-point operations in the SIMD pipeline.

So far we have been assuming matrix dimensions that are evenly divisible by 4, but there are some non-trivial complications that arise when this is not the case. First, we cannot use standard vector memory operations because they must always access *256b-aligned addresses in memory*. We can always enforce this alignment during memory allocation for the first column of a matrix, but if the dimensions are not evenly divisible by the vector width, all subsequent columns are not guaranteed to be aligned. To address this, we use *unaligned* vector memory operations which allow accesses to non-256b-aligned addresses, but are not as efficient as the standard variants. Second, even with unaligned vector memory operations, there is still the corner case when we want to vector load the last elements in a column (i.e., the last 3 elements in a column left to compute, but we vector load in groups of 4). We do not want to modify any elements beyond the current column we are computing lest we run the risk of storing junk data to the next column. To address this, we use *masked* vector memory operations that can specify which of the 4 elements in memory should be accessed (i.e., only load the first 3 elements starting from the base address, instead of 4). However, masked vector memory operations are even less efficient than the unaligned variants.

A conservative approach would be to always calculate the mask and use the masked vector memory operations. This would be slow, but would guarantee correctness. A more aggressive approach would be to prioritize these different types of vector memory operations in the order of highest efficiency whenever possible:

- Standard: for matrix dimensions divisible by 4
- Unaligned: for any matrix dimensions, when we are *not* computing the last elements in a column
- Masked: for any matrix dimensions, when we are computing the last elements in a column

2.3. Results

We evaluate both the regular and irregular memory access variants of the vectorizing optimization with FMAs and support for any matrix dimension using masked vector memory operations. In order to quantify the benefit of manual-vectorization over auto-vectorization, we show the performance of the provided blocked implementation using the `-xAVX` compiler flag. The results of using a combination of manual- and auto-vectorization were also collected. All results are shown in Figure 3.

The regular variant showed the better speedup of 3x over the naive blocked implementation. In contrast, the irregular variant had a slowdown of roughly 2x over the same baseline, highlighting the importance of prioritizing desirable cache access patterns as well as efficient vector memory operations. In addition, we can see that auto-vectorization is significantly worse than manual-vectorization, but combining both techniques outperforms either. This makes sense, since there are still opportunities for vectorization that are not exploited with manual-vectorization that is easier for the compiler to optimize.

Since we are using doubles (64b) for data elements and the SIMD pipeline is 256b wide, we know we can fit 4 elements per vector register. In the ideal case where the matrix dimension is evenly divisible by 4, we are parallelizing computation over 4 elements of the output matrix. Coupled with the use of FMAs which double the throughput of floating-point operations in the SIMD pipeline, the ideal speedup should be 8x.

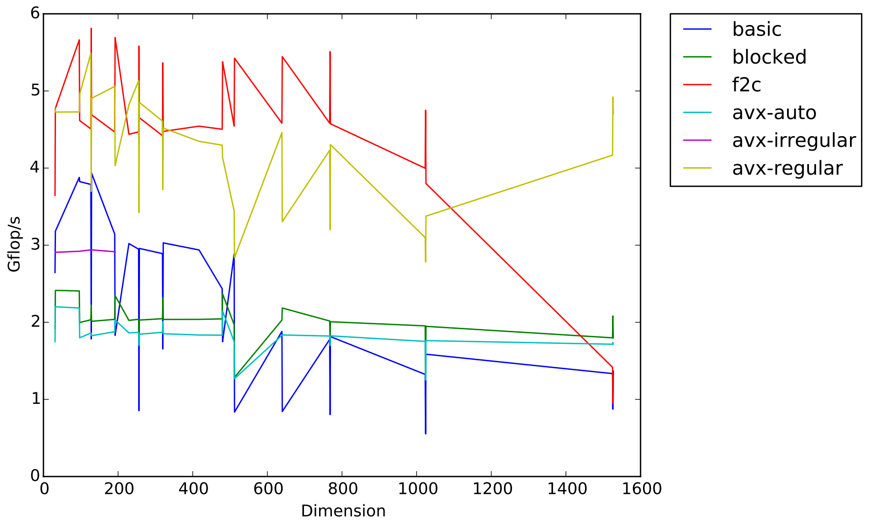


Figure 3: Performance Comparison of AVX Optimizations – Both AVX optimizations with the irregular and regular memory access patterns are shown. In addition, the naive blocking implementation with auto-vectorization is included for comparison. Combining both manual- and auto-vectorization achieves the best results. We intentionally omit the results for BLAS nad MKL implementations in order to focus on the behavior at the performance range of the optimization.

There are several factors that prevent us from reaching this limit. For one, the ideal speedup does not take into account the cache misses and thrashing that occurs in practice. There is also the overhead from copying scalar registers to vector registers, as well as the mask calculation for masked vector memory operations. The overhead from conditional blocks for prioritizing different types of vector memory operations is non-trivial as well.

3. Changing Loop Ordering

3.1. Reason for Optimization

The memory access pattern for DGEMM is largely determined by the order in which the matrices are traversed. Depending on the loop order and the arrangement of the matrices in memory (i.e., row-major vs. column-major), memory could be accessed with a unit-stride that maximizes temporal and spatial locality, or a stride equal to the matrix dimensions that is more likely to cause cache misses.

The goal of this optimization is to find the loop ordering that uses the data in a cache line as much as possible before bringing in another cache line. We explore a variety of loop orderings based on an intuitive analysis of how to achieve unit-stride memory accesses to identify the optimal loop ordering.

3.2. Details of Optimization

The naive loop ordering of `ijk` iterates across the output matrix `C`, traversing the columns before traversing the rows. Computing each element in the output matrix requires traversing the columns of `A` and traversing the rows of `B`. This type of access pattern is generalized in Figure 4.

Because the matrices are arranged in a column-major pattern in memory, it is always preferable to traverse the rows (i.e., vertically) to yield unit-stride memory accesses. Any time we traverse the columns (i.e., horizontally), we only use a single element in a cache line before accessing another cache line, which makes it more likely to have cache misses.

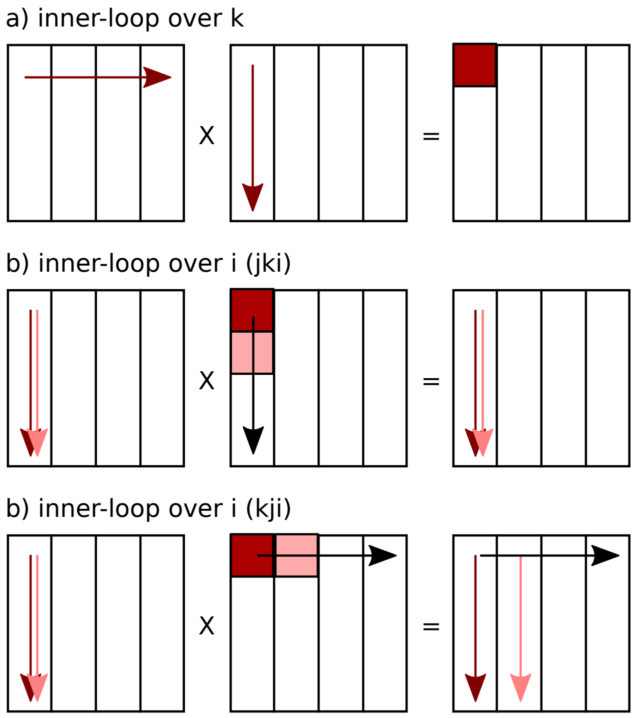


Figure 4: Memory Access Patterns of Loop Orderings – The memory access patterns for the innermost loop for `ijk`, `jki`, and `kji` orderings are shown in dark red. For the latter two orderings, the light red arrows show the change in the accesses with each iteration of the middle loop. Note that `jki` can reuse the data in the cache for both `B` and `C`, whereas `kji` can only reuse the data in the cache for `A`.

Given this, we can see that the naive loop ordering actually generates a very hostile memory access pattern since the innermost loop always traverses the columns of `A` and the middle loop traverses the columns of `C`.

Fortunately, we can leverage this insight to make an educated guess about which loop ordering would be preferable. Since the memory access pattern of the innermost loop is the most critical in exploiting temporal locality, it makes sense to iterate across `i` for the innermost loop, as this will allow us to traverse the rows of both `A` and `C`, while using a single element from `B` to calculate the partial products for an entire column of `C`. Note that if the block size is large, traversing the rows will still require accessing multiple cache lines, but accessing consecutive cache lines will help avoid conflict misses more than if we traversed the columns. This means that either the `jki` or the `kji` ordering will yield the most desirable memory access patterns. The former allows us to reuse the same data for `B` and `C` while only shifting one column over in `A`, whereas the latter only allows us to reuse the same data in `A`. See Figure 4 for visual comparisons.

3.3. Results

The results of all possible loop orderings are shown in Figure 5. We only show the effects of changing the loop ordering of the computation kernel operating on a given block. Although we did experiment with changing the loop ordering of the outer loop that traverses the blocks, the effects on performance were negligible. This was slightly contrary to expectations as the order in which the blocks are traversed should impact the reuse of data in lower-level caches (e.g., L2, L3), whereas the loop ordering of the computation kernel is more focused on L1 cache utilization.

As expected, the `jki` ordering shows the greatest improvement in performance, followed by the `kji` ordering. The other orderings either exacerbate the sub-optimal memory access patterns or have negligible improvements over the naive ordering.

One caveat with not iterating over `k` in the innermost loop is that we cannot accumulate the partial products for a given element in the output matrix in the local register storage. For example, in the `jki` ordering, we must store the partial product for all elements in a given column in the output matrix to memory, then load each one back again for the accumulation when `k` increments. However, since using the optimal loop ordering encourages reuse of data in the L1 cache, the overhead of storing/loading of partial products is not as significant as if we were using the naive loop ordering.

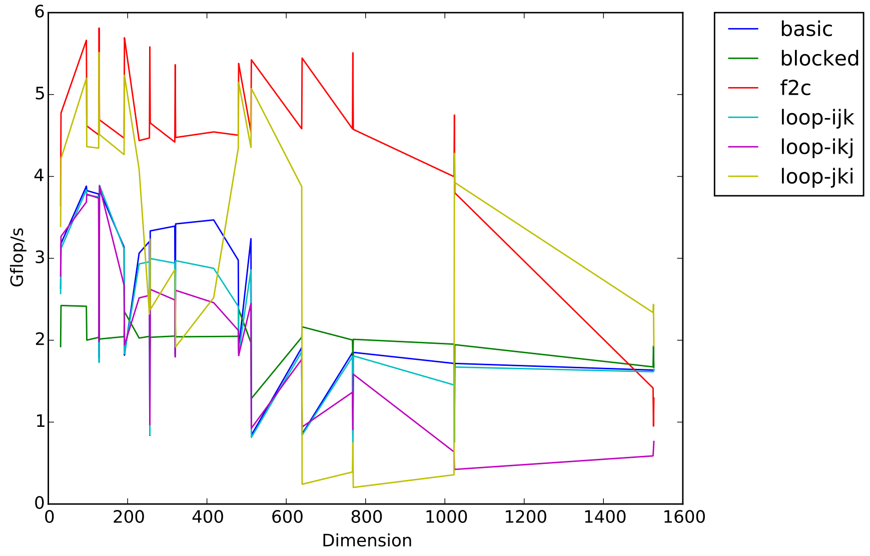


Figure 5: Performance Comparison of Loop Order Optimizations – We intentionally omit the results for BLAS nad MKL implementations in order to focus on the behavior at the performance range of the optimization.

4. Copying Blocks to Scratchpad

4.1. Reason for Optimization

Although the blocking optimization helps the working set of a single iteration of the computation kernel fit in the cache, it does not prevent conflict misses in the cache. This is because blocking only reduces the number of cache lines touched by the computation kernel, but does not change the stride of the memory access pattern when traversing the columns of a matrix. If the number of cache lines in a column is roughly a multiple of the number of sets in the cache, consecutive rows in the matrix will map to the same set. This means that if we traverse the columns of a matrix, we will only be able to use one set of the cache, causing repeated conflict misses. To makes things worse, only one element of each new cache line we bring in will be used by the computation kernel, yielding an inefficient utilization of the cache.

For example, the Intel Xeon processors on the totient compute nodes have an 8-way set-associative L1 cache with a capacity of 32KB and 64B lines. This means we have a total of $32K/64 = 512$ cache lines spread out across $512/8 = 64$ sets. Assuming the matrix is arranged in column-major order and in contiguous memory, the worst case is if the matrix dimension is a multiple of $64 * 8 = 512$ doubles (since each cache line can hold eight 64b doubles). Of course this only accounts for a single matrix being stored in the cache– in reality, cache conflicts can be aggravated with cache lines from multiple matrices being mapped to the same set.

4.2. Details of Optimization

We can alleviate such conflicts and reduce the sensitivity to matrix dimensions by copying blocks into a scratchpad such that the data within each block are contiguous in memory. This allows us to control the stride of traversing the columns within a block regardless of the matrix dimensions by setting the block dimensions such that these conflicts become rare. Furthermore, if we set the block dimensions such that all matrices fit in the L1 cache, we can get closer to fully utilizing the cache capacity without any conflicts. In addition, this copy optimization allows us to align the matrices to the SIMD width regardless of the matrix dimensions to enable vector loads/stores when vectorizing the code. A critical tradeoff to keep in mind is the computational cost of copying the matrices to the scratchpad memory. This optimization only makes sense if the benefit of reducing conflict misses in the cache outweighs the overhead of the copy itself.

Figure 6 shows how an example 4x4 matrix using 2x2 blocks are re-arranged when copying to the scratchpad memory. There are a couple approaches to when and how much data to copy. One approach is

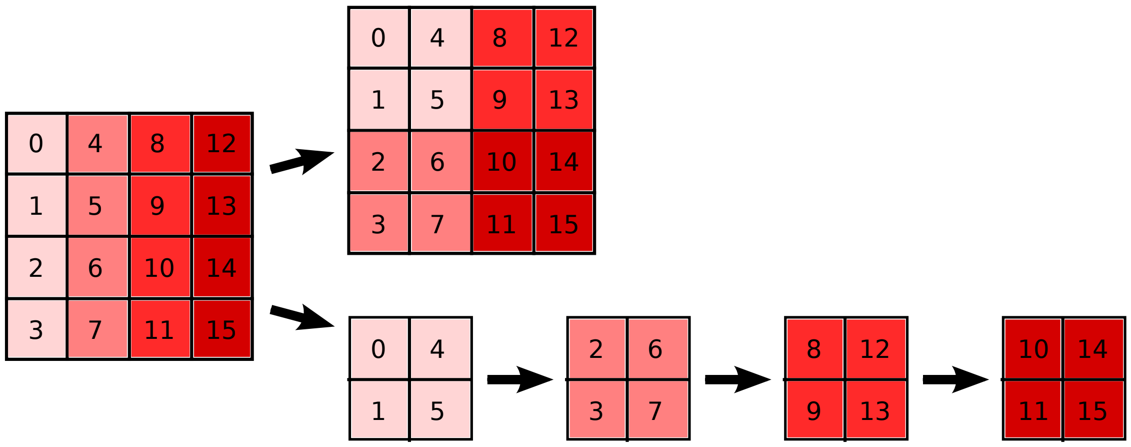


Figure 6: Two Approaches to Copy Optimization – The matrix on the left is arranged in column-major order in memory. It can be copied into a large scratchpad capable of storing all blocks (top path), or copied into a smaller scratchpad with enough space for one block (bottom path). In either case the data in each block is arranged to be contiguous in memory. The indices represent the data element in the original matrix and the colors represent contiguous segments in memory, with darker colors representing increasing addresses in memory.

to copy and re-arrange the entire matrix once before any computation, then copy the result back from the scratchpad to the output matrix in the original format after the computation is complete. This amortizes the overhead of bringing in the data from the input matrices into the cache over the entire computation, but it increases memory pressure by requiring a scratchpad that can hold all three matrices. Another approach is to only allocate enough space in the scratchpad for three blocks and copy the data in and out of the scratchpad every iteration of the computation kernel. This makes more efficient use of memory, but we need to contaminate the cache with data from the original input matrices between every call to the computation kernel.

4.3. Results

Figure 7 shows the results for both large and small scratchpad approaches to copy optimization. The interesting point here is although there are no performance improvements, the performance is much more stable across all matrix dimensions. This emphasizes the impact of conflict misses in the cache from the matrix dimensions.

A quick note on choosing block sizes. We theorize that the optimal block size is one which allows all three blocks to fit completely inside of the L1 cache. With a 32KB cache, the following equation must be satisfied:

$$3M^2 <= \frac{32K}{8} \rightarrow M <= \sqrt{\frac{32K}{8*3}} \rightarrow M <= 37$$

However, a sweep of the block sizes showed that a block size of 128 was optimal across all matrix dimensions. This likely has to do with the fact that the computation kernel requires more data from one block than the other blocks at a given iteration, so it is preferable to have a larger block size to better amortize the overhead of calling a computation kernel iteration.

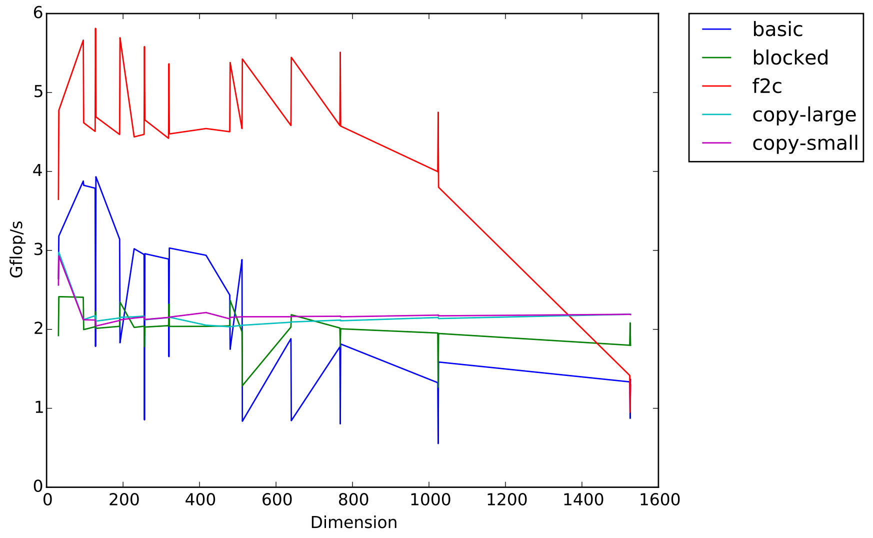


Figure 7: Performance Comparison of Copy Optimizations – We show results for a large scratchpad with a single copy of the entire matrix before/after the computation, and a small scratchpad with multiple copies of just the blocks being used for the current iteration of the computation kernel. We intentionally omit the results for BLAS and MKL implementations in order to focus on the behavior at the performance range of the optimization.

5. Experimenting with Attributes

5.1. Reason for Optimization

Oftentimes the compiler will choose to be conservative with optimization passes because there is not enough information from static analysis to determine whether or not certain optimizations are safe. If the programmer can give hints to the compiler about the intent of the code, we can force the compiler to make aggressive optimizations to generate higher performance binaries.

5.2. Details of Optimization

One significant compiler optimization is the inlining of functions. Inlined functions are not explicitly called (i.e., no jumping and linking in assembly), but appear to be embedded into point from which they are called. This eliminates the overhead of the function call, which can be especially useful when there are many arguments which need to be communicated through the stack in memory, instead allowing us to keep arguments in faster, local registers. Although function inlining is a common pass in compilers, sometimes we need to force the compiler to do so by using the `always_inline` attribute. Such C attributes can be prepended to any function declaration.

Another way to give hints to the compiler is the `restrict` keyword in C. Usually, the compiler tends be conservative about pointer optimizations if it cannot be sure if pointers alias to the same locations in memory. However, if we know that certain pointers do not alias, as in the input matrix pointers A/B/C in DGEMM, we can explicitly mark such pointers as not aliasing in memory. By doing so, we can help the compiler perform more aggressive pointer optimizations.

We can also use pragmas to mark loops that should be vectorized (i.e., `#pragma simd`), or give hints about when to issue prefetch requests to saturate the memory bandwidth (i.e., `#pragma prefetch A`).

5.3. Results

Figure 8 compares the performance of the blocked DGEMM implementation with the three previous optimizations: AVX, loop ordering, copy optimization, with and without the additional compiler hints described above.

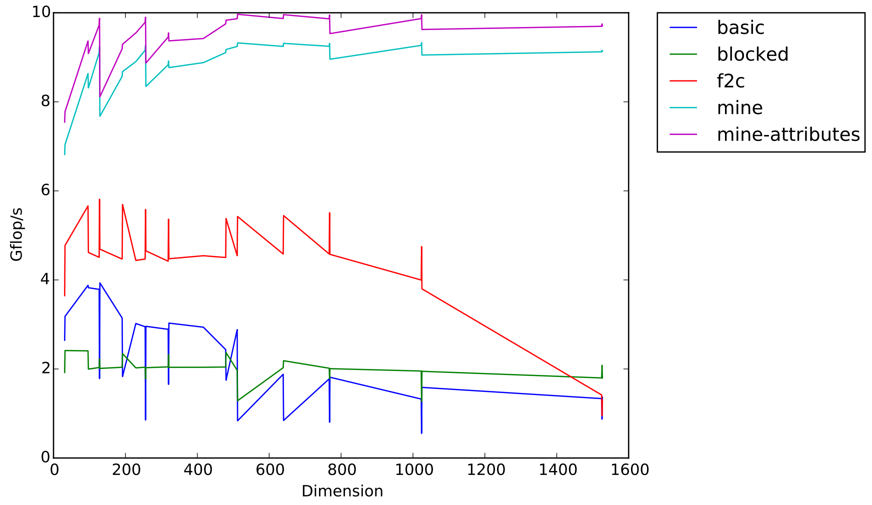


Figure 8: Performance Comparison of Attribute Optimizations – Results for the blocked implementation with the previous three optimizations (i.e., AVX, loop ordering, copy optimization) are shown as `mine`. Results form adding the compiler hints on top of this aggregative implementation are shown as `mine-attributes`. We intentionally omit the results for BLAS nad MKL implementations in order to focus on the behavior at the performance range of the optimization.

6. Utilizing Profile-Guided Optimization

6.1. Reason for Optimization

Modern compilers are quite sophisticated and provide many features that can improve the quality of the generated code. However, a general challenge with compilers is that they can only act on static analysis. This means that it cannot make any assumptions about data-dependent control flow. If the compiler could have access to dynamic information about an application, though, it could predicate branches more aggressively based on the general patterns it detects during runtime.

6.2. Details of Optimization

Fortunately, we can actually provide such dynamic information to the compiler by using profile-guided optimizations (PGO). By enabling PGO in the compiler, every run of an application dumps a .dyn file that summarizes the trends of the application during runtime. This information in turn can be used in a subsequent compilation to aggressively optimize the code.

From our experiments, it seems that the more dynamic information is collected in the form of multiple .dyn files, the more the performance improves. This makes sense since with more samples, the compiler is getting a more accurate portrayal of the average execution of the application.

6.3. Results

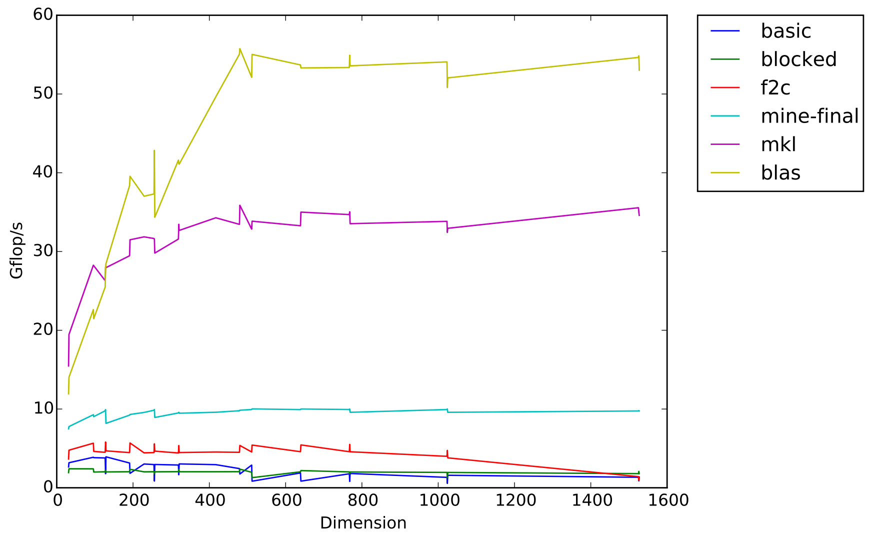


Figure 9: Performance Comparison of PGO Optimizations – Results for the final DGEMM implementation with all optimizations in this assignment, including the profile-guided optimization using data from ten unique .dyn files are shown compared against all provided implementations.

Figure 9 shows the results with PGO enabled using dynamic information collected from ten unique runs. Please see the prof/ subdirectory for all .dyn files. The culmination of all the optimizations explored in this assignment shows that we can reach over 10GFLOPS/s of performance, which is roughly a 5x speedup over the naive blocked baseline implementation.