CS5220 HW1 Report: Preliminary DGEMM Optimizations

Group 8: Ji Kim (jyk46) and Cem Iskir (ci48)

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:

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

We use a standard blocking algorithm that computes smaller pieces of the output matrices at a time as a baseline for our optimizations. The high-level goals of the optimizations are to improve the code generated by the compiler, maximize hardware resource utilization, and produce more desirable memory access patterns.

2. Proposed Optimizations

We explore three optimizations in this report:

- Exploring compiler optimizations to improve the code quality of the generated binary;
- Vectorizing with SIMD extensions to maximize hardware resource utilization;
- Changing the inner loop ordering to produce more desirable memory access patterns;

2.1. Exploring Compiler Optimizations

2.1.1. Reason for Optimization

Although modern compilers are quite sophisticated and usually generate high-quality code, often it is not as simple as using the `-O3` option in order to generate the best code that the compiler is capable of generating. In fact, sometimes being too aggressive can be have negative impacts and it is known that using `-O2` can be preferable. This highlights the fact that compilers are not all-knowing and can benefit significantly from a little nudging from the user by explicitly declaring the intent of the code and identifying opportunities for certain optimizations.

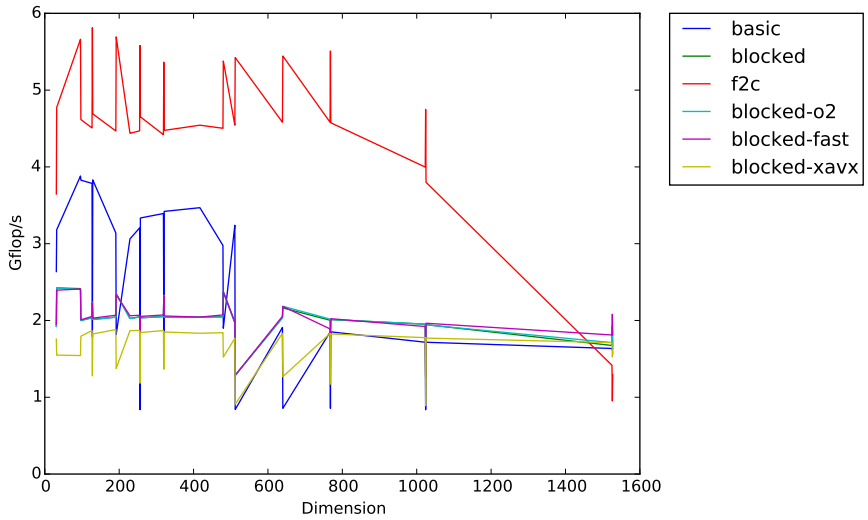

Figure 2: Performance Comparison of Compiler Optimizations – Each compiler option was applied to the naive blocked implementation of DGEMM. We intentionally omit the results for BLAS nad MKL implementations in order to focus on the behavior at the performance range of the optimization.

2.1.2. Details of Optimization

For this preliminary report, we experimented with different combinations of options to the Intel C compiler (icc). We use the naive blocked implementation as the baseline for all of the experiments.

- **-O2 –** Uses less aggressive optimizations. Namely it disables function inlining and auto-vectorization (non-AVX).
- **-fast –** Enables more aggressive floating-point operations on top of `-O3`.
- **-xAVX –** An option that attempts to vectorize the code by converting scalar instructions into vector instructions that utilize the SIMD pipeline in the processor.

2.1.3. Results

The results for using the various compiler options are shown in Figure 2. Unfortunately, none of these extra compiler options improved performance over the naive blocked implementation. In fact, the `-xAVX` option slightly decreased performance. An object dump of the binary showed that the compiler was indeed inserting vector instructions but it seems a naive auto-vectorization can actually degrade performance. The other options actually had roughly the same performance as the naive blocked implementation. One interesting point to note is that the basic implementation outperforms the blocked implementation for smaller matrix sizes. This makes sense since the overhead of communication required through memory between blocks is more noticeable for smaller matrices when there is less computation per block (i.e., lower volume-to-surface-area ratio).

2.2. Vectorizing with SIMD Extensions (AVX)

2.2.1. 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 is quite limited, and it is widely recognized that manually inserting vector *intrinsics* is the preferred method of generating vector instructions.

2.2.2. 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 3 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}
```

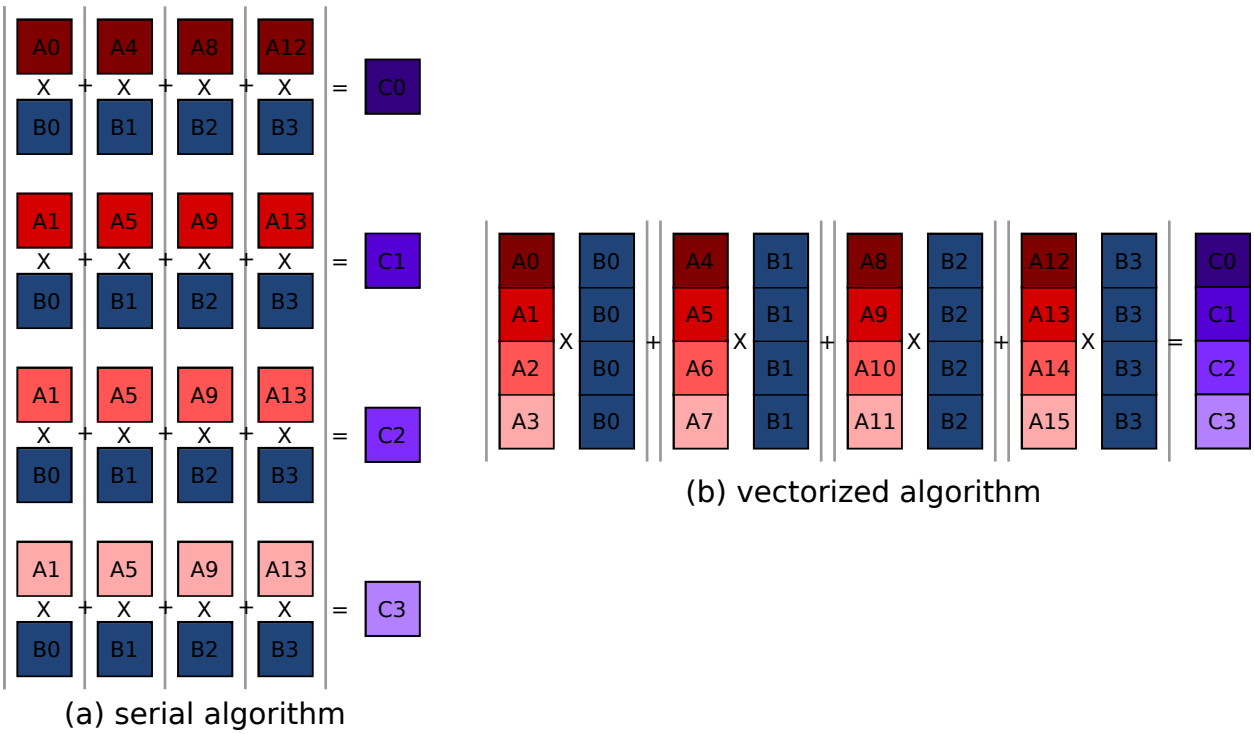

Figure 3: Overview of Vectorization Strategy – For simplicity, 4x4 matrices are shown with elements labeled in the column-major order. The computation required to generate the elements of a single column of the output matrix can be re-arranged to group multiple elements of the input matrix into a vector 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

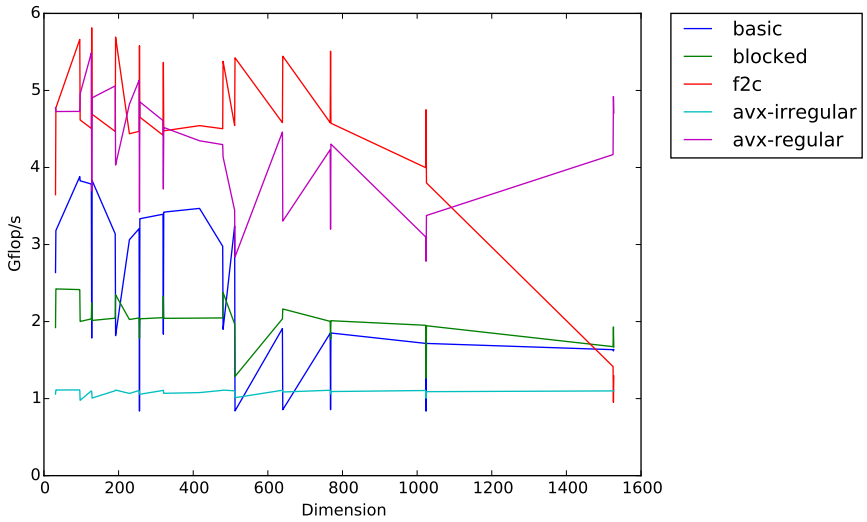

Figure 4: Performance Comparison of AVX Optimizations – Both AVX optimizations with the irregular and regular memory access patterns are shown. We intentionally omit the results for BLAS nad MKL implementations in order to focus on the behavior at the performance range of the optimization.

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

2.2.3. 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. The results are shown in Figure 4.

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.

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. 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.

2.3. Changing Inner Loop Ordering

2.3.1. Reason for Optimization

As many compilers are using loop optimization, we decided on changing the order of loops in order to improve the performance. For this optimization, we basically changed dgemm_basic.c file, Our motivation was to reorder the loops; so that we could use the memory more efficiently. We wanted to access the already accessed parts of memory repeatedly as possible.

2.3.2. Details of Optimization

In the dgemm_basic.c file we had the following code:

```
int i, j, k;
for (i = 0; i < M; ++i) {
  for (j = 0; j < M; ++j) {
    double cij = C[j*M+i];
    for (k = 0; k < M; ++k)
      cij += A[k*M+i] * B[j*M+k];
    C[j*M+i] = cij;
  }
}
```

As mentioned, we want to change the order of the loops to get the best use of memory. This way we can take disadvantage of spatial locality to reduce the overhead of data transfer from main memory, or higher level caches to lower level caches. In the algorithm above, we can see that we access the array as below:

```
A[k*M + i]
B[j*M + k]
C[j*M + i]
```

Since we are going to access the elements of A and B over and over again, our intention was to make as less memory operations on these arrays as possible. In array A we access elements which has index of k*M+i. To make use of the already loaded data, we need to put the loop which we iterate over k to outermost. Following the same idea, we should put j in outer loop. This gave us the order of:

```
for (j = 0; j < M; ++j)
  for (k = 0; k < M; ++k)
    for (i = 0; i < M; ++i)
```

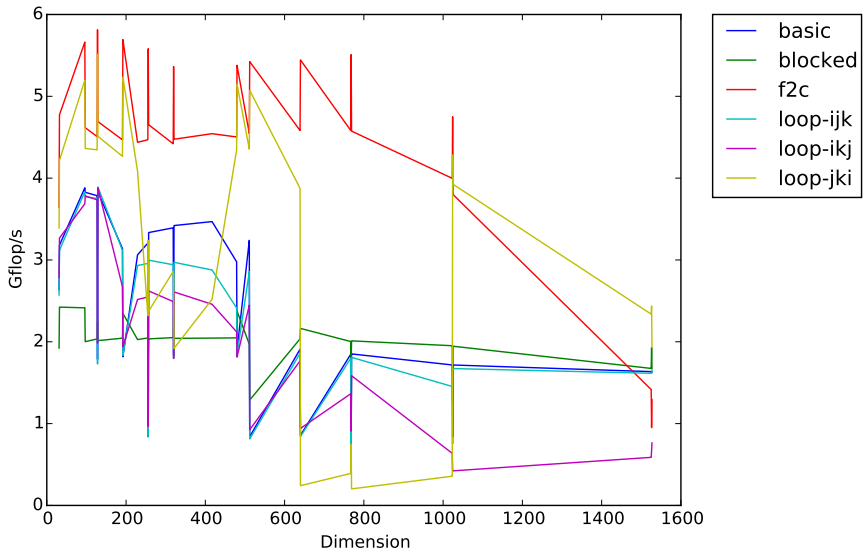

Figure 5: Performance Comparison of Loop Order Optimizations – We intentionally omit the results for BLAS nad MKL implementations in order to focus on the behavior at the performance range of the optimization.

2.3.3. Results

As it is clear from the results in Figure 5, we achieved significant performance improvement just by changing the order of loops; thus accessing memory efficiently. We also got an extra ordinary result between 700-1000 which will be inspected in further iterations.

3. Note to Reader

Our third group member decided to audit the class and we were not notified until the day this report was due. Due to this, parts of this report are not up to the level of quality we were aiming for. We hope the reader will excuse this inconsistency. Thank you.