Optimization of Matrix Multiplication

Markus Salasoo, Vikram Thapar, Yalcin Ozhabes

Our final report is divided into following attempted optimizations for enhancing the performance of double-precision generalized matrix multiplication (DGEMM).

1) Loop Ordering
2) Vectorization using AVX instructions.
3) Blocking
4) Compilation Flags

Section 1) Loop Ordering

The following basic code provided in the matmul repository makes use of `ijk` ordering to perform the matrix multiplication. The `ijk` loop ordering is not the most efficient one because of the spatial locality issues. For instance, the inner loop calls the entries of A which are M apart from each other.

```
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;
  }
}
```

So, we changed the loop ordering to `jki` where the overhead due to spatial locality is minimal. Below here, we show the code associated to `jki` loop ordering. Here, the inner loop calls the entries of A which are adjacent to each other.

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

Figure 1 shows the speed comparison between `ijk` and `jki` ordering. As expected, the `jki` ordering is significantly faster than `ijk` ordering. We also see a dip in the performance of `jki` ordering code as the matrix size goes beyond 1000 by 1000. We believe that the issue is related to memory access from cache which can be resolved using blocking.

Section 2) Vectorization using AVX instructions

Firstly, we would like to mention that we use `jki` ordering while performing matrix multiplication using AVX instructions. As AVX register can hold 4 double precision numbers, we use the strategy in which we perform 4 by 4 matrix computations. This puts the constraint that the matrix size should be divisible by 4. So, for matrices that are non-divisible by 4, we obtain the nearest size which is a multiple of 4, allocate the aligned memory in separate matrices A_k, B_k, C_k, copy matrices A and B into

the aligned matrices and pad their rest of entries with zeros. The 4 by 4 matrix computation makes use of various AVX instructions including `fmadd_pd`, which perform multiplication and addition of vectors in one instruction.

Figure 2 shows the comparison between basic and AVX. The gain in the speed is significant (about an 7-8 times faster than the basic code). But, similar to Section 1, we do see the dip in the performance as matrix size goes beyond 1000 by 1000.

Section 3) Copy Optimization and Blocking

In a matrix multiplication the number of floating point operations goes with N_3 while the memory used goes as N_2. That factor makes the cache use very important as theoretically it should be possible to make order N operations for each memory access. To approach this theoretically limit, our strategy was to use cache effectively.

We are optimizing for the Xeon processor so it is useful to look at the cache specifications for this processor. It has 3 level caching. $L3$ cache is 15 MB, $L2$ is 256 KB and $L1$ cache is 32 KB. We tried to achieve the maximum number of floating point operations without going to a higher level in the cache hierarchy by blocking the initial matrix into submatrices such that three submatrices fit into the corresponding level cache.

We calculate the maximum size of the square matrices (three of them) that fits into $L3$ cache by a straight forward calculation that is provided below:

$$3N2\times(8\ bytes/number) = 15\ MB$$

$$N \approx 800$$

We carried out similar calculations to have an idea about the sub-block sizes that we want to have within the blocks that fit $L3$. After we got the code running, we played with different block sizes and converged to the values that works the best by pure experimentation.

Blocking without the copy optimization relies only on the temporal locality only. If we work on the same block for a long time hopefully all the values will be copied to the cache at the first time we reach to them and stay there for the entire calculation of the block. This did not work as expected and we observed that we need to copy the matrix changing the memory layout and putting the entries that we reach consecutively close to one another in the memory. This way we use the spatial locality also.

Our final version has 3 level blocking. We define compile time constants `L3`, `L2` and `L1` and they correspond the block sizes (the smallest block is `L1xL1`). We calculate each block of C as a dot product expressed by $C_{ij} = \sum_k A_{ik}B_{kj}$. As we walk on A row by row and on B column by column, we have different layouts for them in the memory. The blocks of A are stored in row major order and B is in column major order. C also has row major blocks. The inner most blocks, `L1xL1` blocks, are padded with zeros to make sure that they are square matrices and they both are row major as the kernel we use works fastest with the matrices that have the same layout. We ended up choosing 480, 120 and 12 for the block sizes.

Section 4) Compilation Flags

The assignment supplement, *serial-tuning.pdf*, suggested using a combination of compiler flags to further optimize performance of matrix multiplication. At this point in the project, we already determined algorithmic schemes to take advantage of the cluster architecture in our C files. The compiler reads the C files and transforms the commands to series of simpler instructions the machine can run.

With the Intel hardware, we referenced a document on their website to compare different compiler options (URL https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler). The effects of each flag varied from general optimization (-O1, -O2, -O3), to arithmetic approximation (-no-prec-div, doesn't apply to this project), and processor specific optimization (-xCORE_AVX2). We found the most success using flag –O3. The –O3 flag is the most aggressive version of general optimization. We expected an increase in performance with –xCORE_AVX2 because it specifies our code to run SSE/AVX instruction sets best suited for the Intel Xeon E3 v3 processor family, but the boost was not significant.

Other flags which had no impact or made performance worse included –ipo (interprocedural optimization), -funroll-loops, and –ftree-vectorize. The –O3 flag we use may already capture some of these effects, showing no improvement.

There was an additional flag -prof-gen, which stands for Profile Guided Optimization. This involves instrumenting the program, running it, and then re-compiling it. We were unable to try this option because jobs are sent to the cluster and we cannot run AVX commands on the local machine. It would be worth a shot to attempt this in another setting because after running it the first time, a file is generated which contains usage information such as most commonly used methods and structures. With this information of program usage, the second compile can give higher optimization priority to specific chunks of code.

5) Conclusion

Following are the results shown for 1526 by 1526 Matrix.

Type of Optimization	Performance Enhancement Factor
Basic (`ijk`)	1.0
Basic (`jki`)	2.4
AVX Vectorization	8.3
AVX Vectorization combined with Blocking	14.1

Figures

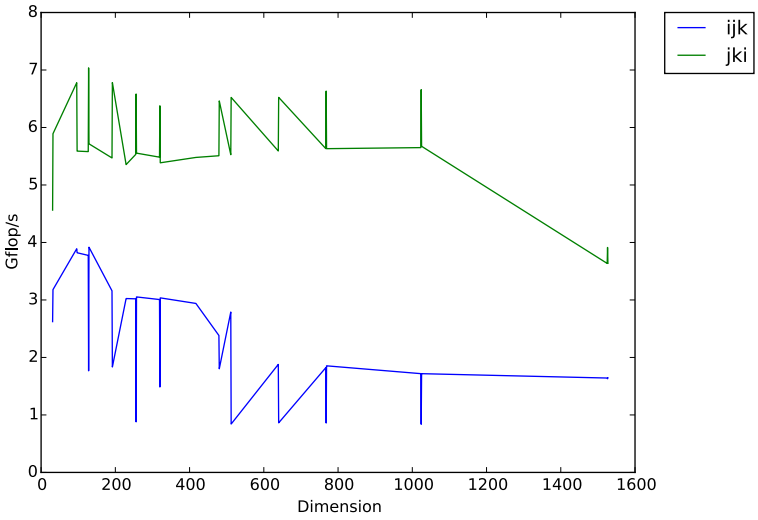

Figure 1: Speed comparison between ijk ordering (basic code) and jki ordering.

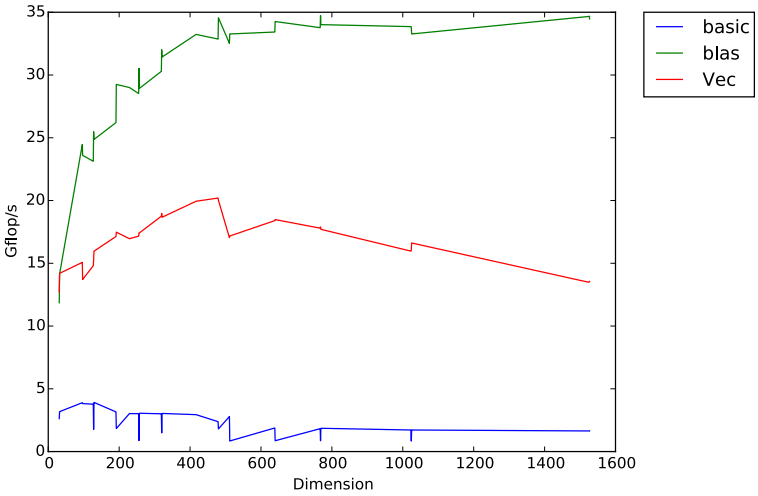

Figure 2: Speed comparison between basic, blas and AVX vectorized code.

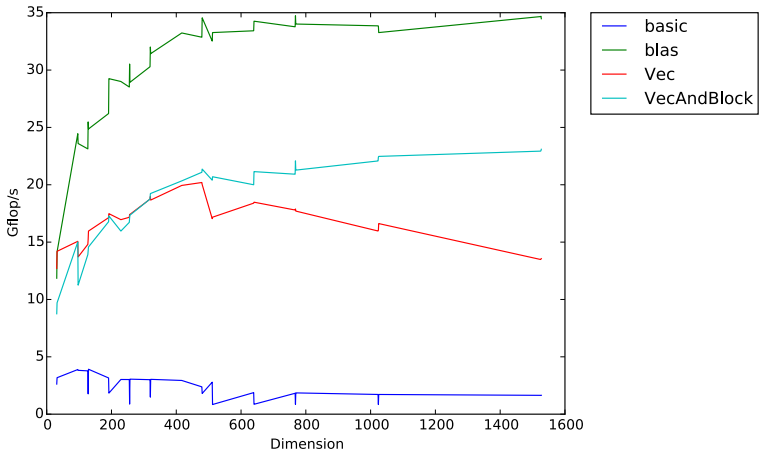

Figure 3: Speed comparison between basic, blas, AVX vectorized code and AVX code combined with blocking.