CSE 260 PA 1 Final Report

Abhishek Gupta
abg006@ucsd.edu

Sashank Vempati
svempati@ucsd.edu

1 Results

1.1 Performance of Optimized code

N	Peak GF
32	13.115
64	18.355
128	19.315
256	20.245
511	18.43
512	19.18
513	17.87
1023	18.22
1024	18.705
1025	17.625
2047	18.12
2048	18.31

1.2 Plot of performance for three versions of code

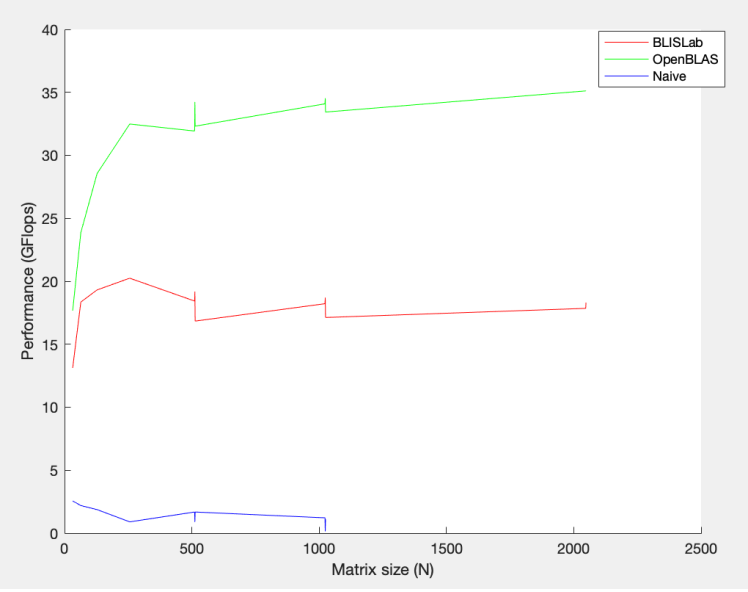

Figure 1: Plot of Performance between 3 matrix multiplies

Preprint. Under review.

2 Analysis

2.1

Just like in the BlisLab tutorial, the matrices A and B are successively divided over loops into smaller sub-matrices. A is divided into size McxKc submatrices which can fit in the L2 cache. B is divided into size KcxNc submatrices that can fit in the L3 cache. The values in A are packed in a KcxMr matrix, such that the Mr-length columnwise data in the McxKc submatrix has its data represented in sequential rows for faster access. Similarly, the values in B are packed in a KcxNr matrix, such that the Nr-length rowwise Nr data in the KcxNr submatrix has its data represented in sequential rows for faster access. These packA and packB matrices are then sent to the optimized AVX2 microkernel for the actual computation.

We use an 8x8 microkernel to perform the matrix multiplication. We first check if the given values of m and n are 8x8, if that is the case we proceed with multiplication, otherwise we copy the values of C in a Result array and pad it with 0s for making an 8x8 array (The values of A and B are already padded during the packing routine). We then do the computation using 2 iterations (in an unrolled loop) of a 4x8 kernel. Thus, we load the values of 4x8 C subarray in 8 _mm256d registers. Then, in a loop over Kc, the values of that k's 4x1 submatrix of A are loaded into 4 registers, each containing 1 value of A broadcasted into a 256 bit register. The value of that k's B is stored in 2 256 bit registers. These values are multiplied and added to the registers of C, using the _mm256_fmadd_pd operation over the loop of Kc, and finally the register values are stored back into C. We then do the same thing with the lower 4x8 subarray of C and the lower 4x1 submatrices of A, thus computing the value of C for the 8x8 microkernel.

2.2

We started off by implementing packing and editing the provided simple microkernel to work with our packing. The packing by itself gave us a performance of around 2.6 GFlops/s (compared to 1.6 GFlops/s for the original code). Then, we tuned our Mr, Nr, Nc, Mc and Kc values according to the cache sizes on our AWS machine (using getconf to get the cache values). The optimization of these values gave us a 2x speedup and the performance increased to 4.5 GFlops/s on average. Here, we decided to implement an 8x8 microkernel with a Kc value of 64.

Interestingly, we also experimented with a design where we would have a 64x64 microkernel with a Kc value of 4. This would still allow the matrices to fit in the L1 and L2 caches as required, and actually gave us a performance of around 7 GFlops/s when implemented without vectorization. However, when implementing this in a vectorized microkernel, the additional loop overhead for servicing the 64x64 kernel and the reloading of registers mitigated all the gains we saw and we were not able to improve the vectorized code performance beyond 10 GFlops/s. Thus, we decided to stick to an 8x8 microkernel.

We also experimented with a 4x8 microkernel, on the logic that a 4x8 size was the maximum we could service without the reloading of registers (It would require 8 _mm256d registers for C, 4 registers for A and 2 for B, making a total of 14 out of the 16 registers). However, we saw that servicing an 8x8 microkernel with a manually unrolled loop comprising 4x8 kernels executed twice provided minimal degradation of the microkernel performance. It had the added benefit of allowing more values to reside in cache. Thus, this was the final microkernel size we implemented, and it gave us a performance of around 23 GFlops/s on the local machine, and 21 GFlops/s on AWS.

We now decided to handle matrix sizes that were not a multiple of kernel size using an if macro (for some reason, the if statements in the microkernel did not work while if macros did). The way we check for and do padding has been described above. This checking would degrade a small part of the code, with a majority of the multiplication running without any padding overhead. While we did not see any significant degradation in performance on our local machine (22.5 GFlops/s from 23 GFlops/s without packing), the performance on AWS degraded significantly (18.2 GFlops/s from 21 GFlops/s without packing). We have referenced the performance in section (d).

Finally, we added 2 optimizations through the compiler flags, -funroll-loops (gcc loop un-

rolling) and -mavx2 (optimizations for avx2). These optimizations had minimal positive effect on our performance on AWS, indicating that the compiler had already made many optimizations for us.

2.3

From the plot of the graph, we see some irregularities in how the performance scales for large values of N. We see a significant increase in performance from around size 32 to 256 because we are able to entirely fill in the microkernel with the matrix values using the packing routine with no wasted or unused space, resulting in higher GFlops. We notice that matrix sizes that are powers of 2 tend to have the highest performance. After n = 256, the performance drops a little bit, and one potential reason why this happens is because the packing routine will call the microkernel more number of times (compared to just a few times for 32 <= n <= 256).

Another irregularity we see is going from an even matrix size to an odd matrix size (for example, n=1023 to n=1024), where we see a drop in performance for odd values of the matrix size n. The reason why the matrix slows down in performance (for example, from n = 512 to n = 513) is because the packing routine either has to perform the operation on just the last row or column which is usually left over, or the packing routine does not fill up the microkernel completely because the size is an odd value leading to unused space in the microkernel that we implemented, whose dimensions are even.

2.4

Using Valgrind on the 512, 1024 and 2048 sized matrices, we found an instruction miss percentage of less than 0.01%. We found a D1 cache miss rate percentage of 14.5% on average, which is much closer to OpenBlas's 7.1% on average than the naive loop implementation (44.5 % on average). In addition, after tuning our parameters, our total data access instructions reduced from around 1.2 Billion to 300 Million, which is much closer to OpenBlas's 120 Million data accesses on average. Our reasoning for our parameter tuning is as follows:

Since the size of the L1 cache of t2.micro is 4 Kb, and we want to fit the KcxNr submatrix of B in L1, we choose Nr as 8 and Kc as 64 as 64*8*8 bytes = 4 Kb, making optimal use of the cache. For the reasons explained above, we choose Mr to be 8 as well. The McxKc submatrix of A needs to fit in L2 cache whose size is 256 Kb. Since Kc is already decided before, we choose Mc to be 512. The KcxNc submatrix of B needs to reside in the L3 cache, which is huge (30 Mb). However, we find increasing the values of Nc to huge values actually degrades the performance slightly and thus we choose Nc to be 8192 to fit in a 4 Mb L3 cache.

In addition, we would like to specify that our performance values on AWS may be around 18GFlops/s, but on our local machine (which has the exact same memory figures as a t2.micro instance, runs Ubuntu 20.04 LTS on an x86 processor and has 4 CPUs), we are comfortably in the performance range of 22 GFlops/s. Our OpenBlas figures are similar to that on AWS. This is despite setting the AWS credits to unlimited for the instance, and we feel that our lower performance is due to an AWS problem than our code. For reference, our BlisLab figures on the local machine we described are:

N	Local Machine GF
32	14.8
64	21.2
128	23.9
256	24.5
511	23.1
512	22.9
513	21
1023	22.5
1024	22.6
1025	21.7
2047	22.3
2048	22.2

The reason why we organized the skeleton code differently from the BLISlab tutorial is because we are following the row major order for matrices which will result in faster memory accesses because the addresses containing values for each row in a matrix are aligned next to each other. This means fewer cache misses and overall higher performance, whereas the BLISlab tutorial implements matrices in column major order so implementing our matrices in column major order will cause more cache misses and reduce performance.

2.5

If we had more time, we would have implemented an even more optimized microkernel in order to achieve a higher performance for large matrix sizes. Additionally, our current microkernel has a fixed size of 8x8 double precision values, so we would have experimented with padding the packed matrices if they have smaller dimensions than the 8x8 microkernel we implemented. After this iteration, our final step would be to implement vectorization in the microkernel which will work with any size matrix c, whose dimensions are M_r x N_r. We would then choose the approach that yields the highest performance.

3 References

BLISLab Tutorial: `https://github.com/flame/blislab/`

Implementing AVX2 intrinsics: `https://www.youtube.com/watch?v=x9Scb5Mku1g&t=1753s`

Reference Guide for implementing AVX2 functions and variables: `https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics.html`