CS267 Homework 1 Part 1: Optimize Matrix Multiplication

Jingbo Wu
Dept. of EECS, UC Berkeley
BWRC
`cuiusoewin@gmail.com`

Abstract

The goal of this report is to investigate the optimization of matrix multiplication code to run fast on a single processor core of NERSC's Cori cluster. Multiple strategies are used in this progress and detailed evaluations are performed. At the end of this homework, We successfully demonstrate that by combining AVX, loop reordering, loop unrolling and block pre-fetching, an 36% average percentage of Peak could be achieved.

1 Write-up guideline

For the write-up guideline:

- The names of the people in your group, **shown on the first page**.
- The optimizations used or attempted, **explained in the main text**.
- The results of those optimizations, **Shown on the figures and text explanation**.
- The reason for any odd behavior in performance, **explained in text**.
- How the performance changed when running your optimized code on a different machine. **Shown on figures and text**.

2 Introduction

Matrix multiplication is a fundamental computing module in almost every engineering and science field. As a result, multiple methods were described in many papers which are targeting higher performance (Goto and Geijn, 2008). In this report, we have done several different optimization methods to improve our matrix multiplication program efficiency. Some of them did boost the performance a lot, while others are not so significant.

In matrix multiplication, most operations are based on following equation:

$$C = C + A \cdot B, \tag{1}$$

where C is an $N \times N$ matrix, and A and B are matrices of appropriate dimensions. The naive algorithm is to use above equation and looping over the matrix C, at the end we will get the correct answer.

While conceptually simple, the naive approach is very cache-inefficient. Since the naive code does not take advantage of spatial locality of cache. A reasonable optimization to optimize cache usage is called blocking. The matrix is truncated into smaller blocks, and then we divide and conquer and compute the smaller matrix multiplications. At the end we combine the smaller blocks to get the final answer.

In fact the full divide and conquer algorithm, which will not stop until the block size is 1, is a cache-oblivious algorithm. Thus we need to make the recursion or iteration stop in meaningful sizes. Another consideration is the size of registers, and one can develop a small kernel to do multiplication to optimize their usage. The kernel should be very efficient in computing certain size matrices and we also want to do pre-fetching for the kernel which can reduce further the time cost.

3 Optimization

This section will discuss the major optimization we implemented during this assignment.

3.1 Loop-unrolling

As the first operation, we implemented loop unrolling with the step size as four. By unwinding the loops we can reduce the instruction amount to improve the performance, since from the assembly language one may notice that control instructions will lead to branch prediction which typically is going to reduce the performance. Also we do not want to compare the loop condition frequently since every time we do so we need to fetch the data from cache. In our example, the performance boosted by 3%. And this turn out to be the limitation of this step. To achieve higher performance wee need to combine this step with pre-fetching and transposing matrix B.

We hope all the elements are stored in row-wise manner (default in C) so that continuously loading elements will not cause much cache misses or conflicts. So we implemented the matrix transpose in the following optimization iteration steps.

3.2 Changing the order of loops & multi-level blocking & prefetching

As we mentioned above, cache misses are the one big problem slowing down the algorithm. So we changed the order of the loop: instead of calling B in the beginning loop, we move it to the second loop. Now the program becomes:

- Prefetching one row of A,
- Load one element of B,
- Perform multiplication, update one row of C corresponding to the row from A and the element from B.

The reordering boosts the performance by 5% or less. Instead of one level of blocking, we also perform multi-level blocking and almost no improvement is observed. Since typically L1 cache is a direct mapped cache, all these elements in the column map to the same cache line. This leads to an extraordinary number of cache misses, or cache conflicts. Thus reordering the loop actually did not solve the problem.

3.3 4 by 4 SIMD using mm128 regs & better prefetching & multi-level blocking

In this part, we try our best to implemented the optimization methods introduced by Goto(Goto and Geijn, 2008) By following the hint slides we found for this class 2 years ago, we implemented a kernel function which will compute the 4 by 4 matrices multiplication using 128 bits AVX registers. By combining this, multi-level blocking, and better prefetching functions, we now achieve about 25% performance on Cori. The kernel function is loop-unrolling to 4 by 4 size matrices computation and we did see a big improvement in this step.

Now the prefetching functions are fetching 4 by K rows and 4 by K columns and store them in stack memory for faster memory access. In this step, we did try to use the aligned heap memory, i.e. the aligned_alloc and posix_memalign functions, but weird things happened and allocating memory this way actually reduced the performance. To use the unaligned memory, we use the _mm_loadu_pd and _mm_storeu_pd. We believe this is actually slowing down our program.

The program becomes:

- Perform two level blocking.
- Divide the matrices into 4 by 4 chunks and perform multiplication.
- Handle the corner cases

3.4 8 by 8 SIMD kernel using mm256 regs & prefetching & multi-level blocking

In this step, by noticing the fact that Cori actually supports 256 bit AVX operation, in order to enhance the performance further, we switched to 256 AVX operations. We reimplemented the kernel function and the prefetching functions, finally boosting our algorithm's performance by 10%. We also implemented 4 by 4 mm256 regs kernel and benchmarked it against the 8 by 8 version. The results can be found in the figure.

Besides, we also tried in-memory transposition of the matrix, which may be useful in conditions where memory is limited. The outcome was roughly identical to the method which copies the data. This insight actually can be illustrated by the fact that the $\mathcal{O}(n^2)$ complexity of the matrix transposition is more or less negligible compared to the $\mathcal{O}(n^3)$ complexity of matrix multiplication.

3.5 Tuning block size

Since we are doing multi-level blocking, we spent a long time on tuning the block size. At the end the block size that yields the best average peak percentage is chosen.

3.6 Compiler flags

We spent a significant amount of time on playing with compiler flags for *gcc*. Initially, these flags were promising and resulted in around 20% average peak performance on a different Haswell CPU with minimal changes to the code (except for reordering loops and adding annotations so that loops could be automatically vectorized). However, this did not translate well to Cori and it was hard to understand or improve on, so we explicitly vectorized the computation. In the end, we kept the flags that contributed to our performance:

- -ftree-vectorize
- -march=core-avx2
- -funroll-all-loops
- -msse

Since the final version of our code explicitly vectorizes the inner multiplication, these flags do not offer much benefits.

3.7 Final implementation

- Multi-level blocking.
- Manual Loop unrolling.
- 4 by 4(or 8 by 8) SSE multiplication.
- Prefetching in only in the inner level.
- Handling corner cases: unaligned SSE load and store instructions, special cases.
- Loop reorder
- Various compiler optimization (-ftree-vectorize -msse -msse2 -msse3 -ffast-math -mavx -mavx2 -march=core-avx2 -funroll-all-loops)

4 Other attempted optimizations

This section discusses some other attempted optimizations that did not make it into our final implementation, but provide some useful insight nonetheless.

We spent about two days on trying to make the memory aligned and add padding to boost the performance. But there are still some bugs due to edge cases that we did not have time to fix. We commented the error detecting code in benchmark.c and did see a significant amount of performance boost by making use of the aligned register load command and adding padding to the matrix to avoid corner cases. Unfortunately due to the complexity of our multi-level blocking structure, the program kept giving us memory errors and computing errors. So if time permits, for further implementation, we will definitely finish the debugging of this part and examine the outcome.

We also considered the possibility that the multi-level blocking has a large overhead cost and we tried removing it, but the performance actually decreased 1-2% by doing so.

5 Performance

Figure **??**(see the end of this report) summarizes the measured performance of the optimizations we attempted. From the figure we can see that most of the optimizations we tried could improve the performance on top of others. The final combination of these optimization techniques gives an average of peak percentage of 36%.

Figure 1 illustrates a subset of the same optimizations running on a laptop. Here we see more erratic behavior and dips in performance for the BLAS implementation. These tests are not well controlled since Intel CPU on this laptop will increase the clock frequency to boost the performance. Also, the tests shows sensitivity to other processes running on the same system.

Backpropagation

6 Conclusion

At the end of this assignment, our implementation reaches an average percentage of peak performance equal to 36%. Given more time, we would like to try our best to implement the padding part and remove the multi-level blocking to make the padding easier to debug(since both of us are not very good at C).

In retrospect, we have spent too much time on the multi-level blocking kernel and padding debugging. It would be better if we simply use one level blocking and unroll to 16 by 16 size matrix multiplication. Or tuning the multi-level blocking to fit the L1, L2 Caches.

7 Team Contribution

Jingbo did most of the work to speed up the serial code, including all of the optimizations included in the final implementation. Ashvin attempted some blocking and automatic vectorization which proved less effective and mainly investigated the parallel section. We met several times to discuss strategies for both sections.

References

Kazushige Goto and Robert A Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):12.

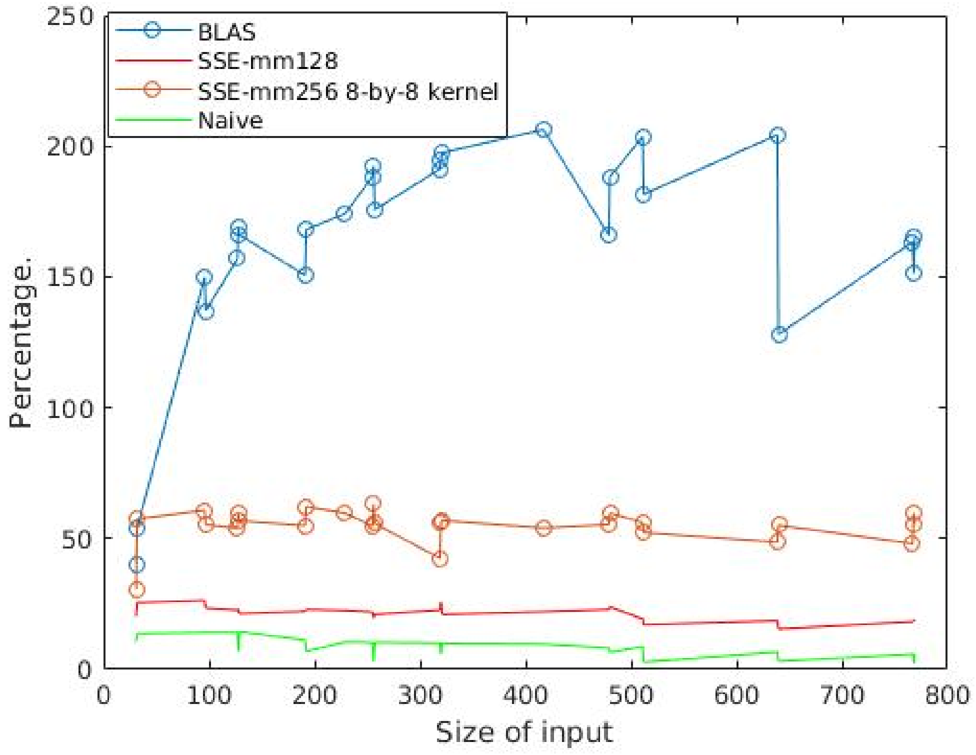

Figure 1: Percentage of peak achieved by different optimization combinations on a Dell XPS 13 with i7-6500U CPU.