Optimizing Matrix Multiplication

CS267 Homework 1

Group01: Xiyu Zhai, Jingwei Jiang, Yufeng Zheng

06 Feb 2020

1. Introduction

In this assignment, we focus on improving the performance of matrix multiplication calculations on a CPU using only single core. Obviously, number of data transferred between memory and CPU is high compared to the number of operations, this program is memory bound, and the performance of the program can be improved by making the data fit better to the memory hierarchy. We improved the CPU utilization from 4% to about 48% through methods such as blocking (adjusting the block size in the original code and implementing multi blocking), matrix pad, loop unrolling and utilizing proper Intel intrinsic functions.

2. Contributions

- Xiyu Zhai did the loop unroll, SSE intrinsic functions and matrix padding part.
- Jingwei Jiang did the multi blocking optimization, AVX intrinsic functions, boundary optimizations, and locality optimization.
- Yufeng Zheng analyze the result and wrote most of the report.

3. Optimizations Attempted

3.1 General orientation of doing optimization

Our purpose in this project is to optimize the code such that it takes less time to solve the same-sized problem — matrix-matrix multiplication than using a naive element-wise scalar product and summation code. To achieve this, we clarify first, that the major time spent when the code is executed can be divided into two kind of work:

- 1. memory access,data transfer
- 2. arithmetic calculations. Since the basic way of doing matrix-matrix multiplication is fixed, the total number of FLOPS are given and unchangeable in our project.

However, we could still manage to increase the FLOPS/s in order to get a more efficient code. To approach the Roofline of theoretical peak FLOPS/s, we implement intrinsic functions in our code. This considerably increase the portion of vector instructions than scalar instructions. In order to decrease time spent in data transfer, we manage to take full use of cache(register) where memory access is much quicker than from DRAM.

The Original block program's Average percentage of Peak is 4.6%

3.2 Multi Blocking

Cori haswell has L1,L2 cache, and L1 cache is separated into L1i(for instruction) and L1d(for data storage)Fig.1. Based on this fact, we use two block size parameters to use both cache levels. After few attempts, we tuned the parameters as Large block size is 18, mini block size is 2. The average percentage of Peak is improved to 6%

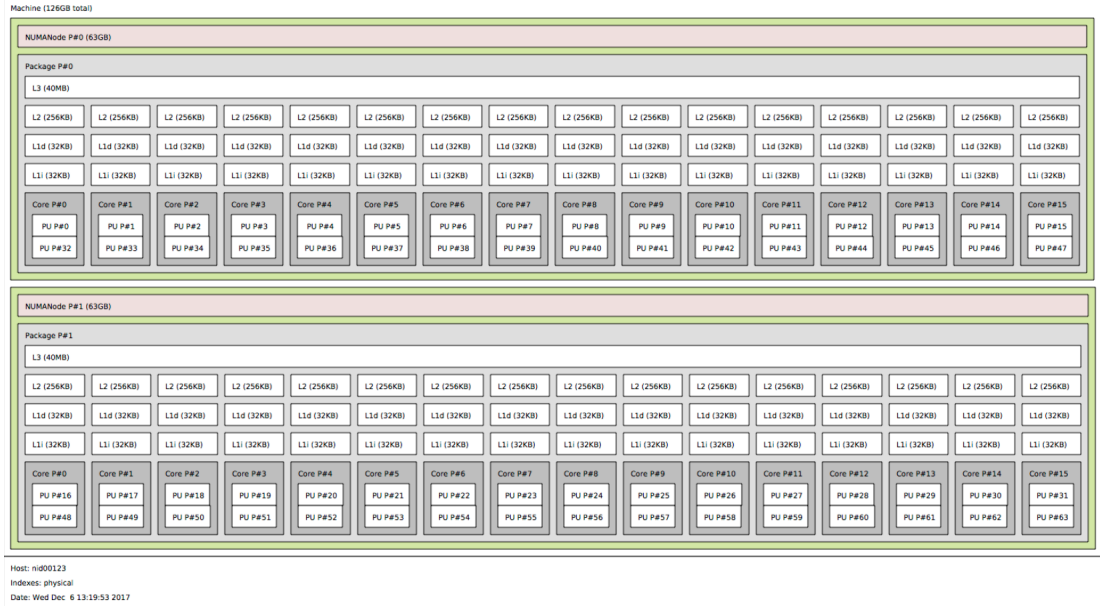

Figure 1: Architecture of Cori nodes, note that we are only using one core in this project, although efficiency might be greatly enhanced by paralleling over multi-core using MPI.

3.3 Change the Order of Loop

In the original setting, the order for looping is i, j, k from outer to inner. i corresponds to the rows of the C matrix, while j corresponds to the columns. k indicates the status of the inner-most loop, that is, the position on i-th row of A and j-th column of B. Because the matrix exists in memory as a column majored method, such a loop order is very unfavorable for reading A into CPU. This inspires us, trying to loop in a different order. The looping order can improve the efficiency of the program slightly based on the matrix storage forms.

3.4 Loop Unroll

Loop unroll can produces larger basic blocks, which allows the compiler to apply strength reduction to simplify expressions. And also exposes an opportunity to perform instruction scheduling for the compiler.

After unroll the basic block computation, we tuned the parameters that large block size is 32, Mini block size of M is 4, of N and K is 2. Based on the existing Blocking code, we performed loop unrolling on the innermost loop of sub-matrix multiplication. Compared without loop unrolling, the performance is improved to 8%.

3.5 Optimize Flag for Compiler

After some tests, we decided to use the following optimize flags.

```
#pragma GCC optimize("O3","Ofast","inline","unroll-loops","prefetch-loop-arrays")
```

3.6 Memory Alignment and SSE Intrinsic Function

The A,B,C matrix given from benchmark do not necessarily aligned in the memory. This makes it slower for the processor to get access to the memory location storing matrix[1]. To use the Intel automatic

vectorization speed-up, we need copy the original matrices A, B and C to allocated contiguous memory space and make sure the memory space are aligned. Thus, we allocated a memory space for the small block of A and B for the block-size multiplication computation each time.

After this, we switch to compiler intrinsic functions which improve the performance significantly. We first managed to use SSE to optimize the multiplication of $2 \times K$ and $K \times 2$ metrics. The performance with SSE is improved to 30%

3.7 AVX Intrinsic Function

After doing all this, it comes to the optimization of small block matrix product. we want the small matrix fit into the register and try to vectorize the instructions as much as possible. Since cori has a 256-bit register, besides using SSE, we then managed use AVX256 intrinsic function[2] as much as possible. This holds 4 double precision float number as a vector. Suppose we have $m \times n$ small matrix for each small block operation using SIMD, both m and n would be a multiple of 4. Thus we use 8×4 to get the most optimized result. And we tuned the parameter Blocksize to be multiple of 8 to ensure every operations for the small block are valid(with a memory trade-off because we need to enlarge our matrix).

As AVX intrinsic function supports 256-bit registers, we can proceed $8 \times K$ and $K \times 4$ metrics multiplication for inner block computation. Our code now run around 35% peak performance.

3.8 Matrix Transposition and Boundary Optimizations

The original matrix was stored in memory in the form of a column major. But for the calculation of A times B, where $A, B \in R^{n \times n}$, we prefer A to be stored in a row major way, in order to achieve spatial continuity and reuse. We tried using loop unrolling and transposing of matrix A simultaneously. For some input metrics size, when doing block division we expand the block matrix for the inner block computation.

Then after we combined with the previous multi-blocking, and tuned the block sizes, the average percentage of peak reached to 37%

3.9 Matrix Padding

After analyzing the performance of different input matrix size, we found that the odd size matrices operation brought most of the poor performance. And previously, we ignored the discontinuous memory of matrix C. To overcome these difficulty, we define new group of aligned matrix A_padded, B_padded and C_padded which are the original matrices filled up 0 to the new Matrix size in multiples of eight.

And then we copy the original matrix element into new aligned matrices we defined after which we always use the new pad matrix, and finally we copy the matrix element of C_padded back to C. The copy process is done only once for each element, this optimization considerably accelerates the code. We also add "restrict" attributes to all pad variable to get further optimization.

By using the matrix padding, the average percentage of peak reached to 41%.

3.10 Locality Optimization

A further step to optimize our code is to add some specific optimization flag for the compiler, including O3, Ofast, inline, unroll-loops, prefetch-loop-arrays et al. However, we find that the gain of using these flags is very limited.
Another way is to increase the locality of the code —try to use the same memory location(same variable)as much as possible[3], and try to use adjacent location when accessing an array. Based on this rule, we further optimize our copy function and function for small matrix product and get a boost of 7%. Finally we have a code that could run at 48% of peak performance.

4. Performance Analysis

All the performance of previous optimizations are shown in Fig2

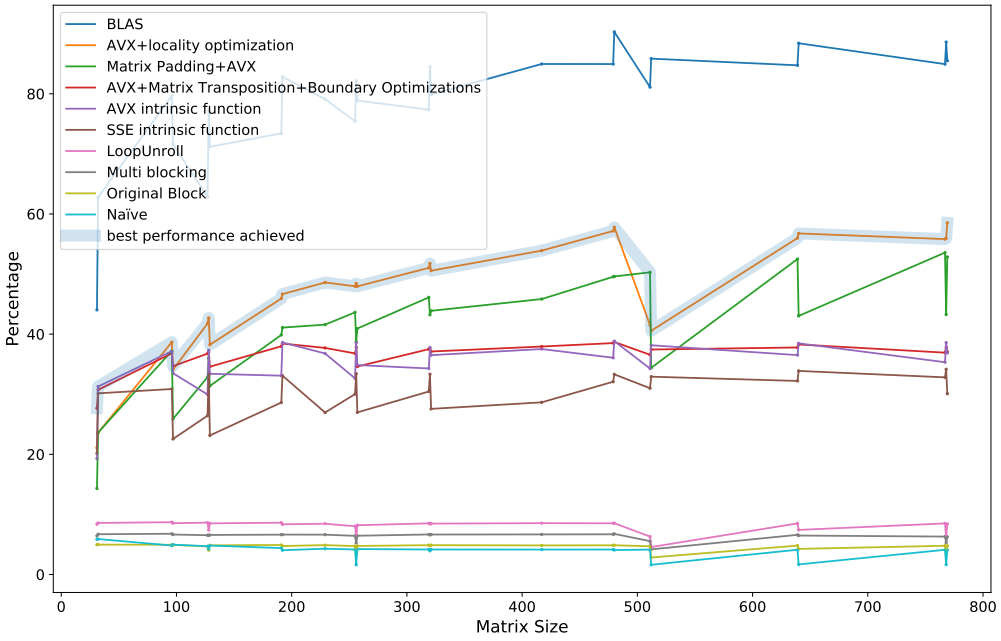

Figure 2: Performances of different optimization methods.

5. Performance on another machine

5.1 Thinkpad X1 PC

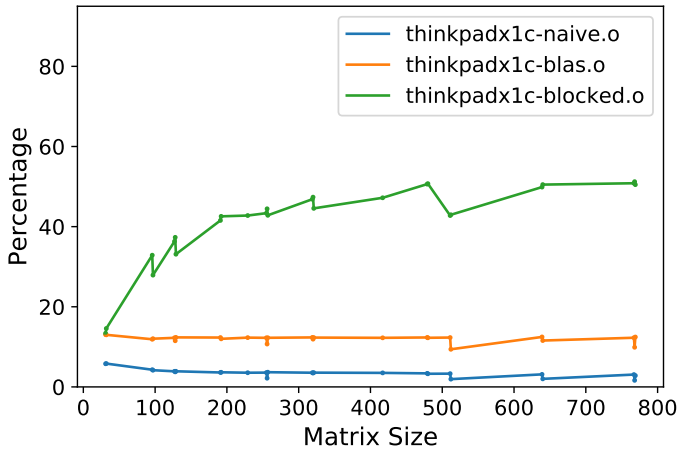

Figure 3: Performances on the thinkpad x1c.

We use our own PC (the clock speed of which is 2.4GHz) to test our final block code, naive block and blas as shown in Fig 3. We think it is the BLAS version that causes the poor performance of the BLAS. While the performance of our own block code is similar with the performance on Cori node.

5.2 KNL Node

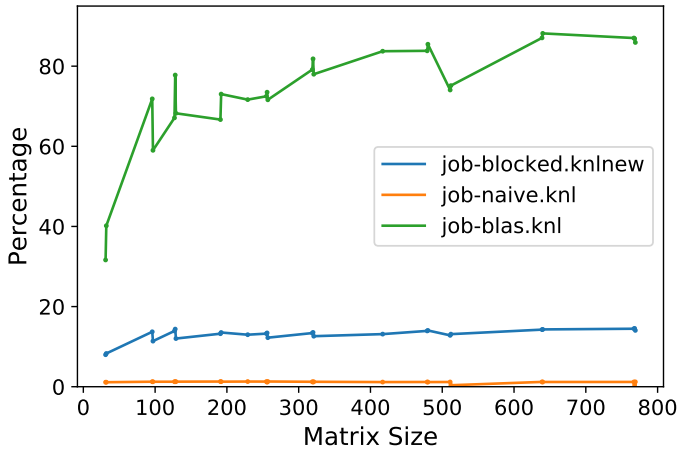

Figure 4: Performances on Cori with KNL node.

We also run our code on KNL node with one single core using same cpu-frequency limit as shown in Fig 4. Since KNL use a 512-bit architecture, our AVX256 intrinsic functions do not perform as well as it was on haswell. It is estimated that a change to AVX512 intrinsic function would enhance the performance considerably.

6. Discussion

Note that the final submitted code cannot beat all other methods we have proposed in all matrix sizes. Even using only the ideas mentioned in this write up, the CPU still has room for further improvement. Performance can still be further improved by the following method: using specific optimization methods for matrices of different sizes.

We observe that, for example, the Multi Blocking method performs better for large-sized matrices, This is expected, because large-sized matrices are difficult to fit into the L1 cache. With the help of the L2 cache, the calculation can be further accelerated. Therefore, in fact, the program can choose the most suitable optimization method through the different sizes of the matrix.

Based on our existing code, we can obtain the performance denoted by the light blue line in figure 2 with slight modification, that is, the maximum value of the CPU utilization of each method on each specific matrix size. We regard this idea will definitely improve the overall performance and will incorporate it into future work.

References

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Morgan kaufmann, 2016.

[2] "Intel® intrinsics guide." `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`.

[3] "Locality of reference and cache operation in cache memory." `https://www.geeksforgeeks.org/locality-of-reference-and-cache-operation-in-cache-memory`.