CSE260 Assignment1 - Cache Blocking Matrix Multiplication

Ho-Lun Wu(A53271935) Tan Chien(A53268778)

Jan. 2019

Contents

1 Introduction

In this experiment, we aimed to optimize square matrix multiplication operations on a single core environment with respect to GFLOPS. We successfully reached the performance of BLAS (23.1 GFLOPS, N=1024) provided by the instructor with our implementation (23.3 GFLOPS, N=1024) [2] [3] and outperformance is more stable than the BLAS.

2 Experiment Environment

We conducted this experiment on an Amazon EC2 t2.micro instance, which runs with x86_64 architecture. The CPU model of the instance is Intel Xeon E5-2676 V3 2.4GHz. The CPU is fully hypervised and we are using 1 CPU, 1 thread and 1 core. The total size of memory is 1GB with a three-level caching of size as follow: 12×32KB 8-way set associative both L1 instruction cache and data cache, 12×256KB 8-way set associative L2 cache and 30MB 20-way set associative shared L3 cache. All cache have line size of 64 bytes [1].

3 Optimizations

3.1 Compiler Optimization

For program compilation, GCC 5.4.0 with the following flags:

- -Ofast: global optimization
- -ffast-math: faster math computation
- -funroll-loops, -funroll-all-loops: loop unrolling
- -mfpmath=sse: auto-generate floating point arithmetic with SSE
- -msse, -mavx, -mavx512f, -mfma: enable SIMD

3.2 Cache Optimization

As the total computation is the same for all implementations (n^3 kernel), the performance will be a function of data accessing latency, which means that the as we have less memory access time, we are getting better performance. In order to lower memory access time, we should decrease cache misses and always let data being accessed frequently to stay in fast memory. We will discuss in detail in the analysis section about the size we allocate for each cache block size.

3.2.1 L3 Cache Blocking

As we are having a L3 Cache size of 30MB, which is able to fit all three matrix of maximum benchmark size ($N = 1024$, 8MB for each matrix, 24MB in total). Therefore, the size of L3 cache block assigned doesn't significantly affect the performance.

3.2.2 L2 Cache Blocking

Since we have L2 cache size of 256KB, which means that we are able to fit up to 256K / 8 doubles. We allocate a double array of size 128×256 to improve cache locality.

3.2.3 L1 Cache Blocking

Since we have L1 cache size of 32KB, which means that we are able to fit up to 32K / 8 doubles. We allocate a double array of size 8×256 to improve cache locality.

3.3 Instruction Optimization

To further optimize the performance of our implementation, we implemented mathematic operations using AVX registers (both 256bit and 512bit on machine if supported). We are able to achieve a higher performance since the calculation is doing in parallel as size of 4 or 8 doubles depending on the size of registers we use.

3.4 Implementation Optimization

3.4.1 Matrix Padding

To avoid dealing with the different size of block in the most inner block, we pad the matrix to the multiple of 4/8/16 which depends on the block size we choose. By doing so, we don't need to tackle the different size of block and the compiler is able to unroll more loops due to the constant block size. Using *memcpy* with SIMD optimization, the overhead of padding the matrix is extremely low.

3.4.2 Memory Aligning and Packing

Because the SIMD instructions have better performance when loading from aligned memory, we pack the partial matrix into an aligned array. Also, the consecutive memory is better for the cache utilization.

3.4.3 Inline Function

By using the *inline* specifier to inline the functions, it reduces call instructions which needs to maintain the stack frames and flush the CPU pipeline.

4 Results

Figure 1: Performance

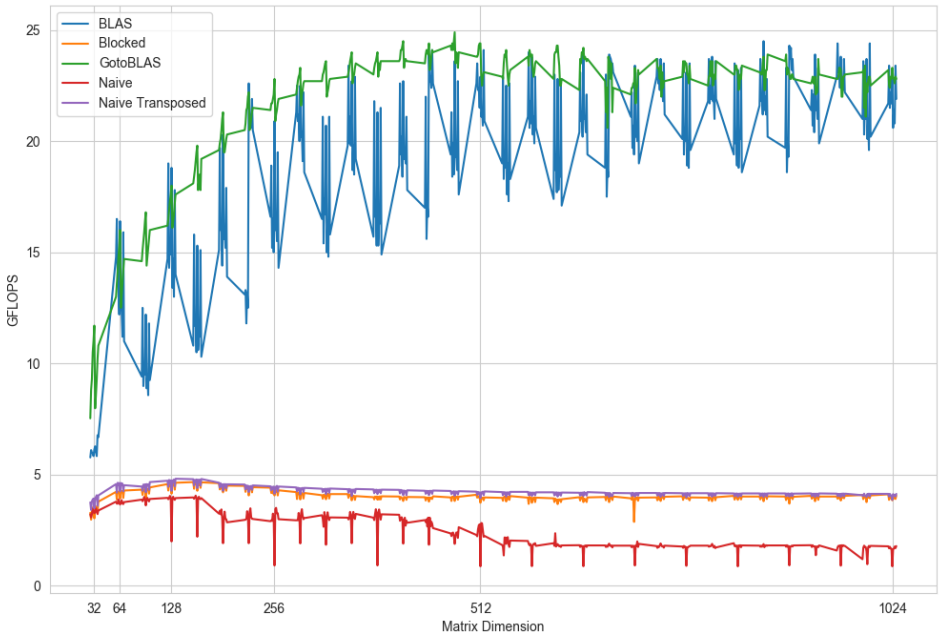

Figure 1 shows our implementation (GotoBLAS) comparing to other implementations, where all implementation other than GotoBLAS are original implementation provided by instructor ($BLOCK\_SIZE = 256$ is selected and $TRANSPOSE$ is enabled for the *Blocked* implementation).

5 Analysis

5.1 Program Behavior

Figure 2: Algorithm [3]

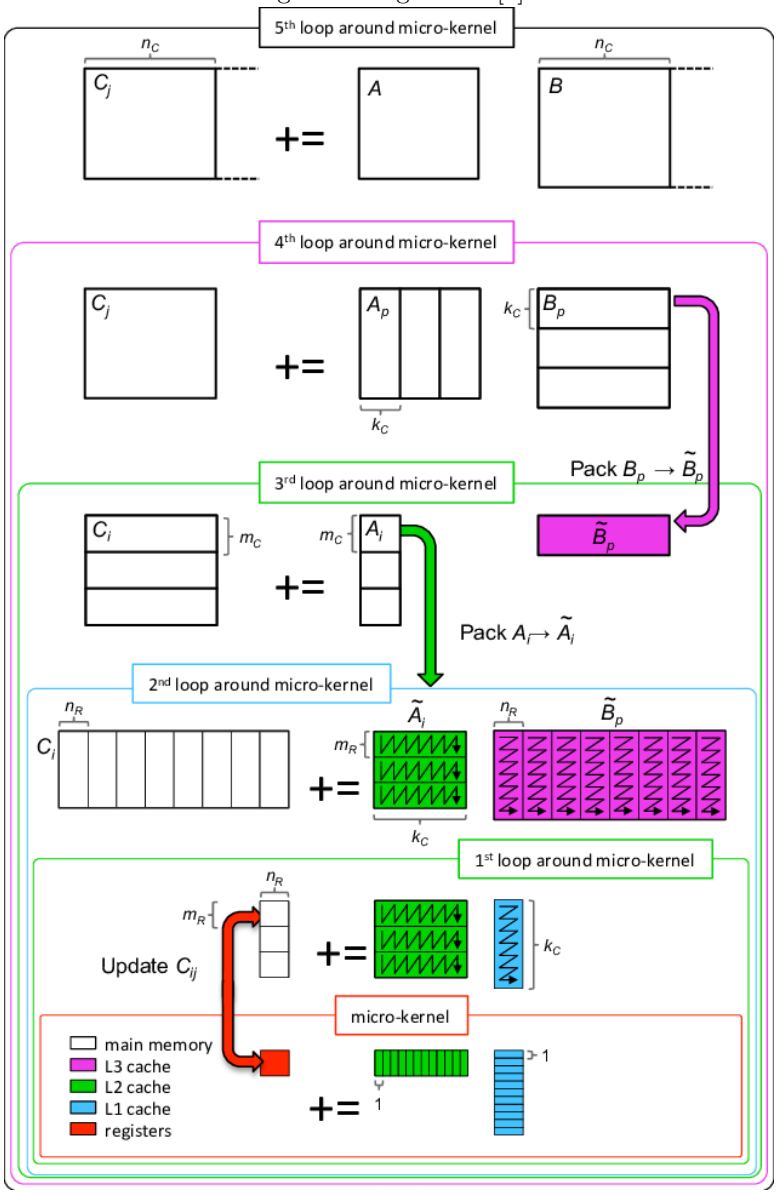

Figure 2 shows the hierarchy of how the algorithm works. At 5^{th} loop, we have a parameter n_c which is the number of columns we would like to process per iteration. We are having $\frac{LDA}{n_c}$ iterations in 5^{th} loop. At 4^{th} loop, we have another parameter k_c which will be the number of rows we would like to process per iteration. We are having $\frac{LDA}{k_c}$ iterations in 4^{th} loop. Now we would like to fit the block into L3 cache for faster memory access. The block size (purple region) is $n_c \times k_c$ which should be less or equal to size of L3 cache. At 3^{rd} loop, we have a parameter m_c which is the number of columns we would like to process per iteration. We are having $\frac{LDA}{m_c}$ iterations in 3^{rd} loop. And we have some optimization here for memory accessing. We unrolled the purple region into a consecutive array, which is denoted as pB in our implementation. Notice that we would like to fit another block (green region) from matrix A into L2 cache. The block size (green region) is $m_c \times k_c$ which should be less or equal to size of L2 cache. We also unrolled the green region into a consecutive array, which is denoted as pA in our implementation. At 2^{nd} and 1^{st} loop, we have two parameters m_R and n_R which are the block dimensions we would like to process per iteration. We would like to have the arithmetic operands in micro-kernel located in registers, thus the dimension $m_R \times k_R$ should be less or equal to the number of registers. And n_R should be multiple of 4/8 because we can optimize it with 256/512 bit AVX registers, which can hold 4/8 doubles. Also, we would like to fit one of the operand block (blue region) in the 1^{st} loop into L1 cache; therefore, $k_c \times n_R$ should be less or equal than size of L1 cache.

Figure 3: Purple Region Block Size (L3 cache)

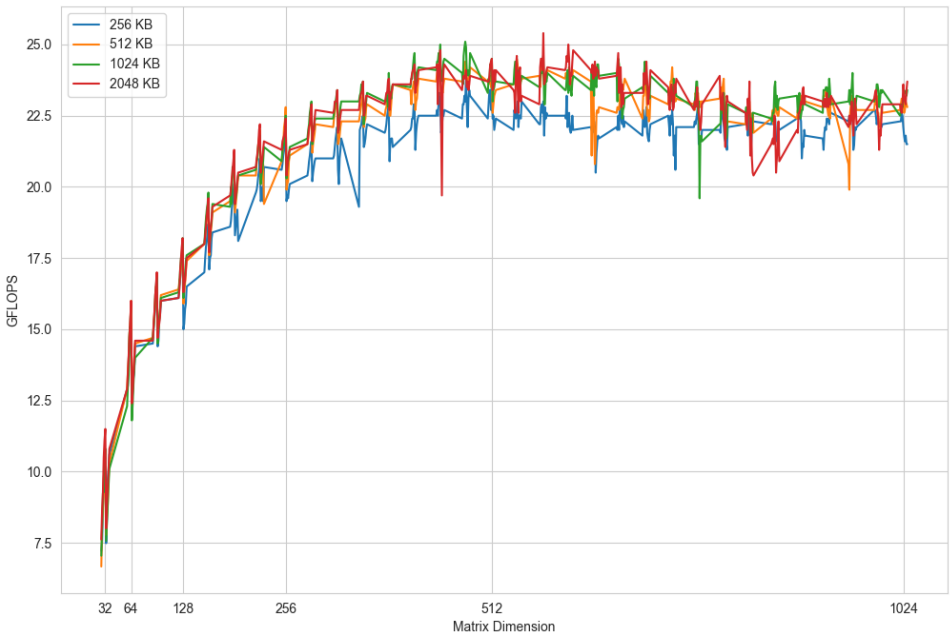

Figure 3 shows the performance for different block size we try to fit (purple region in Figure 2). It has little difference in performance with respect to the block size we assign. The main reason is that we have L3 cache size of 30MB which is totally sufficient to hold all the data in it. We can still see that if we assigned a larger region for the block, we are getting a better performance. However in our experiment, we found that the best performance will be using 1MB instead of 2MB. One possible reason is that we have overhead copying data to the block we assigned and the best performance sweet spot is 1MB.

Figure 4: Green Region Block Size (L2 cache)

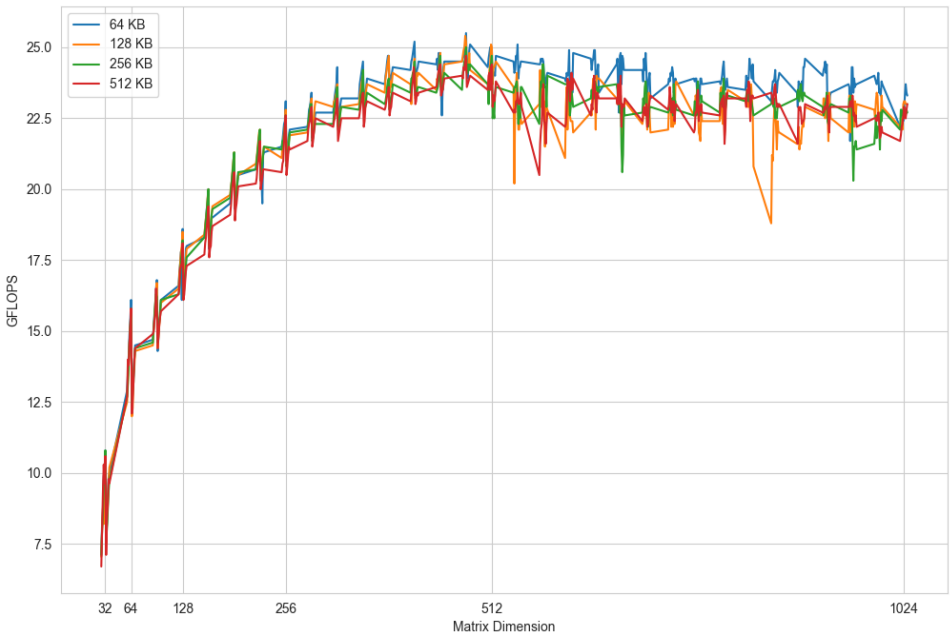

Figure 4 shows the performance for different block size we try to fit (green region in Figure 2). Interestingly, we have little performance difference with different block size. One possible reason is that we are in a virtualized environment and the cache miss penalty of L2 cache might not be that significant since we are having enough L3 cache.

Figure 5: Blue Region Block Size (L1 cache)

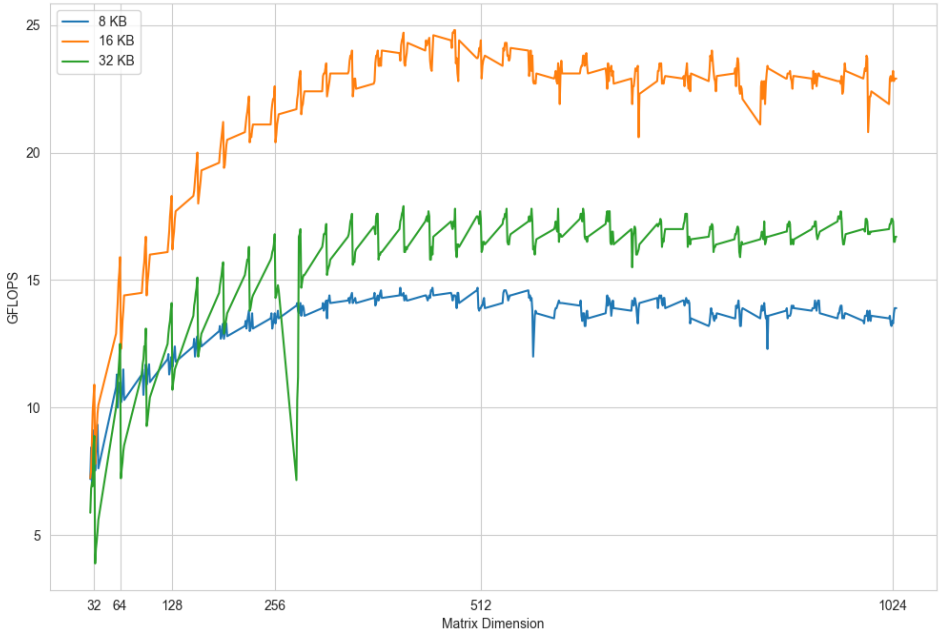

Figure 5 shows the performance for different block size we try to fit (blue region in Figure 2). As expected, the larger block size we assigned, the better performance we will get to utilize the L1 cache. However, it is interesting that though we have a 32KB L1 cache, we are getting worse when assigning a block size of 32KB. One possible reason is we might not be able to use all 32KB of L1 cache thus we might get cache misses if we assigned the same amount of memory. We believe that there should be a sweet spot between 16KB and 32KB however our implementation restricts that block size should be certain values (n_c is multiple of n_R) otherwise we will be penalized by more overhead.

Figure 6: AVX Optimization

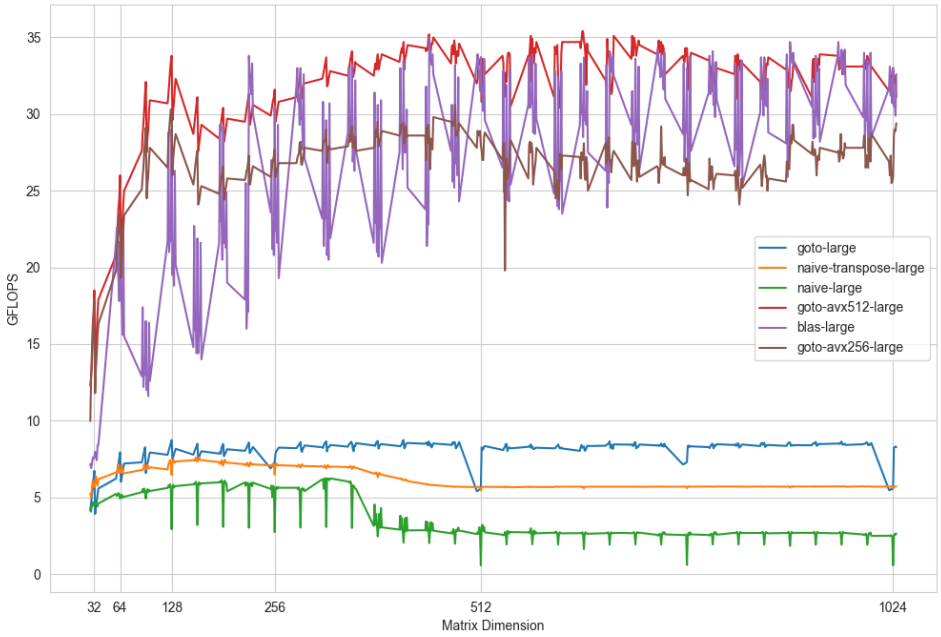

Figure 6 shows the performance for enabling 256/512 bit AVX registers. Since only c5.large supports 512-bit AVX register, we have the performance comparison environment in c5.large. Naive-large is the original implementation of matrix multiplication, which will be the baseline of performance evaluation and blas-large is the library implementation, which will be the target performance we would like to achieve. Goto-large is our implementation not using AVX registers and both goto-avx256-large and goto-avx512-large enable AVX registers operations where the register bit size is denoted. As we can see, we get little improvement when not enabling AVX registers (blue line) but massive performance boost when enabled. And as expected, using 512-bit AVX registers has better performance than using 256-bit ones.

5.2 Development Process

In the beginning, we enable several optimization flags of GCC compiler and the transposed naive multiplication achieves Gflops = 4 and the BLAS multiplication can achieve Gflops = 23. Then we referred to the provided blocked multiplication and modify the program to apply multi-level blocked multiplication including L3, L2, L1 cache and registers. After tuning the multi-level blocked multiplication program, its Gflops reaches around 7 or 8 which is much slower than BLAS program. To obtain better performance, we make research on several papers, such as Goto multiplication [2]. In this paper, Goto provides a different way to utilize the different levels of cache. After implementing the Goto multiplication and tuning the parameters, we found that even if we reduce

the number of cache misses dramatically, the performance just improves a little bit. Due to that, we considered our program as a CPU-bounded program. In order to improve the performance, we the SIMD instructions (AVX 256/512) to vectorize the floating-point operations. After enabling the SIMD instructions, our program is able to reach the performance of BLAS multiplication and our performance is more stable than the BLAS. Finally, when evaluating the performance of the programs, we notice that the performance of t2.micro isn't stable enough. To deal with this issue, we modify the benchmark. Instead of using the mean value as the Gflops, we eliminate the outliers of the samples and calculate the mean value of the remaining samples.

6 Future Work

It would be interesting challenging to paralyze the algorithm if we have more CPUs. It would definitely perform much better since we almost reach memory limitation.

References

[1] Cpu world intel xeon e5-2676 v3 specification. `http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2676%20v3.html`.

[2] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.

[3] Jianyu Huang and Robert van de Geijn. Blislab: A sandbox for optimizing gemm. 08 2016. `https://www.researchgate.net/publication/307564216_BLISlab_A_Sandbox_for_Optimizing_GEMM`.