CSE260 PA1 Report

Weixiao Zhan

April 25, 2024

1 Results

1.1 peak performance

N	GFLOPS
32	8.79
64	18.5
128	21.7
256	23.3
511	21.6
512	21.9
513	23.3
1023	23.6
1024	24.2
1025	22.6
2047	24.3
2048	24.7

Table 1: peak GFLOPS performance by N

1.2 graphs

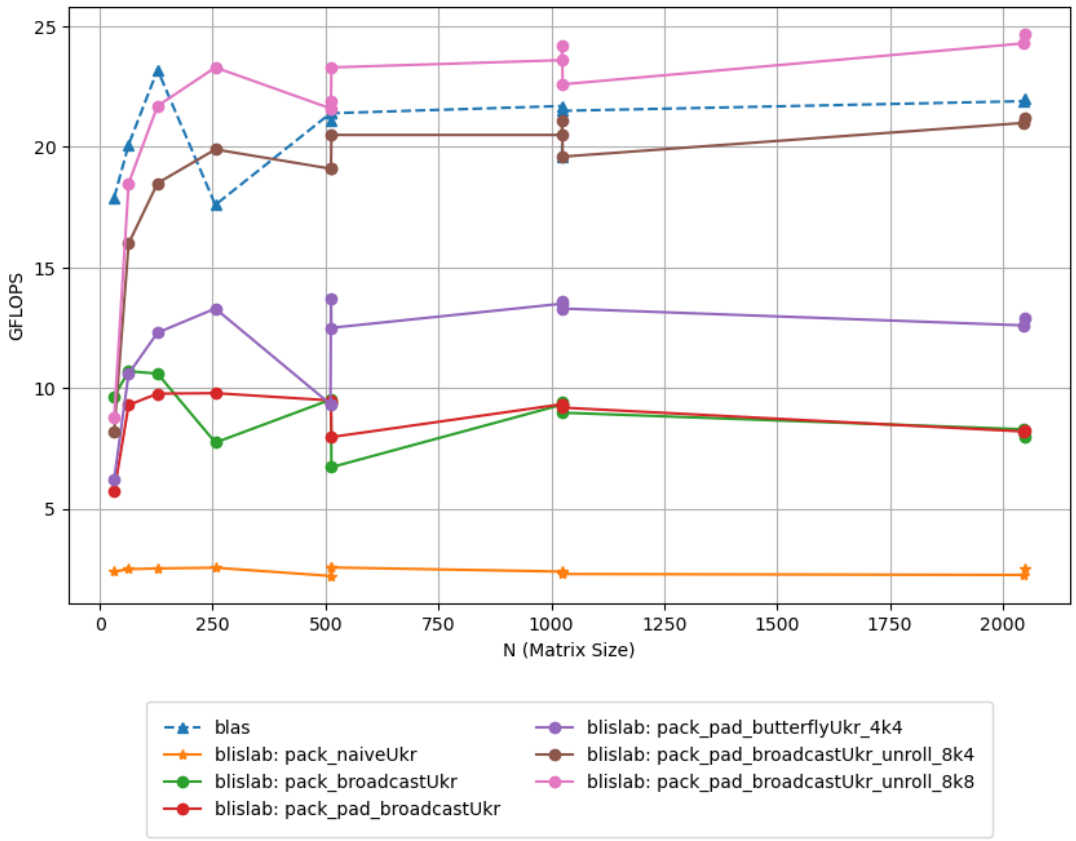


Figure 1: GFLOPS Performance by N

2 Analysis

2.1 How program works

The skeleton of this program is the 5-layer GoTo routine mentioned in lecture and shown in Figure 2. There are a few options to consider: whether to pad data; what micro-kernel to use; and some methods, such as unrolling and butterfly, works best with padded data. In addition, there are 5 hyper-parameters, m_c, k_c, n_c, m_r, n_r, to be tuned based on hardware.

Based on results, the best scheme for large matrix multiplication is to **pack and pad data, and use** $[8 \times k \times 8]$**-unroll broadcast micro-kernel**.

2.1.1 pack

Packing is to rearrange data into cache friendly layout, also called panel(s). Specifically, packing matrix `A` involves transposing from row major to column major layout. Packing matrix `B` is copy distant data to adjacent memory address.

2.1.2 pad

When matrix `A` and `B` aren't the size of multiple of m_r and n_r, the micro-kernel sometimes need to handle non-full matrix multiplication. To avoid performing brach condition in every micro-kernel execution, one can zero-pad matrix `A` to the size of $\left\lceil \left\lceil \frac{m}{m_r} \right\rceil m_r \times k \right\rceil$ and `B` to $\left\lceil k \times \left\lceil \frac{n}{n_r} \right\rceil n_r \right\rceil$.

Since the packing process moves data around once anyways, integrating padding of `A` and `B` into the packing procedure can reduce communication cost. On the other hand, padded-`A` and padded-`B` produce padded-`C`. To handle the padded result, The program first allocate enough memory for padded-`C`, store intermediate result at that location, and finally use `memcpy` to copy desired values to `C`. In summary, the overhead of padding is one extra copy and one extra memory consumption of `C`, and some negligible cost for `A` and `B`.

With padding, the micro-kernel only need to handle fixed-sized ($[m_r \times k \times n_r]$) multiplications, which enables unrolling in micro-kernel. Moreover, the signature of micro-kernel can be simplified (no longer need to pass m and n) to reduce cost of function call.

2.1.3 broadcast micro-kernel

The broadcast micro-kernel leverages arm-ave instructions. This micro-kernel implemented the method described in lecture and shown in Figure 3a. The hardware supported arm-ave 256bits instructions, so m_r, n_r should be multiple of 4 for best arm-ave performance.

2.1.4 unroll broadcast micro-kernel

When the data is padded, it is possible to unroll the loop that iterate through each row of `B`-panel and column of `A`-panel in micro-kernel. Three types of unrolling are implemented:

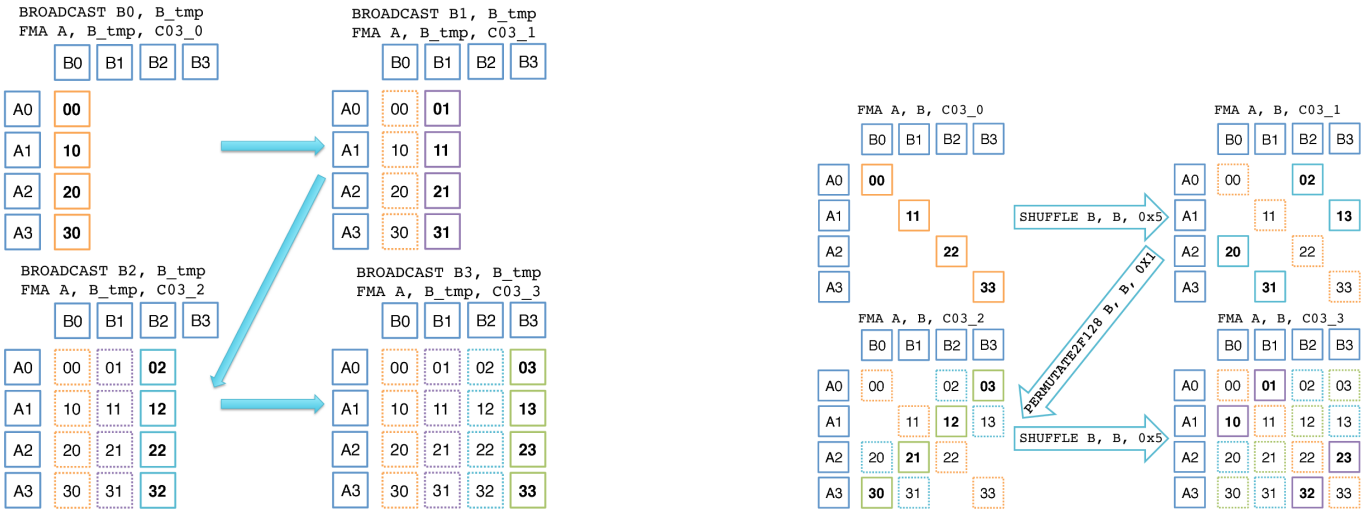
- $[4 \times k \times 4]$: load 1 vector from A, broadcast 4 scaler from B, and store result in 4 vectors of C.
- $[8 \times k \times 4]$: load 1 vector from A, broadcast 8 scaler from B, and store result in 8 vectors of C.
- $[8 \times k \times 8]$: load 2 vectors from A, broadcast 8 scaler from B, and store result in 16 vectors of C.

2.1.5 butterfly micro-kernel

The butterfly micro-kernel also leverages arm-ave instructions. This micro-kernel implemented the method described in lecture and shown in Figure 3b. Butterfly micro-kernel is inherit $[4 \times k \times 4]$ unrolled.

The 16 offsets (dependent on n) are pre-computed once, and store at a static location. The micro-kernel loads the offsets and uses `svld1_gather_s64offset_f64` and `svst1_scatter_s64offset_f64` to load and store C vectors.

In contrast to the AVX instructions used in the Figure 3b, `svrev_f64` and `svext_f64` are used to generate all 4 combinations.



(a) broadcast micro-kernel (b) butterfly micro-kernel

Figure 3: Micro-kernels for broadcast and butterfly operations

2.2 Development process

2.2.1 develop feature

Features are developed in following order: packing, broadcast micro-kernel, padding, unroll broadcast micro-kernel, and butterfly micro-kernel. In the process, compiler keywords, such as `#define`, `#ifdef`, are used to ensure backward compatibility. And features can be turned on or off quickly by modifying config file.

2.2.2 pick micro-kernel

After $[4 \times k \times 4]$ unroll broadcast micro-kernel and butterfly micro-kernel are implemented, a simple comparison between the two is conducted. I noticed: 1) $[4 \times k \times 4]$ unroll broadcast kernel is faster than the butterfly kernel; 2) unrolling produced the most performance gain; 3) more aggressive unrolling can be easily implement on broadcast kernel. So I further implemented $[8 \times k \times 4]$ and $[8 \times k \times 8]$ unrolling on the broadcast kernel, and prioritize optimize hyper-parameters for this type of kernel.

It is worth note that the $[8 \times k \times 8]$ kernel already occupies 19 vector registers at minimum to void ping-pong effect. Kept increasing kernel size and more unrolling may not further improve the performance.

2.2.3 hyper-parameter search

Based on the number of double float each cache can hold (obtained via `lscpu`), The heuristic of GoTo routine requires hyper-parameters satisfy following equations:

$$
\begin{aligned}
m_r \times n_r &\leq \#\text{ of vector registers} \\
m_r \times k_c &\leq L_1 = 64KiB && = 2^{13}\,(\text{double}) \\
n_c \times k_c &\leq L_2 = 1MiB && = 2^{17}\,(\text{double}) \\
k_c \times m_c &\leq L_3 = 32MiB && = 2^{22}\,(\text{double})
\end{aligned}
$$

With m_r, n_r set to 8, and to keep some free space for instructions and lower level caches, initial hyper-parameters are set to $k_c = 512, n_c = 64, m_c = 1024$. An auto-search python script at `auto.py` is developed for finer parameter search. However, due to time limits, the script only searched 125 parameter combinations around the initial value. The results shows their performance are very close, and the initial values are used as final hyper-parameters.

2.3 Irregularities in the data

2.3.1 slow init

As shown in Figure 1, fancy methods are slow on small matrix multiplications, such as $N = 32$, compare to non-padded methods; This is largely because those fancy operations involves fixed cost: pay upfront and benefit the program in the long run.

2.3.2 slow N=1025

As shown in Figure 1, padding methods have a performance drop at $N = 1025$ compare to $N = 1024$. Since $m_c = 1024$, there are twice macro-kernel call in $N = 1025$ compared to $N = 1024$. In addition, $N = 1025$ probably caused packing and padding encountering more cache misses and page fault, since the data are not aligned with the typical power of 2 cache and memory page sizes.

2.3.3 slow butterfly

Butterfly kernel involves fewer memory access than broadcast kernel, which, in theory, should yield higher performance. However, the experiment shows the other case. The reason might be the matrix `C` is not packed, making gather and scatter instructions accessing memory locate at very far distances, thus slower than consecutive load and store.

2.4 Future Work

2.4.1 packed `C` for butterfly kernel

Since the padding method need to copy result matrix `C` once at the very end, it may be beneficial to store the intermediate padded-`C` in a butterfly friendly layout, and convert back to normal row major layout when unpad `C`.

2.4.2 more auto tuning

I am not sure how ec2 virtualize the CPU hardware and manage caches, so the hyper-parameters based on `lscpu` may not be best. The hyper-parameter selection would be more robust if the auto-tuning script can explore more parameter space.

2.4.3 multi-core parallel

The GoTo scheme only leverages single core. It would be interesting to implement multi-core parallel.

References

[1] Jianyu Huang and Robert van de Geijn, *BLISlab: A Sandbox for Optimizing GEMM*, 2016.

[2] ARM, *C Language Extensions for SVE*, https://developer.arm.com/documentation/100987/latest/

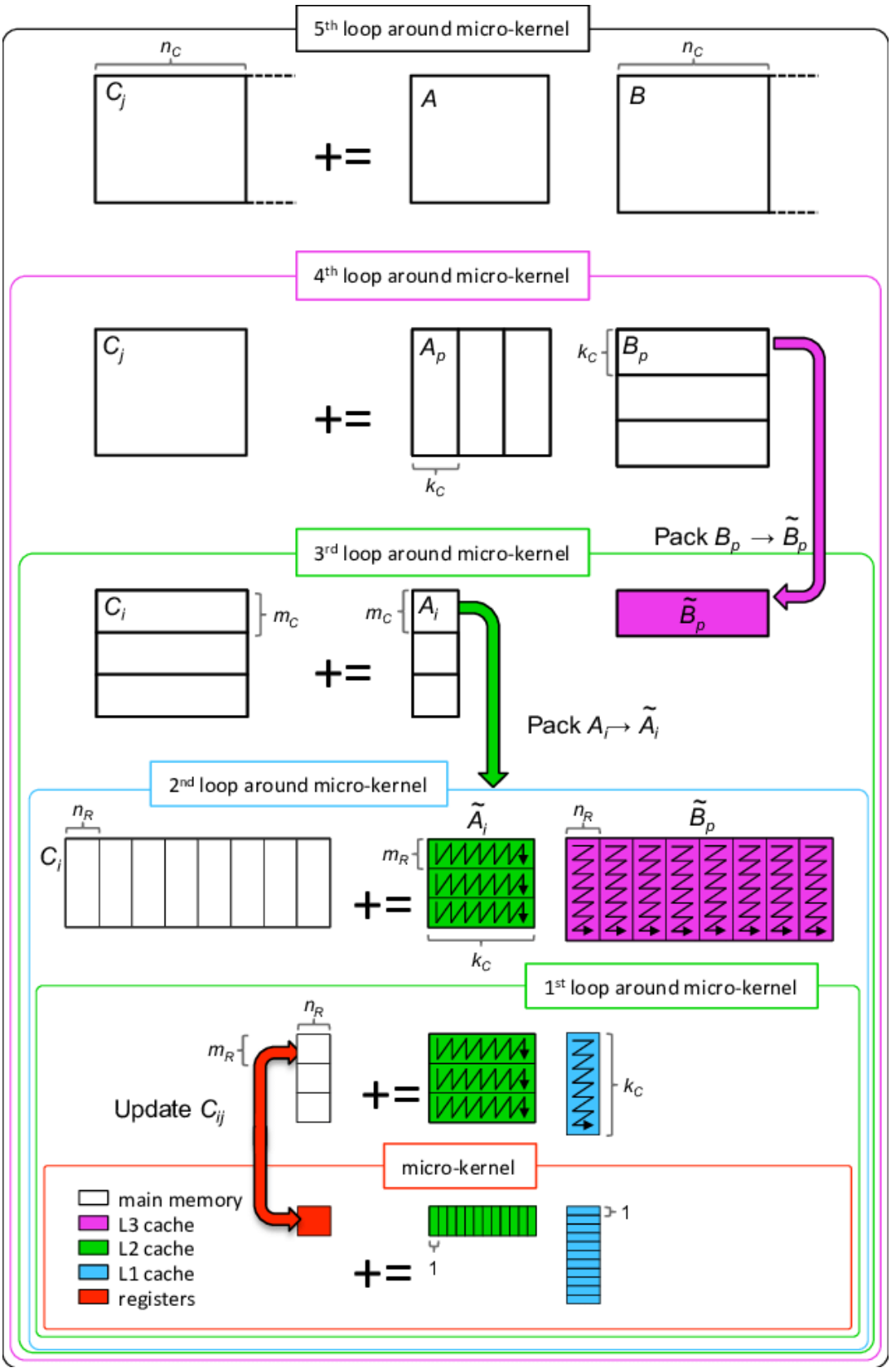


Figure 2: GoTo method