CSE 260

PA1 Report

Yijian Liu, Yue Pan

10/17/2022

Q1. Results - 15 pts

Give a performance study for a few values (about 12 different values) of N from 32 to 2048 on your optimized code both in Q1.a. and in the file data.txt (see "what to submit->data file"   for specific format. You will lose points if you do not follow this format.)

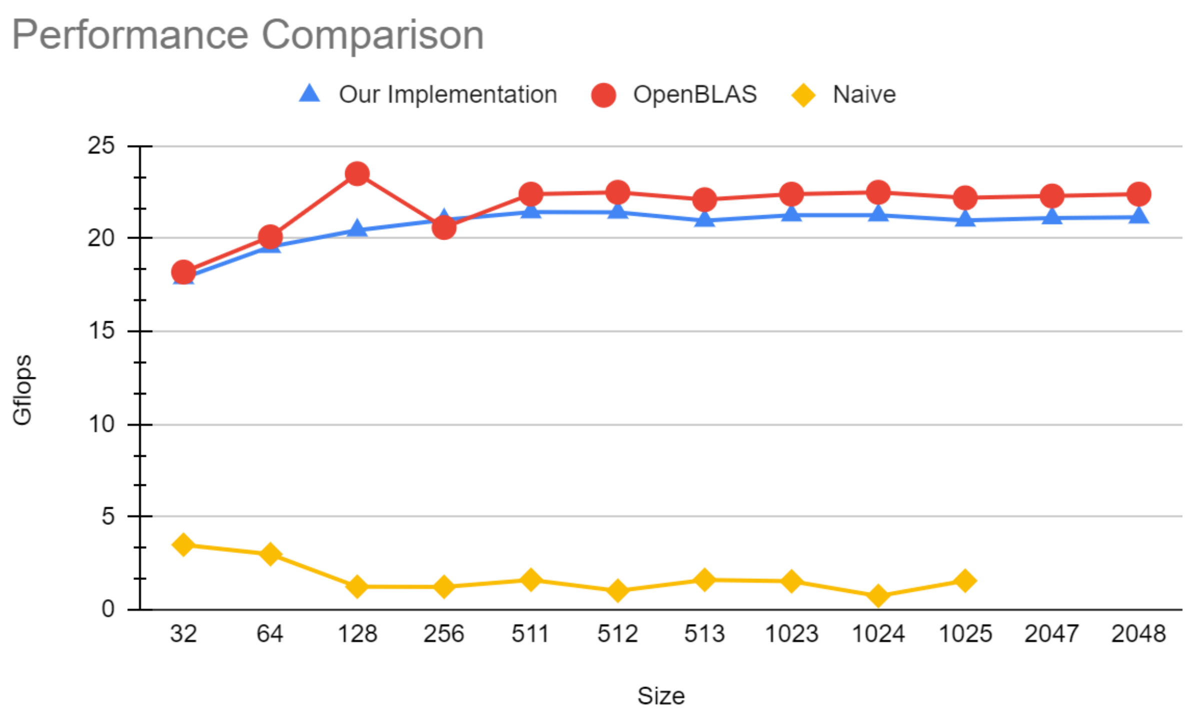
Q1.a. Show performance of your optimized code for the following numbers (fill out the table):

N	Peak GF
32	17.885
64	19.565
128	20.455
256	21.01
511	21.43
512	21.415
513	20.975
1023	21.265
1024	21.265
1025	20.985
2047	21.11
2048	21.15

Q1.b. Make a plot of the performance of the three versions of code: the naive code, the OpenBLAS code, and your optimized code. OpenBLAS and your optimized code should include all N values from the table. The naive code only has to include N <= 1025.

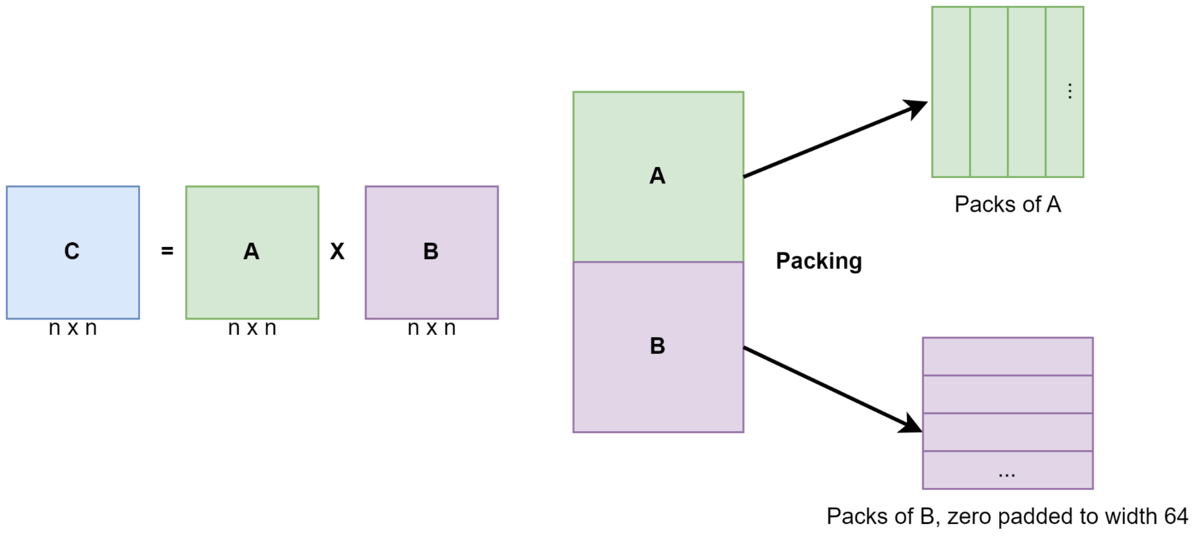
Average performance

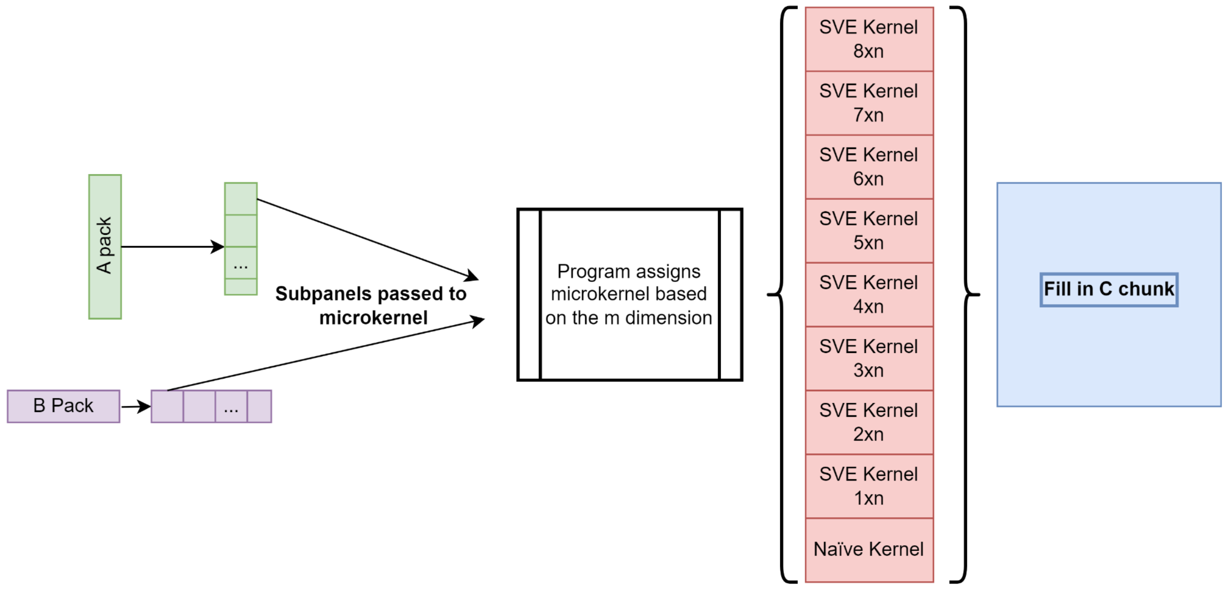
Our implementation:	20.68
OpenBLAS:	21.72
Naive:	1.54



Q2. Analysis - 33 pts

Q2.a. How does the program work - **don't include the source code**, instead describe it in prose, flow chart, pseudo-code, etc.





Q2.b. Development process: What did you try, what worked, what didn't work, theories on why. Negative results are sometimes as illuminating as positive results, so try to explain as best as you can. Include the necessary graphs for the optimizations implemented. Use your github check-ins to show your development process (e.g. commit logs).

The development process starts with understanding representations of different matrix dimensions. Initially, the performance of our program was at about 3 to 4 on average. After we moved onto the implementation of packing.

After packing, we verified the correctness of packing with the C-based naive kernel, which applies a 3-level for loop on the incoming subpanel.

Then, we moved onto designing the optimized microkernel using SVE intrinsics. We first designed the microkernel to be 4x4 for the C subpanel. At this stage, the program would use the microkernel if the subpanel happens to be 4x64, otherwise it would simply apply naive matrix multiplication. Using SVE intrinsics, we were able to leverage the cpu and achieve around 16 gflops/s.

After confirming the 4x4 kernel to be working, we began tweaking the sizing macros (`DGEMM_KC, DGEMM_MC, DGEMM_NC, DGEMM_MR, DGEMM_NR`), intending to better utilize the available cache space. By running `lscpu`, we learned that the caches for the Neoverse-V1 processor are,

L1d:	64 KiB, 256 sets, 4-way associative, 64B cache line
L1i:	64 KiB (irrelevant to us)

L2: 1 MiB, 2048 sets, 8-way associative, 64B cache line
L3: 32 MiB, 32768 sets, 16-way associative, 64B cache line

By simple calculation, we can get how many double precision float numbers we can fit into the cache. For L1 data cache, number of doubles that may present = 64KB / 8B per double = 8192 doubles. For L2 cache, the number is far greater at 131072 doubles. Therefore, the original subpanel dimensions are far from fully utilizing the L1 and L2 caches. An analysis of how DGEMM sizes are influential to the performance can be found in Q2.c

In the ideal case, the optimized microkernel will receive A subpanel in size 8x64, and B subpanel in size 4x64. This will produce a subpanel of C in size 8x4 by the optimized kernel. However, matrices often cannot be evenly sliced into the most efficient chunk size, and the remaining irregular panels will be processed less efficiently. To support B subpanels of various sizes, we pad all B subpanels with 0 so that all B subpanels have the same MR row length, which is the ideal size supported by the optimized kernel. As a result, the SVE kernels effectively handles both 4xn and 8xn output size, where n is the row length of B subpanel.

As a final major speed-up of our project, we introduced different microkernels sizing from 8xn to 1xn. Once a pair of A and B subpanels are passed in, the if statements will check for their sizes and choose the appropriate kernels so that SVE instructions are used as much as possible.

Q2.c. Point out and explain at a high level irregularities in the data (Places where performance scales in a non-linear way) - referring to your graph in Q1.b.

Microkernels are very efficient. In our implementation, regular (divisible by 4) microkernels are more efficient than others. In addition, 8xn kernels are slightly faster than 4xn kernels because the overhead of moving data is amortized over more calculation. Although the curve is relatively smooth with greater n, we can see slightly lower performance on those matrices whose dimensions are not divisible by 8, for example 31, 63 all the way to 1023, 1025. For example, with a matrix of size 31*31, 75% of the kernels calls are handled by the 8xn kernel and 25% are handled by the 7xn kernel, which is less performant. As the matrix size becomes larger, impacts of irregular kernels become less prominent because a smaller portion of subpanels will be processed by kernels of smaller size.

Q2.d. Supporting data

Parametric search of different DGEMM sizes

Here we explore the design space of DGEMM macros: KC, MC and NC. First, we look at changing KC while fixing MC and NC to be 64.

KC = MC = NC = 64, 18.64 GFlops
KC = 128, MC = NC = 64, 19.56 GFlops
KC = 256, MC = NC = 64, 19.45 GFlops
KC = 512, MC = NC = 64, 19.56 GFlops
KC = 1024, MC = NC = 64, 19.36 GFlops

This suggests a larger KC is beneficial to the overall performance of the program. Once KC passes 128, the performance appears to be stable around 19.5 GFlops.

Then, we fix KC at 512 and look at how MC and KC may influence the performance.

KC = 512, MC = NC = 64, 19.56 GFlops
KC = 512, MC = NC = 128, 19.91 GFlops
KC = 512, MC = NC = 256, 20.48 GFlops
KC = 512, MC = NC = 512, 20.47 GFlops
KC = 512, MC = NC = 1024 19.87 GFlops

The experiments suggest that MC and NC at 256 and 512 yield the best performance. After numerous trials, we found that the KC = 512, MC = NC = 256 combination appears to give slightly more stable performance in our implementation. Further increasing MC and NC results in drops in suboptimal performance, potentially because they become too large for the cache to efficiently store. Note that this does not mean that the cache capacity cannot hold all these doubles, but they may cross cache line boundaries or incur conflict misses, which may negatively impact the performance.

Cache Behavior

In our implementation, we expect the A and B subpanels to reside in L1 and L2 cache, respectively. We made such a guess based on the loop order when passing A and B subpanels to the microkernel. In the outer loop, we increase the index for A subpanel, and in the inner loop, we increase the index for B subpanel. Therefore, in the loop body, the same A subpanel is passed to the microkernel every time, while B subpanel iterates through the pack. So, it makes sense for the more frequently accessed A subpanel to stay in L1, and the intermittently accessed B to reside in L2.

Moreover, by designing the mostly used kernel size to take A subpanel in size 4x64, B subpanel in size 64x4. We took advantage of the cache line size of 64B – 8 doubles in a single cache line. With spatial locality, we expect that every row or column of the subpanels can fully occupy a few cache lines. Furthermore, we expect that some B subpanels can also be stored in the L1 cache since L1 is 4-way associative: there may be a chance of a subpanel of B not getting evicted by the replacement policy of the cache.

Q2.e. Future work - what could you do if you had more time?

We certainly could have looked into SVE intrinsics further given more time. For example, the way we implemented different size microkernels is done explicitly: a series of if statements checks for the dimension of incoming pack, and the pack gets assigned to the kernel with matching dimensions. If there isn't a matching microkernel, we use naive matrix multiplications to handle it. With better understanding of SVE intrinsics, such as using predicates, the code will appear more concise and will probably have a slightly better performance.

We could also better study the cachegrind tool given more time. Currently the tool cannot run and suggests a segmentation fault.

Q2.f. (Optional) Any additional insight or optimizations that you tried implementing and how it affected the performance.

Switch-case versus if-else clauses

Because a multi-level if-else block determines which microkernel will be used to process the incoming subpanel based on its m dimension, we tried both switch-case and if-else for this logic. Theoretically, switch-case slightly is faster than if-else because the jump location (which microkernel to use) is determined at constant time (assuming a hash-table like implementation). In contrast, if-else statements require multiple evaluations for the code to fall into the correct chunk.

However, after using a switch case in our project, we noticed that the performance is worse (by ~0.5GFlops). There could be two reasons for this. For one, with compiler flag -O4, if-else is well optimized, potentially more so than switch-case because it is a more prevalent logic in all codes. Another reason could be that since in the majority of cases the subpanel will be processed using the 8xn kernel, the branch predictor is very good at hiding the latency of if-else evaluations. Yet switch-case should benefit from the same branch predictor. We are uncertain why this should happen.

Do-while versus for loop

In class, we discussed that do-while loops should be slightly faster than for loops, because do-while loops incur less branch instructions, only one at the end of the loop body. For loops have two, one for checking the loop condition, and one jumping back to the start of the loop body at the end. We found this difference is around 0.1 GFlops.. Considering the difference between 1 vs 2 branch calls in a thousand-line loop body, it makes sense that there's only a little difference between the two implementations.

<u>Restrict pointers</u>

By ensuring the compiler that a memory address will be pointed to by only one pointer variable, we are able to receive quite significant speedups. In compiled assembly, a restricted pointer helps save another memory read before each subsequent write after the first one. For example, a function without restrict pointer as arguments,

```
void foo_restrict(int* a, int* b) {
     *b    = *a + 3;
     int d = *a + 4;
}
```

will be compiled into assembly language roughly as

```
Read a
Add 3 to a and put in b
Read a again
Add 4 to a and put in d
```

With restricted pointers, the second `Read a` will not be present. Because the compiler is certain that a and b passed in will not be pointing to the same address, therefore a will not have changed after the first line. If a and b are indeed given the same address, the program results in undefined behavior.

The benefit from restrict pointers is most noticeable with small matrices. In the naive implementation, we received about 0.5 GFlops improvements on small matrices. As the code became more optimized later, we found the benefit from restricting pointers to be insignificant and hard to realize. We believe O4 optimization has contributed to much of the available speedup even without restricted pointers.

Q3. References - 2 pts

1. https://github.com/flame/blislab - Jianyu Huang, Robert A. van de Guijn, BLISlab: A Sandbox for Optimizing GEMM, August 31, 2016

2. Interactive ARM SIMD reference pages
https://developer.arm.com/architectures/instruction-sets/intrinsics/
3. John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. A family of high-performance matrix multiplication algorithms. In Vassil N. Alexandrov, Jack J. Dongarra, Benjoe A. Juliano, Ren´e S. Renner, and C.J. Kenneth Tan, editors, Computational Science
4. K. Goto, R, Geijn, "Anatomy of high-performance matrix multiplication", ACM Transactions on Mathematical Software, May 2008
https://dl.acm.org/citation.cfm?id=1356053