CSE260 -PA1

DEBADITYA BASU , MONIL SHAH

GIT REPO : hw1-pa1-dbasu-m3shah

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.)

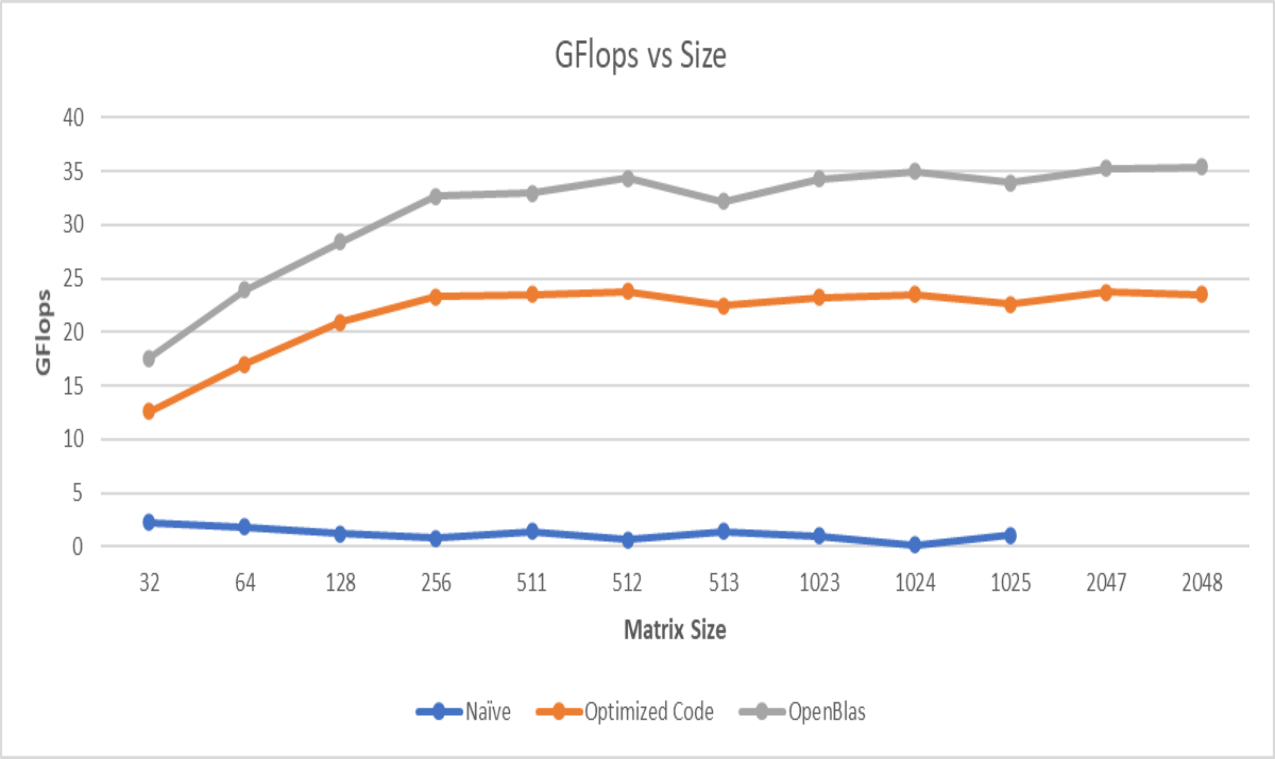
Q1.a. Show performance of your optimized code for the following numbers (fill out the table):

GIT REPO : hw1-pa1-dbasu-m3shah BRANCH NAME : main

N	Peak GF
32	12.585
64	16.995
128	20.895
256	23.33
511	23.525
512	23.815
513	22.46
1023	23.21
1024	23.49
1025	22.605
2047	23.73
2048	23.48

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.

BRANCH NAME : main



Q2. Analysis - 33 pts

Clearly Describe:

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

Solution:

Input Matrix:	**Output Matrix:**
MATRIX A : m * k	MATRIX C : m * n
MATRIX B : k * n	

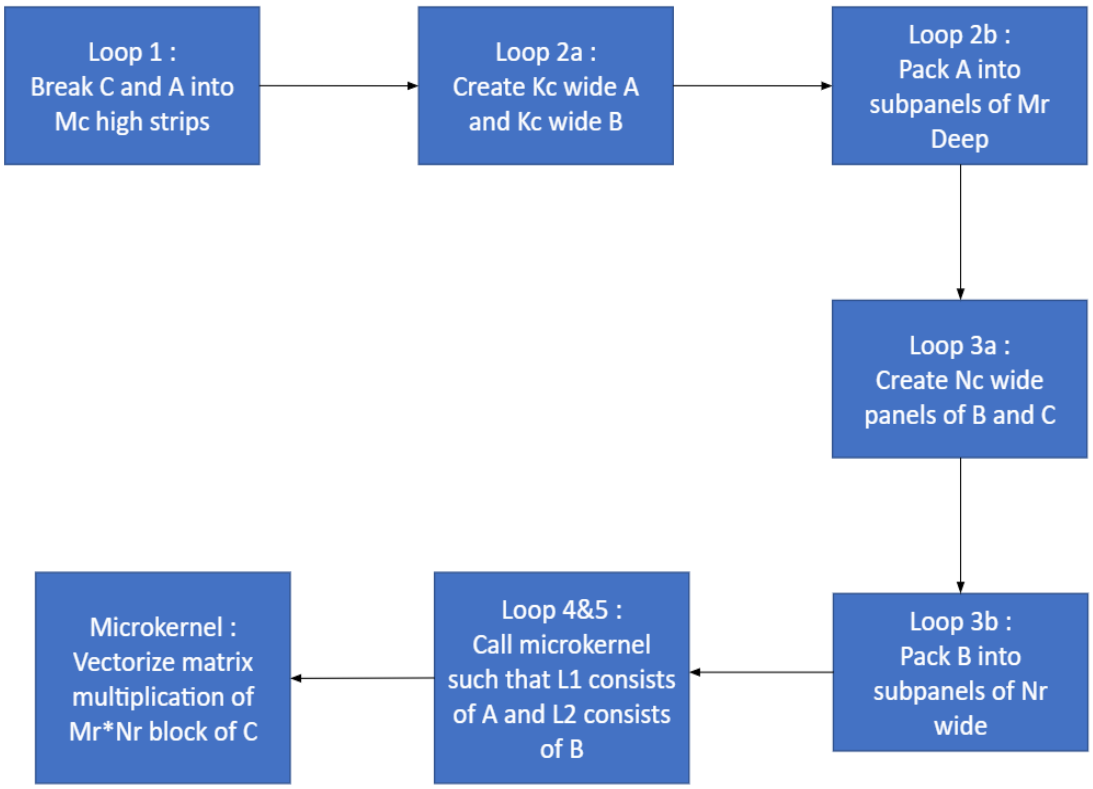
1. The first loop breaks C matrix into panels of Mc*n and A matrix into panels of Mc*k.

2. Second loop converts the panels created above into Kc wide panels. This is done for A matrix (Mc*Kc panels). For B the stripes are created Kc high(Kc*n panels)
   - ➢ Next panels of A are packed into subpanels of Mr height(Mr*Kc) in row major order of accesses in case of outer product. The column elements become sequential in memory.

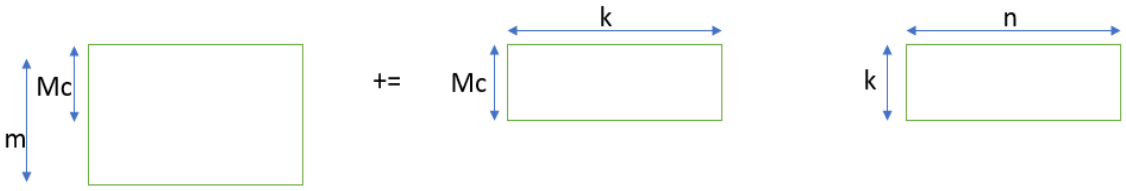
3. Third loop converts C and B into panels that are Nc wide. This is done for C matrix (Mc * Nc panels) and B into (Kc*Nc panels).
   - ➢ Next panels of B are packed into subpanels of Nr wish t(Kc*Nr) for cache friendly access. The row elements are sequential in memory.

4. The inner two loops are designed to call the microkernel such that A panel fits into L1 and B panels fit into L2.
   - ➢ A matrix panels are further subdivided into subpanels of Mr*Kc.
   - ➢ B matric panels are further subdivided into subpanels of Kc*Nr.

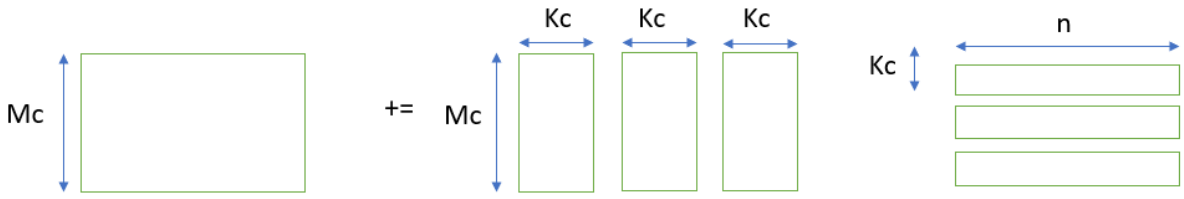
5. This is for the microkernel implementation which implements vectorization of Mr*Nr block of C matrix multiplication



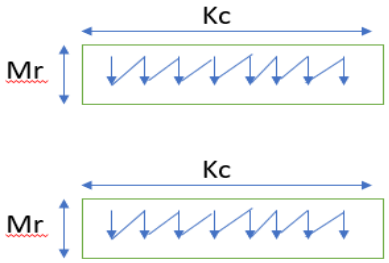
LOOP 1:



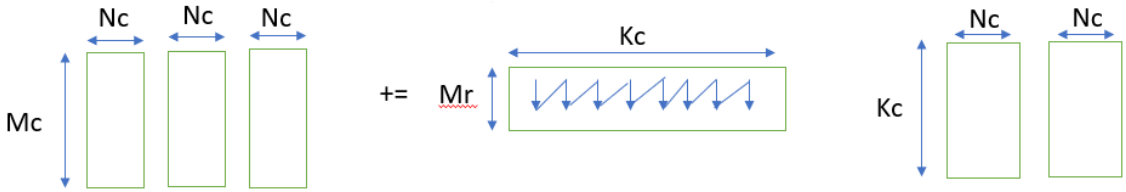
LOOP 2a:



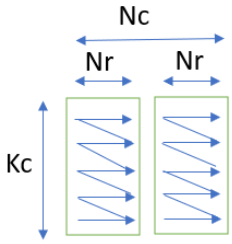
LOOP 2b:



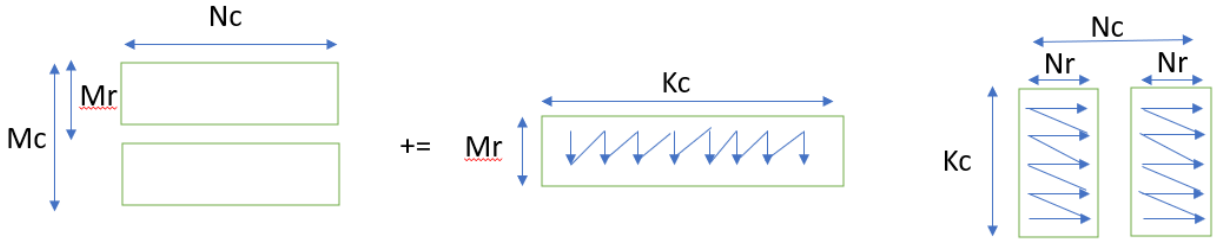
LOOP 3a:



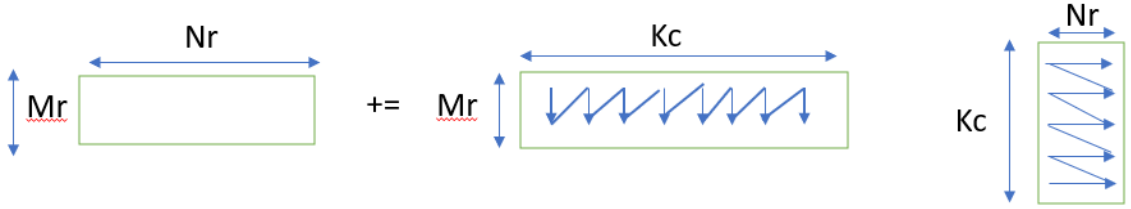
LOOP 3b:



LOOP 4 & LOOP 5



Microkernel



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.

1. **Implementing Packing and Microkernel**

   ➢We first tried to get packing work. To debug if packing was fine we implemented an outer product based microkernel.

   ➢While running different size matrix we noticed, that the matrix multiplication used to fail if the size of matrix was not multiples of Mr or Nr.

   ➢We noticed that our padding of 0 was wrong and once we fixed that with necessary changes to the macro-kernel and implemented the microkernel accordingly we got the first performance mean as **2.79 Gflops/s**.

2. **Vectorization**

   We tried to implement AVX2 based vectorization and changed the micro-kernel accordingly.

   ➢ We first tried with implementing broadcast method that used 4 256 bit registers for both C and A matrix sub panels and 1 256 bit for B matrix sub panel.**(The value would be Mr=4 and Nr=4)**
   By using 9 256 bit registers in total, we got performance of **12.20 Gflops/s**

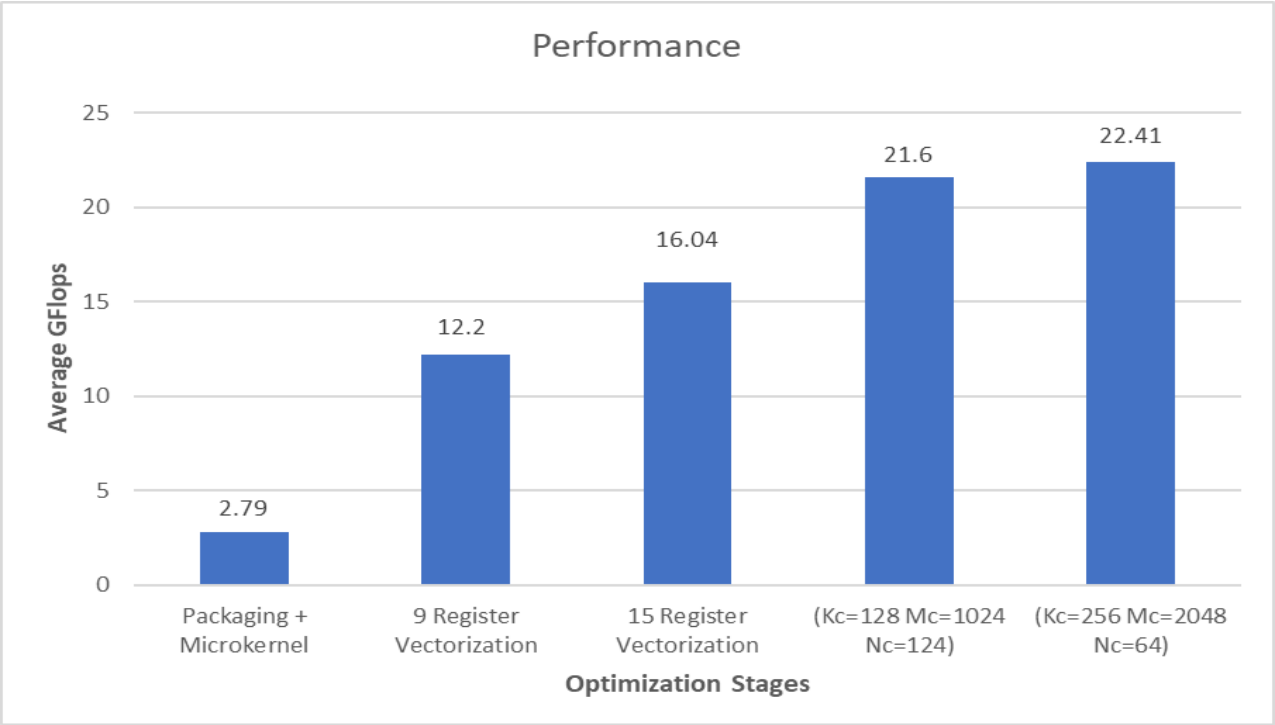
   ➢Since we have 16 256bit registers available, to maximize the advantage of AVX2 based vectorization we implemented broadcasting with 7 256 bit registers for both C and A matrix sub panels and 1 256 bit for B matrix sub panel. This way we were able to use 15 of the available 16 256 bit registers. **(The value would be Mr=7 and Nr=4)**
   By using 15 bit 256 registers, we achieved performance of **16.04 Gflops/s.**

3. **Tuning Parameters of Kc,Mc,Nc**

   Next we tried to tune the parameters of Kc,Mc and Nc to best utilize the cache sizes and tried a few parameters to arrive at the parameters which gave us the best performance.

   ➢ After changing the parameters to **(Kc=128 Mc=1024 Nc=124)** we got performance of **21.60 Gflops/s.**

   ➢ After further tuning the parameters to **(Kc=256 Mc=2048 Nc=64)** we got performance of **22.41 Gflops/s, which was the highest we got.**



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.

1. **Smallest Matrix Size have the least performance**
   Since the caches are not utilized to its full capacity, the blocking followed by padding of 0s for packaging do not optimize and add a lot of useless calculations which results in a lot of overheads.

2. **Drop in Performance of sizes that are 1 more than Even Powers**
   The reason for the drop from Even powers to the next size is because of extra padding that we have to introduce to make sure that computation is correct. This results in a lot of unnecessary calculations for the zero padded data and reduces performance.

3. **Performance is highest at the size of 512**
   This is likely because the speedup that we are able to achieve through caching is diminishing at higher sizes because of capacity misses. Cachegrind tool shows that size of **512** as having the lowest capacity miss rate at **5.2%** and **513** and **1024** having miss rate as **5.4%** and **6.1%** respectively.

4. **Same trend among Blas and Optimized Code Performance**
   From the figure it can be seen that Blas and our Optimized kernel is having negative and positive slopes of peak performance in the same direction
   Only transition from 256 -> 511 size is in opposite directions

Q2d. Supporting data - e.g. analysis of cache behavior, parametric searches, or whatever will support your conclusions. Feel free to use tools such as cachegrind and knowledge of the machine's micro-architecture to support your theory. Note: cachegrind is slow therefore it is ok that you only measure a subset of n. Explain why we organized our skeleton code in a different way from the BLISlab tutorial.

The L1 sizes are 32KB and L2 size is 256KB. This gives us the estimate of how much Mr, Kc and Nc to keep so that they fit sufficiently within the cache

L1 Cache is expected to have both subpanel of A and subpanel of B to perform the matrix multiplication by Micro-kernel..

Microkernel operates on subpanel A = Mr * Kc ;
Microkernel operates on subpanel B= Kc* Nr

By calculation assuming 32KB = Kc*(Nr+Mr)
Kc is approximately 372
But since we cannot use full L1 for just storing subpanel A and B as L1 must store other things as well we can
narrow down to the nearest power of 2 which is 256.
Thus Kc = 256, resulting in sub panel A and sub panel B occupying 22KB out of the 32KB allowed which is fair.

L2 Cache should be able to store panels of B as subpanels of Bp are moved in and out of the L1 from L2.
Bp panel is Kc*Nc.
Since L2 Cache has both instruction and Data instructions, assuming half is for data. Available Size is 128KB.

Assuming 128KB is used to store Kc*Nc Bp panel we get Nc as 64.

Based on the following inputs we used Cachegrind to get some data and eventually arrived with **(Kc = 256 ; Nc = 64 ; Mc = 2048)**

Cachegrind results:

CONFIG	**SIZE**	**L1 Miss Rate**
(Kc = 256 ; Nc = 64 ; Mc = 2048)	**2048 | 1024 | 512**	**6.3% | 6.1% | 5.2%**
(Kc = 256 ; Nc = 64 ; Mc = 1024)	**2048 |1024 | 512**	**6.5% | 6.1% | 5.2%**

The BLISlab tutorial code is compliant with the column major storage principle as it was implemented for FORTRAN programmers. Since we are using C, which is row major storage based, our skeleton becomes different.

C : $C_{xy} = A_{(x\text{-row})} * B_{(y\text{-column})}$
FORTRAN : $C_{xy} = A_{(x\text{-column})} * B_{(y\text{-row})}$

BLISlab operates upon the inputs received as Transpose of the actual input matrices, hence the skeleton code becomes different.

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

1. We wanted to explore and see if there are any compiler hints that we can add to the code like prefetching something while the thread is busy doing something else.
2. We also want to see if we use vectorization with 4 256 bit registers for A and 2 256 bit registers for B and 8 256 bit registers for C. Though it will use 14 16 bit registers in total which is 1 less than what our current implementation has , but we could have utilized it for performance gain as Mc would have been a multiple of Mr.
3. We wanted to observe the objdump and see what best configuration we can keep so that registers don't spill on the stack.

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

1. We tried to remove as many redundant instructions as possible. Earlier we were using some additional variables, but later to get speedup we moved to removing those additional variables.

Q3. References - 2 pts
List all your references here.

- UCSD CSE 260 SP22 Lecture Slides
- https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=AVX2
- https://github.com/flame/blislab
- https://www.cyberciti.biz/faq/check-how-many-cpus-are-there-in-linux-system/

Q4. Extra Credit - 5 pts
(5 pts) Q4.a. Report performance for AVX512 (you may use the table from Q1) below. Also please add a note in a few sentences describing your AVX512 implementation (only describe significant differences from the AVX2 implementation).

BRANCH NAME : avx512
Implemented AVX512, here are the results:

N	Peak GF
32	17.225
64	25.685
128	31.715
256	32.96
511	31.36
512	32.285
513	30.42
1023	33.495
1024	34.925
1025	33.06
2047	36.795
2048	36.57

Difference with AVX2 are as follows:
With AVX512 we are able to perform more operations in the microkernel.
We are using 8 512 bit registers for both A and C and 1 8 bit register for B.
So in microkernel we are able to do 64 calculations in a single call.
(**Mr = 8 and Nr=8**)
In our AVX2 implementation we could use only 15 256 bit registers and perform 28 calculations in a single microkernel call. **(Mr=7 and Nr=4)**

(5 pts) Q4.b. Parallelize your code with OpenMP and report your results and conclusions. See BLISlab tutorial Step 4 for instructions on how to do this.

(5 pts) Q4.c. Implement your algorithm (including the SIMD optimization) on ARM ISA. You could use NEON, which is a SIMD ISA on ARM similar to AVX on x86 architecture. Make sure to select the ARM version of Linux on AWS. We suggest choosing t4g.micro as your instance type.