Cpu GEMM

Authors: Khaled Matar | Yousef Mohammed

Macbook pro M3

L1 instruction cache: 192KB (hw.perflevel0.l1icachesize: 196608)
L1 data cache: 128KB (hw.perflevel0.l1dcachesize: 131072)
L2 cache: 16MB (hw.perflevel0.l2cachesize: 16777216)

How we validate

By using numpyt to write the output of `A @ B` to `tmp/matmul` we can read the and compare the output of our GEMM

How we measure improvements

Gflops

our base metrics we will comparing with is Numpy with OpenBlas, however Numpy has the copability to do gemm with CBlas and Apple Accelerate framework.

Run commands and flags

`-Xpreprocessor -fopenmp -O3 -march=native -mcpu=native -ffast-math`

Naive implementation

Python

```
for i in range(N):
    for j in range(N):
        for k in range(N):
            C[i][j] += A[i][k] * B[k][j]
```

average Gflops: 0.015153
N: 256

cpp

```
void gemm_stupid(){
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < N; k++)
                C[i][j] += A[i][k] * B[k][j];
```

```
        }
}
```

average Gflops: 3.738789
N: 256

Numpy OpenBlas (single threaded)

To confirm that Numpy is using OpenBlas we can view the output of `print(np.__config__.show())` and we see the output inlucde `name: scipy-openblas`. depending an how we installed numpy to the venv we would sometime get `Accelerate` or `cblas`.

We use the library threadpoolctl to force numpy into only using a single thread as shown below.

```
from threadpoolctl import threadpool_limits

with threadpool_limits(limits=1):
    pass
```

We use numpys `@` operator to do the matrix multiplication.

It is also important to note that these matricies are initialized as type float32 as shown below. This is to give numpy a fair chance since we are using float in Cpp

```
A = np.random.randn(N, N).astype(np.float32)
B = np.random.randn(N, N).astype(np.float32)

with threadpool_limits(limits=1):
    for i in range(itr):

        start = monotonic()
        C = A @ B
        end = monotonic()
        s = end - start

        gflops = (N*N*2*N) / (s * 10**9)
        avg += gflops

        print(f"GFLOPS: {gflops:.6f}")

    print(f"avg: {avg/itr:.2f}")
```

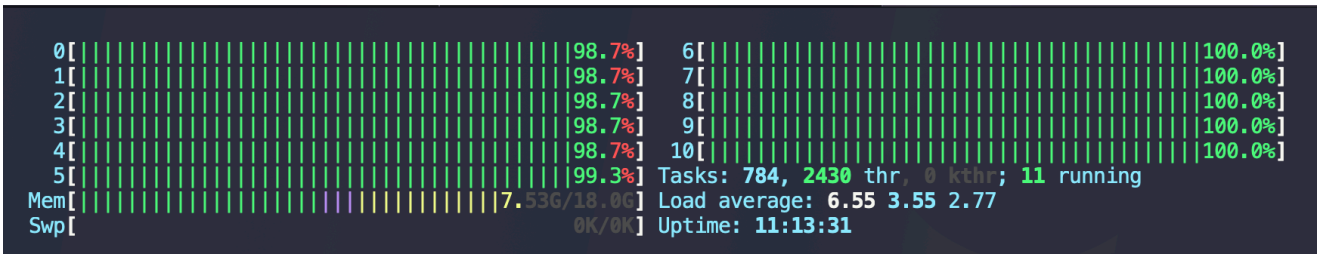
average Gflops: 43.073460
N: 2048

Numpy (multi threaded)

Numpy by default is multithreaded so we will be testing out how good default numpy is by removing
`with threadpool_limits(limits=1):`

We can also confirm that this is multithreaded by checking `htop`
This gives us 173.38 avg Gflops with N = 2048

Lets also confirm that this is multi threaded.



Here is the current table

Descritiopn	Gflops	N
Python Stupid	0.015	256
C++ Stupid	3.738	256
ST_Numpy	43.073	2048
Numpy	173.38	2048

Improving CPP: Transposing

By transposing B, the data we fetch should be more continuous, thus reducing our cache misses and being generally more cache friendly.

Having the matrices 1D should also help with cache misses, but with our testing it does not give us any
preformance advantage, rather it makes it a bit slower. So we will stick with 2D arrays.

We think that the -O2 / -O3 might be optimizing the 2D arrays better.

We get an average of 35.651 Gflops for the transposed version, which is a huge imporvement over the non transpose.

Descritiopn	Gflops	N
Python Stupid	0.015	256
C++ Stupid	3.738	256

Descritiopn	Gflops	N
Numpy ST	43.073	2048
Numpy MT	173.38	2048
C++ Transposed	35.651	2048

Improving CPP: Blocking (Loop tiling)

Next we are trying blocking, we will add three outer loops to create a smaller matrix that we focus on, then tiling up as we finish the matrix.

```
void gemm_omp() {
    int bi, bk, bj, i, k, j;

    for(bi = 0; bi < N; bi += BLOCKSIZE)
        for(bj = 0; bj < N; bj += BLOCKSIZE)
            for(bk = 0; bk < N; bk += BLOCKSIZE)

                for(i = 0; i < BLOCKSIZE; i++)
                    for(j = 0; j < BLOCKSIZE; j++) {
                        for(k = 0; k < BLOCKSIZE; k+=1) {
                            C[bi + i][bj + j] += A[bi + i][bk + k] * BT[bj
+ j][bk + k];
                        }
                    }
}
```

why does blocking even help?

Blocking helps with cache efficiency because it ensures that smaller portions (blocks) of the data fit within the L1 cache, reducing the frequency of cache misses and minimizing memory latency.〔2〕

testing best

blockSize	Matrix Size	FLOP/s
4	2048	31.93 GFLOPS
8	2048	24.00 GFLOPS
16	2048	31.15 GFLOPS
32	2048	25.36 GFLOPS
64	2048	35.90 GFLOPS
128	2048	33.29 GFLOPS
256	2048	24.18 GFLOPS
512	2048	23.18 GFLOPS

It is important to know that his is the minumum we can do with blocking, and there is a lot of room for improvemnt.

For example, we can optimize our tiling based on the cache size, we could also tile X and Y differntly, optimizing for L1 and L2 caches, however this is very complicated. We might look into this if we have more time.

Threading with OMP

Before going over to doing Blocking and Unrolling, we will test out the transpose version with OMP threading.
We are doing this because we think that Blocking and Unrolling will have a much bigger affect once we do gemm with threading.

The simplest OMP we could use is `#pragma omp parallel for` on the outer most loop,

We also did testing on blocksize with threading as it might not be the same.

blockSize	Matrix Size	FLOP/s
4	2048	198.258 GFLOPS
8	2048	132.660 GFLOPS
16	2048	60.810 GFLOPS
32	2048	62.349 GFLOPS
64	2048	169.871 GFLOPS
128	2048	224.167 GFLOPS
256	2048	121.976 GFLOPS

We see here that these numbers are much better than single threaded. It turns out that the cache advantage given by blockign is utilized much more when threaded.

Average flops: 224.167 with block size 128

As we observer here we are very close to numpy, even beating by a couple of Gflops. However, this is not because we are fast, rather it is becuase Numpy with OpenBlas is quite slow. We have beaten Numpy here but we will keep pushing for faster and faster.

Unrolling

Unrolling is very nice idea that will help us remove the inner most loop by unrolling it.

- doing the if statement logic (compiler wise) is expensive

our currently inner loop looks like this

```
..
for (j = 0; j < blockSize; j++)
    for (k = 0; k < blockSize; k++){
        C[bi + i][bj + j] += A[bi + i][bk + k] * B_trans[bj + j][bk + k];
    }
..
```

We have an issue where the inner most if loop is being updated for each entry of `k` and this is expensive since the compiler needs to do extra work, and extra memory usage is done here to fetch and upate `k`

we can reduce this workload by unrolling `x` amount of instruciont and writing them down manually.

for example unrolling by 4 instrucionts would look like this

```
..
for (j = 0; j < blockSize; j++)
    for (k = 0; k < blockSize; k+4){
        C[bi + i][bj + j] += A[bi + i][bk + 0] * B_trans[bj + j][bk + 0];
        C[bi + i][bj + j] += A[bi + i][bk + 1] * B_trans[bj + j][bk + 1];
        C[bi + i][bj + j] += A[bi + i][bk + 2] * B_trans[bj + j][bk + 2];
        C[bi + i][bj + j] += A[bi + i][bk + 3] * B_trans[bj + j][bk + 3];
    }
..
```

However, our testing shows that this barely gives any performace advantage. Neither did unrolling 8 instrctions.

The problem is that we are still using the if statement, so the best scenario would be to unroll all the instrucions and remove the if statement.

the innest most loop runs up to blocksize, meaning there would be the same amount of instrcionts as the block size, so we need to unroll 16 instrucions
to be able to remove the if statement

```
..
for (j = 0; j < blockSize; j++) {
        C[bi + i][bj + j] += A[bi + i][bk + k] * B_trans[bj + j][bk + k];
        C[bi + i][bj + j] += A[bi + i][bk + k + 1] * B_trans[bj + j][bk +
k + 1];
        C[bi + i][bj + j] += A[bi + i][bk + k + 2] * B_trans[bj + j][bk +
k + 2];
        C[bi + i][bj + j] += A[bi + i][bk + k + 3] * B_trans[bj + j][bk +
k + 3];

        C[bi + i][bj + j] += A[bi + i][bk + k + 4] * B_trans[bj + j][bk +
k + 4];
        C[bi + i][bj + j] += A[bi + i][bk + k + 5] * B_trans[bj + j][bk +
```

```
k + 5];
        C[bi + i][bj + j] += A[bi + i][bk + k + 6] * B_trans[bj + j][bk +
k + 6];
        C[bi + i][bj + j] += A[bi + i][bk + k + 7] * B_trans[bj + j][bk +
k + 7];

        C[bi + i][bj + j] += A[bi + i][bk + k + 8] * B_trans[bj + j][bk +
k + 8];
        C[bi + i][bj + j] += A[bi + i][bk + k + 9] * B_trans[bj + j][bk +
k + 9];
        C[bi + i][bj + j] += A[bi + i][bk + k + 10] * B_trans[bj + j][bk
+ k + 10];
        C[bi + i][bj + j] += A[bi + i][bk + k + 11] * B_trans[bj + j][bk
+ k + 11];

        C[bi + i][bj + j] += A[bi + i][bk + k + 12] * B_trans[bj + j][bk
+ k + 12];
        C[bi + i][bj + j] += A[bi + i][bk + k + 13] * B_trans[bj + j][bk
+ k + 13];
        C[bi + i][bj + j] += A[bi + i][bk + k + 14] * B_trans[bj + j][bk
+ k + 14];
        C[bi + i][bj + j] += A[bi + i][bk + k + 15] * B_trans[bj + j][bk
+ k + 15];
    }
..
```

Suprising, this also gave us very little performace advantage, so we played aroudn with the block size and found that a blocksize of 4 (corresponding to 4 unrooled instrctions) gives the best performace

Final loop:

```
..
for (j = 0; j < blockSize; j++) {
    // Fully unrolled for blockSize = 4
    C[bi + i][bj + j] += A[bi + i][bk + 0] * B_trans[bj + j][bk + 0];
    C[bi + i][bj + j] += A[bi + i][bk + 1] * B_trans[bj + j][bk + 1];
    C[bi + i][bj + j] += A[bi + i][bk + 2] * B_trans[bj + j][bk + 2];
    C[bi + i][bj + j] += A[bi + i][bk + 3] * B_trans[bj + j][bk + 3];
}
..
```

using pragma

We could also use `#pragma unrool(4)` to unroll loops, which we be using.

average gflops: 299.380005
N = 2048

Table update:

Descritiopn	Gflops	N
Python Stupid	0.015	256
C++ Stupid	3.738	256
Numpy ST	43.073	2048
Numpy MT	173.38	2048
C++ Transposed	35.651	2048
C++ Transposed, OMP, Blocking=128	224.167	2048
C++ Transposed, OMP, Unrolled (blocking 4)	299.380	2048

Q: We dont fully understand why block size 4 is giving best perfomace on the unrolled loop, we tried larger but got very bad results .

Cache locality

After further invastigation we relised that we can calculate the matrix in multiple ways, initially we were using the ijk approach, but the ikj approach should have a much better cache locallity

We relise that this is the same as transposing B, but are all also fixating on A. Our testing shous that the ikj outperformce the ijk version

```
for i=1..n
  for j=1..n
    for k=1..n
      c[i,j] += a[i,k]*b[k,j]
```

```
for i=1..n
  for k=1..n
    for j=1..n
      c[i,j] += a[i,k]*b[k,j]
```

These implementations are illustrated in figure 1.19 The first implementation constructs the (i, j) element

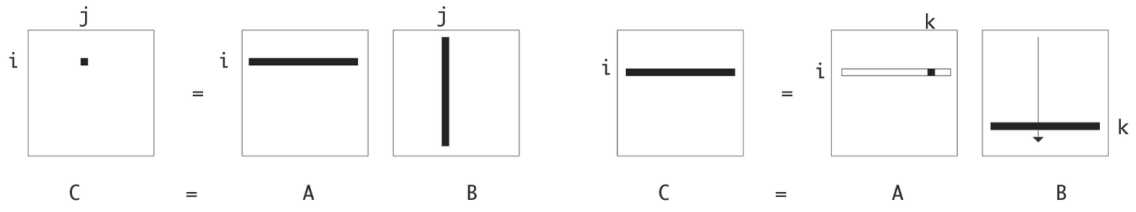


Figure 1.19: Two loop orderings for the $C \leftarrow A \cdot B$ matrix-matrix product.

This also means that we save time by reducing instructions since we do not have to refetch C value when updating it, the A value does not need to be updated.

doing this gives an an impressive boost up to 410.236 Gflop!

Descritiopn	Gflops	N
Python Stupid	0.015	256

Descritiopn	Gflops	N
C++ Stupid	3.738	256
Numpy ST	43.073	2048
Numpy MT	173.38	2048
C++ Transposed	35.651	2048
C++ Transposed, OMP, Blocking=128	224.167	2048
C++ Transposed, OMP, Unrolled	290.167	2048
C++, ikj, OMP, cache locallity, Unrolled	410.236	2048

FMA: NEON, AMX

Fused Multiply-add is is an instruction that can execute X 〈- ax + b in the same amount as doing addtion and the multipcation, this is could potentioally help the processor with asymptotic speed per clock cycle〔1〕

At first we were not sure if -O3 was enabling FMA operations by default, so we tried the C++ Transposed and unroleld version with the `-mno-fma` and got 291.007 Glops, and with the `mfma` we got the same amount.

So the compiler might not be doing this by default.

we added `#pragma omp simd` to help the comlier and got a concistent 5-6 Gflop increase,

```
#pragma omp simd
for (j = 0; j < blockSize; j++) {
    C[bi + i][bj + j] += A[bi + i][bk + 0] * BT[bj + j][bk + 0];
    C[bi + i][bj + j] += A[bi + i][bk + 1] * BT[bj + j][bk + 1];
    C[bi + i][bj + j] += A[bi + i][bk + 2] * BT[bj + j][bk + 2];
    C[bi + i][bj + j] += A[bi + i][bk + 3] * BT[bj + j][bk + 3];
}
```

`#pragma omp simd` and `#pragma vector always` give the perfoamnce

Looking at the assembly code on a cpp explorer (https://godbolt.org/noscript) we can find some mentions of fmadd, suggesting that the compiler is using fma, even thoug hthe speed up of 5-8 Gflops is underwhelming - maybe we are not using its full potential?

Neon on Mac

We can use Neon operations to use FMA, for example we have

`float32x4_t` - 32x4 bits (128) SIMD register
`vdupq_n_f32` - Duplicate single value too all four lanes in the register
`vld1q_f32` - Load four `float32` values from memory into a `float32x4_t` register

`vmlaq_f32` - Performs fused multiply-add
`vst1q_f32` - stores four `float32` values from the SIMD register into memory

We can use `vdupq_n_f32` to duplicate the A value since that needs to be multiplied by all B values.

```
void gemm_omp_ikj_neon() {
    int bi, bk, bj, i, k, j;

    #pragma omp parallel for private(bk, bj, i, k, j) shared(A, B, C)
    for(bi = 0; bi < N; bi += BLOCKSIZE)
        for(bk = 0; bk < N; bk += BLOCKSIZE)
            for(bj = 0; bj < N; bj += BLOCKSIZE)

                for(i = 0; i < BLOCKSIZE; i++)
                    for(k = 0; k < BLOCKSIZE; k++) {
                        // store a_val in the cache since it will be
reused
                        float a_val = A[bi + i][bk + k];
                        // broadcast A value to four lanes
                        float32x4_t a_vec = vdupq_n_f32(a_val);

                        for(j = 0; j < BLOCKSIZE; j+=8) {
                            // four values from A and C
                            float32x4_t b_vec = vld1q_f32(&B[bk + k][bj +
j]);
                            float32x4_t c_vec = vld1q_f32(&C[bi + i][bj +
j]);

                            // FMA
                            c_vec = vmlaq_f32(c_vec, a_vec, b_vec);

                            vst1q_f32(&C[bi + i][bj + j], c_vec);
                        }

                    }
}
```

Neonx2 and Neonx4

We also experimented with loading two and four 32x4 bits.

Howerever we could not get it to work fully porperly and it was never fast enough. The implemntation was quite complicated and there is a high risk we did it wrong.

We used

`float32x4x2_t` - load two `float32x4`
`float32x4x4_t` - load four `float32x4`

AMX

We found the AMX header online, we got the basic operations to work, like load, store*, add** etc howerver we were never able to get the matrix multiplaction to work.

Table after FMA

Descritiopn	Gflops	N
Python Stupid	0.015	256
C++ Stupid	3.738	256
Numpy ST	43.073	2048
Numpy MT	173.38	2048
C++ Transposed	35.651	2048
C++ Transposed, OMP, Blocking=128	224.167	2048
C++ Transposed, OMP, Unrolled	290.167	2048
C++, OMP, cache locallity, Unrolled	410.236	2048
C++, OMP, cache locality, FMA (NEON)	413.167	2048

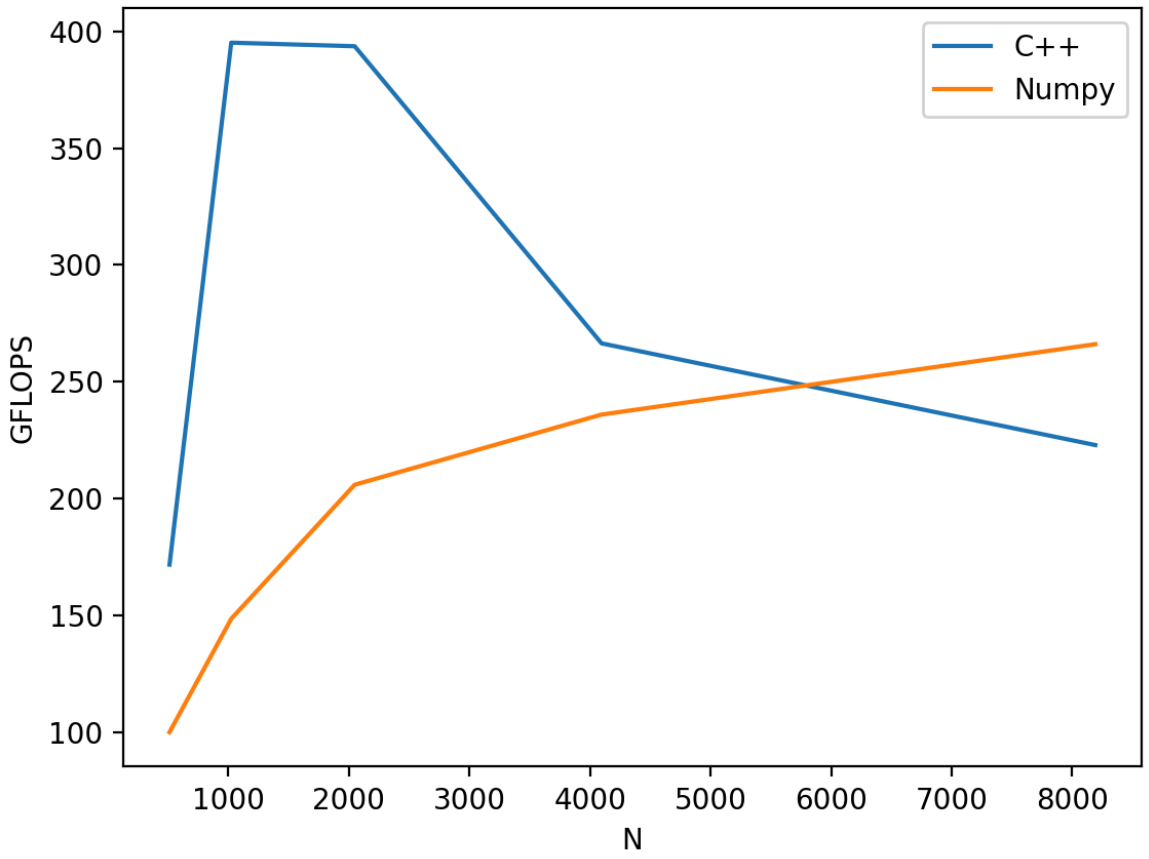
There is seems to be a slight increase in performance but it is very close.

Larger N and memory bound

While we are very close to numpy with what we have, once we scale up N to 4096 and larger, we can see a quite large decrese in Flops.
This could be because we are not optmizing layer 2 cache, and when the matrix is too large, (larger than L2) we start losing performance.

Blocking should be the solution here, but changing the block size does not seem to help us.

A `2048x2048` matrix with `float32` elements is $2048 * 2048 * 4 = 16,777,216$ bytes. The Mac we are running these tests on has 16MB L2 cache, this means once we go over this value we are unable to fin the matrix in the L2 cache



Memory bound

Our Mac has a memory bandwidth of 150 GB/sec, we are getting around 400 Gflops in the peak.
We calculate FLOPs per Byte using the formula:

FLOPs per Byte is calculated as FLOPs/B = Peak FLOPs / Memory Bandwidth. Given 400 GFLOPs (400×10^9 FLOPs/sec) and 150 GB/sec (150×10^9 Bytes/sec), we get $400 / 150 \approx 2.67$ FLOPs/Byte.

This is pretty good, bur taking ito account larger N (around 8000) we get $225 / 150 \approx 1.5$*, So we confirm that we are memory bound

Below is Apple acceleate, which average about 1400Gflops for large matrices,

Meaning it is getting around ~9 FLOPs/Byte (very impressive)

Memory Alignment

```
float A[N][N] __attribute__((aligned(32)));;
float B[N][N] __attribute__((aligned(32)));;
float BT[N][N] __attribute__((aligned(32)));;
float C[N][N] __attribute__((aligned(32)));;
```

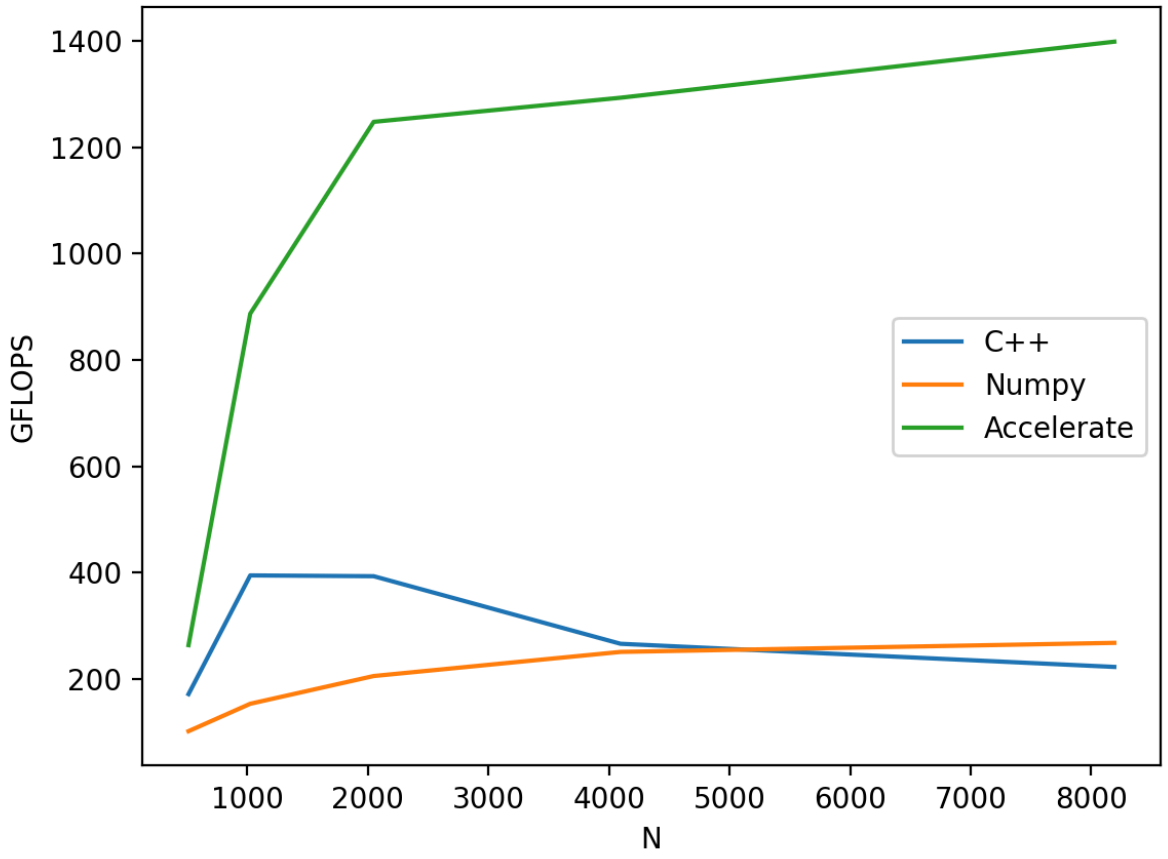
```
float Cvals[N][N] __attribute__((aligned(32)));;
```

This did not give us any improvement, even though some sources online claim it to be benefitial

Apple Accelerate framework

Apple Accelerate framework is extremly fast, averaging ~1390 Gflops (1.4 Tflops)

```
vDSP_mmul(A,
          1,
          B,
          1,
          C,
          1,
          N, N, N);
```



Sources

〔1〕 The Art of HPC page 15/16
〔2〕 The Art of HPC page 268-269

Blocking

- https://gist.github.com/metallurgix/8ee5262ed818730b5cc0
  George Hotz
- https://www.youtube.com/watch?v=VgSQ1GOC86s&t=4313s&pp=ygUcZ2VvcmdlIGhvdHogbm9vYiBsZXNzb24gZ2VtbQ%3D%3D
  Meduem Article
- https://vaibhaw-vipul.medium.com/matrix-multiplication-optimizing-the-code-from-6-hours-to-1-sec-70889d33dcfa
  Neon Arm Instructions
- https://developer.arm.com/architectures/instruction-sets/intrinsics/#f:@navigationhierarchiessimdisa=[Neon]&f:@navigationhierarchiesreturnbasetype=[float]&f:@navigationhierarchieselementbitsize=[32]&q=vld1
  Apple Accelerate
- https://developer.apple.com/documentation/accelerate/vdsp-library
  AMX
- https://zhen8838.github.io/2024/04/23/mac-amx_en/
  Loop unrolling
- https://en.wikipedia.org/wiki/Loop_unrolling#:~:text=Loop%20unrolling%2C%20also%20known%20as,or%20by%20an%20optimizing%20compiler