CS267 HW1 Report

Yichen Zhou, Ryan Yang, Mostafa Rohaninejad

I. Group Member Work Divisions:
Yichen Zhou: Block size, AVX Intrinsics, Report
Ryan Yang: Block size, AVX Intrinsics, Report
Mostafa Rohaninejad: Block size, AVX Intrinsics, Prefetching, GPU machine

II. Optimizations Used or Attempted:
(1). Block Size: We tried out different block sizes from 4 to 512 and ended up using 32 as the block size. We found that if the block size is equal or slightly bigger than the input, we could achieve good optimization. However, for example, if we have an input size of 129 by 129 and choose the block size of 128. We would end up creating four 128 by 128 blocks which wasted a lot of space and the time to allocate the memory. We could not set the block size to be too big, since it would perform poorly on the matrix with small input sizes. Also, we could not set the block size too small. Otherwise, it would be just like doing the naive matrix multiplication. We decided to make the block size interactive over different sizes of the matrix. By default we set the block size to 16, but as the size of the matrix increased, we got better performance by scaling up the block size in powers of 2.

Tiled matrix multiplication:
(2). Padding: We tried to pad the original A, B and C matrices to have the size of multiples of 4 so that when we are using the block_simd which is doing 4 x k to k x 4 multiplication, we do not need to consider the residual part that we got from divided by 4.

(3) AVX Intrinsics: We split each block matrix into smaller 4 size block matrices and used AVX2 intrinsics to multiply these efficiently. Since we padded the matrices to have sizes that are multiples of 4, we did not need to deal with edge cases where the original block matrix does not have a size that is a multiple of 4. Originally, we also tried to split the block matrix into 8 size block matrices for better SIMD optimization, but we realized that even though the Xeon processor on the Cori machine supports AVX512, the CMake file does not include the flag to enable AVX512. Therefore, we had to use AVX2 and just do 4x4 block matrix multiplications.

(4) Loop Unrolling: For the SIMD function and the main square_dgemm function, we unrolled the outer loops, giving us marginally better performance.

(5) Other: One thing we tried was to use _mm_malloc instead of the regular malloc to create copy matrices. This resulted in marginally better performance but had memory corruption issues so we stuck with the regular malloc. In addition, we also added

__restrict__ keywords to multiple locations with pointers, giving us slightly better performance.

III. Results of Optimization:

The following is our performance on the cori machine using the padding method:

```
Description:    Simple blocked dgemm.

Size: 31        Mflop/s:  6650.32       Percentage: 18.07
Size: 32        Mflop/s:    12220       Percentage: 33.21
Size: 96        Mflop/s:  11704.6       Percentage: 31.81
Size: 97        Mflop/s:  8377.79       Percentage: 22.77
Size: 127       Mflop/s:  8147.07       Percentage: 22.14
Size: 128       Mflop/s:  10375.1       Percentage: 28.19
Size: 129       Mflop/s:  8647.02       Percentage: 23.50
Size: 191       Mflop/s:  8190.86       Percentage: 22.26
Size: 192       Mflop/s:  9950.71       Percentage: 27.04
Size: 229       Mflop/s:  7806.62       Percentage: 21.21
Size: 255       Mflop/s:  6778.85       Percentage: 18.42
Size: 256       Mflop/s:  7470.08       Percentage: 20.30
Size: 257       Mflop/s:  7441.64       Percentage: 20.22
Size: 319       Mflop/s:  9010.29       Percentage: 24.48
Size: 320       Mflop/s:  10824.6       Percentage: 29.41
Size: 321       Mflop/s:  9002.46       Percentage: 24.46
Size: 417       Mflop/s:  9227.16       Percentage: 25.07
Size: 479       Mflop/s:  9202.27       Percentage: 25.01
Size: 480       Mflop/s:  10977.4       Percentage: 29.83
Size: 511       Mflop/s:  4429.62       Percentage: 12.04
Size: 512       Mflop/s:  4476.24       Percentage: 12.16
Size: 639       Mflop/s:  8005.14       Percentage: 21.75
Size: 640       Mflop/s:  8814.27       Percentage: 23.95
Size: 767       Mflop/s:  7211.09       Percentage: 19.60
Size: 768       Mflop/s:     7623       Percentage: 20.71
Size: 769       Mflop/s:  6711.92       Percentage: 18.24
Average percentage of Peak = 22.9177
~
```

IV. Odd behavior in Performance:

There are performance issues when the block size is small, close to a multiple of 128, or exactly on a multiple of 128. The reason for this dip in performance is due to the fact that we would need to add a significant amount of padding in order to make the matrix evenly divisible for use by the AVX intrinsics. This would create overhead, but this overhead is worth it since without AVX intrinsics, these cases would be even slower.

V. Performance on a different machine:

We also ran the code on a GPU machine. The average performance reached is 39.8%. It is much higher than what we got on the cori.