Team:
Eliot Samuel Flores Portillo esf2150
Ahuva Malka Bechhofer amb2572
Emily Soto es4060
Aiko Gerra alg2252

Matrix Multiplication

Initial Optimization Plan and Type of Expected Speedup

We initially chose the tiled iterative algorithm, similar to what is described here. The overall concept is to harness spatial and temporal locality by computing the portions of all dot products that the values within a given tile, or segment of the matrix can be used for. This ensures that memory and cache access patterns leverage both spatial and temporal locality, as close-together data is accessed in quick succession.

```C/C++
float dotProd = 0;
  for (int I = 0; I < SIZE; I += tileSize)
  {
    for (int J = 0; J < SIZE; J += tileSize)
    {
      for (int K = 0; K < SIZE; K += tileSize)
      {
        for (int i = I; i < I + tileSize; i++)
        {
          for (int j = J; j < J + tileSize; j++)
          {
            for (int k = K; k < K + tileSize; k++)
            {
              dotProd += matrixA[i][k] * matrixB[k][j];
            }
            resultMatrix[i][j] += dotProd;
            dotProd = 0;
          }
        }
      }
    }
  }
```

More specifically, matrix multiplication of A x B generally involves repeatedly computing the dot-product of a given row of A with each column of B. For a 2048 x 2048 matrix and our cache line size of 64 B, this means accessing many different cache lines just to calculate one value in

the result matrix. A row would be split into about 32 cache lines (C uses row-major order), and a column would be distributed across cache lines, involving 2048 lines per column. With only 512 cache lines in the L1 cache, even one column cannot fit at once, and doing so would mean the cache lines for the row would not fit at all. Therefore, we expected tiling to yield spatial locality benefits. With a 32x32 tile, for example, an entire tile row can easily fit in one cache line, and the tile column can fit in just 32 cache lines. Plus, the same column cache lines are accessed over consecutive iterations in the tile, each multiplied with a given tile row, taking advantage of temporal locality, too. This is an improvement over the typical calculation, where the cache lines containing the first values of a given column probably need to be pulled into L1 cache again by the time the adjacent column is needed, as all the cache lines for a 2048 column can't fit in L1.

Initial Observations

Our observations showed that the tiling worked. We initially saw a slowdown, but testing proved that calling a min() function as part of our loops caused the slowdown - the overhead associated with control moving to min() and the conditional branching inside of it overwhelmed the benefits of tiling.

```
for (int k = K; k < min(K + tileSize, SIZE); k++)
```

We then removed min() and tested tile sizes that fit perfectly inside 2048, which is when we saw a 50.98% speedup with tile size 32 x 32.

	Clock Ticks		
	Baseline	tileSize = 32	tileSize = 256
	131426842506	62041383990	111069281566
	127672218576	61681613270	110459600074
	126641934238	61578976776	111724627906
	121513641476	61594456292	112797710078
	121176627278	61566268326	109940278980
Average Ticks	125686252815	61605328666	111230554260

Percent Speedup (vs. baseline)	N/A	50.98	11.50

Further Optimizations

We then added an additional optimization. We transposed matrix B when we took it in as input. We weren't sure if this would work, but we thought that we might improve the system's prefetching and cut cache access time by reading from fewer cache lines/memory blocks (1 cache line for a tile row instead of 32 for a tile column) at once, and switching between tile columns by changing row index rather than column index. We edited other lines to adjust for this optimization and verified that it produced output in the same format as before.

In these next trials, we achieved a 53.10% speedup with transposition and a tile size of 32, which was better than the 50.98% above and better than the 49.72% from 32 without transposition.

Noticeably, 32 is smaller than the tile size proposed in the standardized algorithm (the square root of cache size, or about 108 for us), and larger sizes such as 256 seemed to cause slowdowns above as well. That algorithm is for an ideal fully associative cache, while our L1 cache has an associativity of 8, and so it makes sense that the ideal size differs.

	tileSize = 64	tileSize = 64, transpose matrix B	tileSize = 32	tileSize = 32, transpose matrix B	tileSize = 16, transpose B
	65268128316	59211676084	63099258886	59041133430	59919183670
	64424769412	59381479930	63422661054	59071024058	59862883740
	64063646556	59149962758	63356964922	59148909824	59951254776
	63814397046	59203222452	63567610504	59026500354	60306592468
	63677949642	59016260268	62967052962	58974795048	59851765464
	63907797966	58786630468	62982107370	58620457098	59851510504
	63922829288	58461309520	62967909234	58749727246	59866516788
Average ticks	64154216889	59030077354	63194794990	58947506723	59944243916

Percent Speedup (vs. baseline)	48.96	53.03	49.72	53.10	52.31

We then tried:

1) Perform minor loop unrolling. We expected this to slightly decrease execution time since we only did it for about 7 lines.

2) Vectorize (with intrinsics) an unrolled portion of the loop. We expected this to have a more significant effect than purely unrolling, as this would decrease time spent on executing the dot product calculations rather than just removing control overhead.

```
C/C++
        for (int k = K; k < K + tileSize; k += 16)
            {
              const __m256 vectorA = _mm256_loadu_ps(matrixA[i] + k);
              const __m256 vectorB = _mm256_loadu_ps(matrixB[j] + k);
              const __m256 vectorC = _mm256_loadu_ps(matrixA[i] + k + 8);
              const __m256 vectorD = _mm256_loadu_ps(matrixB[j] + k + 8);
              const __m256 product1 = _mm256_mul_ps(vectorA, vectorB);
              const __m256 product2 = _mm256_mul_ps(vectorC, vectorD);

              const __m256 sum = _mm256_add_ps(product1, product2);
              const __m128 sumA = _mm256_extractf128_ps(sum, 0);
              const __m128 sumB = _mm256_extractf128_ps(sum, 1);
              const __m128 sumReduced = _mm_add_ps(sumA, sumB);
              const __m128 sumCombined = _mm_hadd_ps(sumReduced, sumReduced);
              const __m128 allResultVec = _mm_hadd_ps(sumCombined, sumCombined);

              resultMatrix[i][j] += _mm_cvtss_f32(allResultVec);
            }
```

We managed to achieve a 71.8% speedup.

	Additional trials without further optimization	Clock Ticks (partially unroll the k loop)	Vectorize an unrolled portion of the k loop (with gcc -mavx2 and intrinsics)
	59041133430	55920223912	35332647560

	59071024058	55884498042	35326397412
	59148909824	55853992278	35229839324
	59026500354	55699805412	35517153124
	58974795048	55 778 395 306	35748285942
Average ticks	59052472543	55839629911	35430864672
Percent Speedup (vs. baseline)	53.02	55.57	71.81

The above were done with a transposed matrix B and tileSize 32.

Final Result

We found that a tile size of 32 x 32 with the iterative tile algorithm, combined with the small effect of matrix B transposition and the larger 18% boost from vectorized loop unrolling, can achieve a speedup of 71.81%.

Softmax

Optimization Plans and Type of Expected Speedup

To accelerate the softmax function, we first set out to vectorize the code using the single input multiple data (SIMD) instruction set. We hoped to see a significant speedup from the ability to complete four computations in parallel rather than one. However, when implementing this optimization, we found that the **'_mm_exp_ps'** function does not work with the C compiler (gcc). The '_mm_exp_ps' function is part of the short vector math library (SVML), and can only be used using an Intel-based compiler. We tried to resolve this at first by using threading, but the overhead caused by context switching and thread management caused a major slowdown of 219.56%.

We solved this issue by not using the SIMD exponential function and reverted to the **'expf()'** function used in the baseline softmax code.

Observations

Below, you can see two SIMD softmax implementations. The first takes advantage of SIMD's parallelization functions to try and parallelize the exponential computation as much as possible.

```
for (int i = 0; i < SIZE; i+= 4) {
      const __m128 xVals = _mm_loadu_ps(vector + i);
      const __m128 subtracted = _mm_sub_ps(xVals, maxVector);
      _mm_maskstore_ps(buffer, storeMask, subtracted);
      const __m128 eXVec = _mm_set_ps(expf(buffer[3]),
      expf(buffer[2]), expf(buffer[1]), expf(buffer[0]);
      sumVec = _mm_add_ps(sumVec, eXVec);
      _mm_maskstore_ps(result + i, storeMask, eXVec);
}
```

The second SIMD implementation just uses the 'expf()' alone without the intrinsic **'_mm_set_ps()'** function.

```
for (int i = 0; i < SIZE; i+= 1) {
      result[i] = expf(vector[i] - max);
      sum += result[i];
}
```

However, this causes a slowdown due to the loss of parallelization. We tried it because we suspected that doing fewer set or store operations may speed up the code, but the benefit of the parallelization seems to have outweighed it.

	Clock Ticks			
	Baseline softmax	Threading softmax	SIMD softmax (same algorithm, but as many operations as possible parallelized)	SIMD softmax (parallelize less code around expf() call)
	1664080 1651154 1679530 1666966 1675278 1694672 1661012 1721904 1694092 1693852 1692480 2888334 1843732 1753650 1727896	6298182 5712806 5616380 5584120 5573916 5551010 5501122 5528778 5824826 5754508 5689710 5707012 5752590 5601930 5654292	1141864 1101342 1096146 1095830 1106112 1157192 1099346 1055104 1129748 1112270 1106828 1098142 1136926 1115994 1087590	1252128 2143724 1770788 1413366 1123798 1107954 1152196 1165998 1077168 1064520 1038016 1286470 1211784 1091824 1057628

Average Ticks	1780575.467	5690078.8	1109362.267	1263824.133
Percent Speedup (vs. baseline)	N/A	-219.56%	37.70%	29.02%

Final Result

Calculating expf() with scalar functions resulted in a 29.02% speedup but parallelizing operations such as finding the maximum, calculating exponents and summing up vector elements created a speedup of 37.70%.

Embedding lookup

Initial Optimization Plan and Type of Expected Speedup

Initially we chose to implement an algorithm that sorted the indices while also keeping track of the order of the indices. The thought was that if the indices are sorted, when fetching the data we will have more cache hits since we will be accessing data that is contiguous or close to each other in memory with enough temporal locality for it to still be in the cache. However, once we implemented the code we did not see any significant speedup. This is because the indices array is only 20 randomly initialized values and in such a huge space, seeing a speedup is unlikely. We therefore pivoted to think about other ways we can optimize the data access. This is where we found the usage of memcpy extremely useful.

This is our **unoptimized** code:

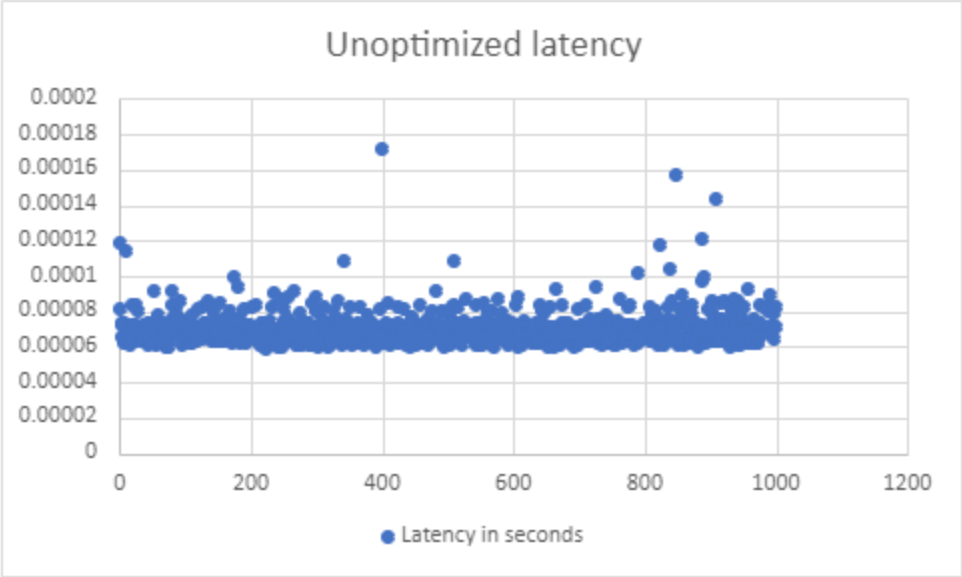
```
void  embedding_lookup(double *embedding_matrix, IndexedValue *indices, double **embedding){

        //quickSort(indices, 0 , INDICES_SIZE - 1); We did not use this in the end

        for(int i = 0; i < INDICES_SIZE; i++){
                int idx = indices[i].value * COL;
                //memcpy(embedding[i], &embedding_matrix[idx], COL * sizeof(double));

                for (int j = 0; j < COL; j++) {
                        embedding[i][j] = embedding_matrix[idx + j];
                }
        }

}
```

This is our **optimized** code:

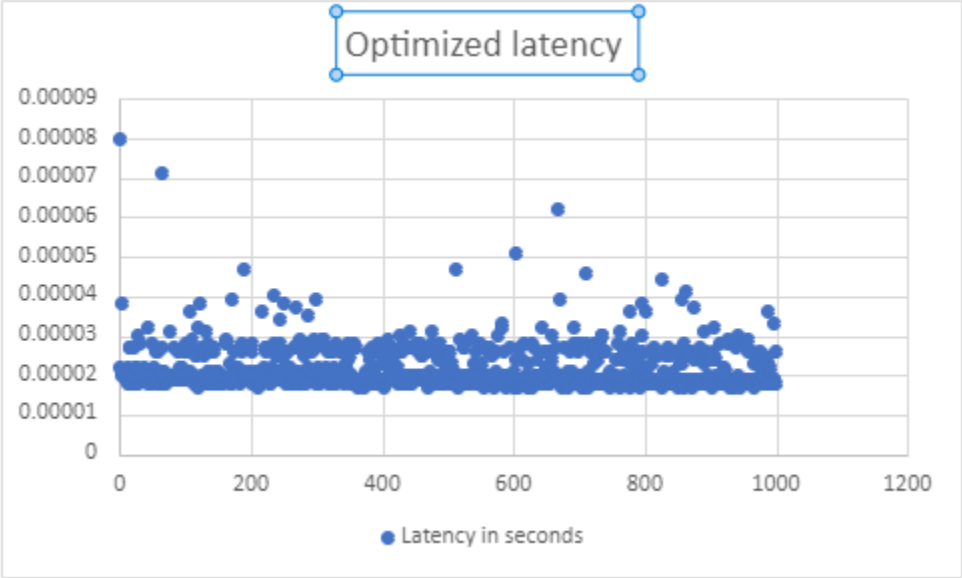
```
void  embedding_lookup(double *embedding_matrix, IndexedValue *indices, double **embedding){

        //quickSort(indices, 0 , INDICES_SIZE - 1); We did not use this in the end

        for(int i = 0; i < INDICES_SIZE; i++){
                int idx = indices[i].value * COL;
                memcpy(embedding[i], &embedding_matrix[idx], COL * sizeof(double));

                /*for (int j = 0; j < COL; j++) {
                        embedding[i][j] = embedding_matrix[idx + j];
                }*/

        }

}
```

We ran 1000 trials in which we called the embedding lookup with different indices arrays on the **same embedding space** and calculated how long it took for the embedding lookup function to complete. The trials for the optimized vs unoptimized were **identical (i.e. on the same embedding space)** besides the embedding lookup optimization using memcpy.

```
        for(int i = 0; i < INDICES_SIZE; i++){
                embedding[i] = (double *)malloc(COL * sizeof(double));
        }

        for(int call = 0; call < 1000; call++){
                for(int i = 0; i < INDICES_SIZE; i++){
                        indices[i].value = rand() % ROW; //this was not used for optimization in the end
                        indices[i].index = i; //same as above comment

        }



        clock_gettime(CLOCK_MONOTONIC, &start);


        embedding_lookup(embedding_matrix, indices, embedding);


        clock_gettime(CLOCK_MONOTONIC, &end);

        time_spent = ((end.tv_sec * 1000000000.0 + end.tv_nsec) - (start.tv_sec * 1000000000.0 + start.tv_nsec))
/ 1000000000.0;

        acc += time_spent;

        printf("Elapsed time in seconds: %f \n", time_spent);

        }
```



MEAN = 7.0263E-05 , MEDIAN = 0.00007, MODE = 0.00007
Each dot is one trial



MEAN = 2.1567E-05 , MEDIAN = 0.000019, MODE = 0.000019
Each dot is one trial

For an embedding lookup to be optimized with respect to memory access, the manner in which the data is fetched from the large embedding needs to be optimized. By using memcpy instead of manually looping through the embedding matrix and copying the data into the embedding, we achieve around a 70% speedup as we can see from the above mean, median and mode.
In order to determine what causes the speedup we decompiled the code to see the operations called during runtime by memcpy. Just using objdump even when running the program as static did not provide the exact instructions implemented, rather it referenced a location in the plt

(which just referred to the GOT) and all the different memcpy type functions from the library. Since we were unable to view the GOT, we were unable to find the exact instructions memcpy uses.

In order to have a more detailed breakdown we installed **gdb**, there we set a **breakpoint at memcpy** so we can view exactly what instructions are being called during runtime. This revealed the following:

Breakpoint 1, memmove () at ../sysdeps/x86_64/multiarch/memmove-vec-unaligned-erms.S:143
143 ../sysdeps/x86_64/multiarch/memmove-vec-unaligned-erms.S: No such file or directory.

We see that memcpy uses **memmove** which is part of the standard C library (libc), specifically an optimized version for x86_64 architectures (which is the architecture of our machine).
We then disassembled the code to show the implementation of memmove being executed.

disassemble:
(gdb) disassemble
Dump of assembler code for function memmove:
=> 0x00007ffff7ff1ff0 <+0>: mov %rdi,%rax
0x00007ffff7ff1ff3 <+3>: cmp $0x10,%rdx
0x00007ffff7ff1ff7 <+7>: jb 0x7ffff7ff2010 <memmove+32>
0x00007ffff7ff1ff9 <+9>: cmp $0x20,%rdx
0x00007ffff7ff1ffd <+13>: ja 0x7ffff7ff2056 <memmove+102>
0x00007ffff7ff1fff <+15>: movups (%rsi),%xmm0
0x00007ffff7ff2002 <+18>: movups -0x10(%rsi,%rdx,1),%xmm1
0x00007ffff7ff2007 <+23>: movups %xmm0,(%rdi)
0x00007ffff7ff200a <+26>: movups %xmm1,-0x10(%rdi,%rdx,1)
0x00007ffff7ff200f <+31>: ret
0x00007ffff7ff2010 <+32>: cmp $0x8,%dl
0x00007ffff7ff2013 <+35>: jae 0x7ffff7ff2027 <memmove+55>
0x00007ffff7ff2015 <+37>: cmp $0x4,%dl
0x00007ffff7ff2018 <+40>: jae 0x7ffff7ff2038 <memmove+72>
0x00007ffff7ff201a <+42>: cmp $0x1,%dl
0x00007ffff7ff201d <+45>: ja 0x7ffff7ff2045 <memmove+85>
0x00007ffff7ff201f <+47>: jb 0x7ffff7ff2026 <memmove+54>
0x00007ffff7ff2021 <+49>: movzbl (%rsi),%ecx
0x00007ffff7ff2024 <+52>: mov %cl,(%rdi)
0x00007ffff7ff2026 <+54>: ret
0x00007ffff7ff2027 <+55>: mov -0x8(%rsi,%rdx,1),%rcx
0x00007ffff7ff202c <+60>: mov (%rsi),%rsi
0x00007ffff7ff202f <+63>: mov %rcx,-0x8(%rdi,%rdx,1)
0x00007ffff7ff2034 <+68>: mov %rsi,(%rdi)
0x00007ffff7ff2037 <+71>: ret

```
0x00007ffff7ff2038 <+72>:    mov    -0x4(%rsi,%rdx,1),%ecx
0x00007ffff7ff203c <+76>:    mov    (%rsi),%esi
0x00007ffff7ff203e <+78>:    mov    %ecx,-0x4(%rdi,%rdx,1)
0x00007ffff7ff2042 <+82>:    mov    %esi,(%rdi)
0x00007ffff7ff2044 <+84>:    ret
0x00007ffff7ff2045 <+85>:    movzwl -0x2(%rsi,%rdx,1),%ecx
0x00007ffff7ff204a <+90>:    movzwl (%rsi),%esi
0x00007ffff7ff204d <+93>:    mov    %cx,-0x2(%rdi,%rdx,1)
0x00007ffff7ff2052 <+98>:    mov    %si,(%rdi)
0x00007ffff7ff2055 <+101>:   ret
0x00007ffff7ff2056 <+102>:   cmp    $0x80,%rdx
0x00007ffff7ff205d <+109>:   ja     0x7ffff7ff20cf <memmove+223>
0x00007ffff7ff205f <+111>:   cmp    $0x40,%rdx
0x00007ffff7ff2063 <+115>:   jb     0x7ffff7ff20ac <memmove+188>
0x00007ffff7ff2065 <+117>:   movups (%rsi),%xmm0
```

This reveals why we achieve such significant speedups with memcpy. Since this implementation for data sizes between 16 (0x10) and 32 (0x20) bytes movups is used with a SSE register (Streaming SIMD Extensions) seen by xmm0 and xmm1, it shows an optimization for moving blocks of data more efficiently. More specifically, SSE is a type of SIMD instruction set extension for x86 processors which allows for parallel processing of floating point data. Registers xmm0 and xmm1 are 128 bit wide which mean they can hold and operate on multiple pieces of data in parallel with a single register. The movups, which stands for “move unaligned packed single-precision floating-point values,” is used to move 128 bits of data, which could be four 32 bit single precision floating point values or two 64 bit floating point numbers (as it is for us since we are using doubles). The unaligned porting means it can work with addresses that are not aligned on 16-byte boundaries, thus making it more flexible in terms of memory access patterns. So, by using movups with the SSE registers for memory sizes between 16 and 32 bytes, it takes advantage of parallel processing capabilities. Moving up to 16 bytes at a time in a single instruction reduces the total number of instructions executed and can significantly speed up the memory transfer.

Final Result

The usage of the memcpy() function inside the embedding lookup function provides a speedup of around 70% as the result of optimized memory accesses.