HW 3

Problem 1

Part 1

x Value	Num Terms	Float Sin(x)	Double Sin(x)	Time (Float) (s)	Time (Double) (s)
1	200	0.8414709568	0.8414709848	0.1531352997	0.1540260315
2	200	0.9092974663	0.9092974268	0.1537399292	0.1546249390
4	200	-0.7568024397	-0.7568024953	0.1521701813	0.1529121399
8	200	NaN	NaN	0.1533546448	0.1586914062

Because of the intrinsic constraints of floating-point arithmetic, the float and double outputs differ slightly for lower x values (1, 2, and 4). Doubles provide around 15–16 significant digits, whereas single-precision floats provide about 7. This results in small differences as rounding and accumulation mistakes happen throughout the alternating series summation. Since the Taylor series for sin(x) converges near zero, the intermediate terms stay relatively accurate for smaller values.

The reason that when the x value is 8 both the floating point and double can't hold the intermediate values. They get too large and overflow, this then causes undefined behavior for the next term which tries to cancel it out and that leads to NaN values.

Part 2

Float

2.1 = 0x40066666 = 01000000000001100110011001100110

6300 = 0x45C4E000 = 01000101110001001110000000000000

-1.044 = 0xBF85A1CB = 10111111100001011010000111001011

Double

2.1 = 0x4000CCCCCCCCCCCD = 0100000000000000110011001100110011001100110011001100110011001101

6300 = 0x40B89C0000000000 = 0100000010111000100111000000000000000000000000000000000000000000

-1.044 = 0xBFF0B4395810624E = 1011111111110000101101000011100101011000000100000110001001001110

Problem 2

Part 1

The performance results demonstrate a notable increase in speed. With a speedup factor of nearly 2.74×, the AVX512 version finished in about 0.005892 ms, whereas the non-vectorized matrix–vector multiplication took about 0.016136 ms. The efficiency of fused multiply-add operations, which lower the number of instructions and the loop overhead, and the higher data throughput provided by processing 16 items at once are the main causes of this improvement.

Part 2

vmovaps %zmm0, -112(%rbp)

At the address provided by –112(%rbp), this instruction transfers 512 bits (16 packed single-precision floats) from the ZMM register (zmm0) into memory. The term "aligned packed single-precision," or "aps," in vmovaps indicates that the destination memory is presumed to be 64-byte aligned. Usually, this instruction is used to effectively store intermediate or final vector results.

vmulps -432(%rbp), %zmm0, %zmm0

In this case, packed single-precision floating-point data are multiplied element-wise by vmulps. It sends the result back to zmm0 after multiplying the 16 floats that were loaded from memory at –432(%rbp) by the 16 floats that were already in zmm0. When compared to scalar multiplication, this method greatly improves performance by enabling the software to process 16 multiplication operations simultaneously.

vaddps -112(%rbp), %zmm0, %zmm0

Packed single-precision floating-point values are added via the vaddps instruction. Here, 16 floats are loaded from memory at –112(%rbp), added elementwise to the 16 floats in zmm0, and the result is stored in zmm0. This is usually used to vectorize the accumulation of partial sums (e.g., adding the result of a multiplication to an existing total).

Problem 3

See code

Problem 4

Part 1

My implementation of matrix multiplication on a 256x256 with ten iterations of the loop takes on average 9.6ms and the OpenBLAS version takes on average 5.3ms. This is a significant difference especially with larger and larger problem sizes. There are a few optimizations that I could have made that might have helped get closer to OpenBLAS. If I had used pthreads rather than OpenMP there would be less overhead and that would help. There are also different matrix multiplication algorithms that I could have utilized that would net a speedup. Additionally my implementation was operating on doubles rather than floats, if it were operating on floats I would be able to use a bigger stride for my vector operations which would lead to significant speedup. Another reason my implementation is slower than OpenBLAS is because a team of dedicated researchers spend their careers working on it.

Part 2

Intel Xeon Platinum 8276

- Total Cores: 56 (28 cores per socket, 2 sockets)
- Threads per Core: 1
- Base Clock Speed: 2.20 GHz
- Max Clock Speed: 4.00 GHz
- L1 Cache: 1.8 MiB (Data) + 1.8 MiB (Instruction)
- L2 Cache: 56 MiB
- L3 Cache: 77 MiB
- NUMA Nodes: 2 (even cores on Node 0, odd cores on Node 1)
- AVX-512 Support: Yes
- Dense Matrix Multiplication Time: 5.33 ms

Intel Xeon Gold 6132

- Total Cores: 28 (14 cores per socket, 2 sockets)
- Threads per Core: 1
- Base Clock Speed: 2.60 GHz
- Max Clock Speed: 3.70 GHz
- L1 Cache: 896 KiB (Data) + 896 KiB (Instruction)
- L2 Cache: 28 MiB
- L3 Cache: 38.5 MiB
- NUMA Nodes: 2 (even cores on Node 0, odd cores on Node 1)
- AVX-512 Support: Yes
- Dense Matrix Multiplication Time: 4.87 ms

Clock Speed

The base clock of the Platinum 8276 (2.2 GHz base, 4.0 GHz max) is marginally lower than that of the Xeon Gold 6132 (2.6 GHz base, 3.7 GHz max).
The Gold 6132 might be boosting to a higher average frequency if OpenBLAS is operating in a power-saving mode.

Thread and Core Utilization

Although the Platinum 8276 has 56 cores instead of 28, not all of them may be utilized effectively since OpenBLAS can have memory-bandwidth limitations.
OpenBLAS might not be making use of all of the Platinum 8276's cores if it is dynamically choosing thread counts.

L3 Cache Differences

Despite having twice as many cores, the Platinum 8276's per-core cache availability is comparable despite having a bigger L3 cache (77 MB vs. 38.5 MB).
The CPU's ability to keep frequently used data near to execution units is influenced by cache efficiency.

Memory Bandwidth and NUMA Effects

Since both CPUs have two sockets, NUMA effects—also known as non-uniform memory access—may be involved.
There might be additional latency if memory allocation isn't optimal (for example, if threads are accessing remote memory across sockets).

Problem 5

Source: https://arxiv.org/abs/2307.06305

This paper discusses the Diagonally-Addressed (DA) storage format for sparse matrices. The main principle is that it stores the indices in relation to the diagonal. This allows this format to use smaller integer types than both CSR and ELLPACK formats. By using smaller types they were able to achieve better memory efficiency which also led to better cache utilization. According to the paper it was 11% faster in a single threaded environment and 17% faster in a multithreaded environment. This method works especially well for matrices that already have a low bandwidth and could be very applicable for applications like finite element analysis.