**Project 1: SIMD Advantage Profiling**

Setup & Methodology

This project was written as a set of experimental C files utilizing a custom library “kernel.h” to implement all relevant kernel functions. The kernel functions chosen for this benchmarking program are: Streaming AX + Y (SAXPY), Dot Product Reduction, and Element-wise Multiply. Each of these functions are implemented as a callable function that takes in the data arrays to be operated on as well as their size and completes the entire kernel run within the function, allowing for easy and highly runtime measurement utilizing the C time library.

*Software Specifications:*

The provided c files and the associated kernel library are intended to be built using gcc. They can be built by running the following command:

gcc -mavx2 -mfma <filename>.c kernel.c -o <exename>

<filename>: name of the experiment file

<exename>: name to be given to the resulting compiled executable

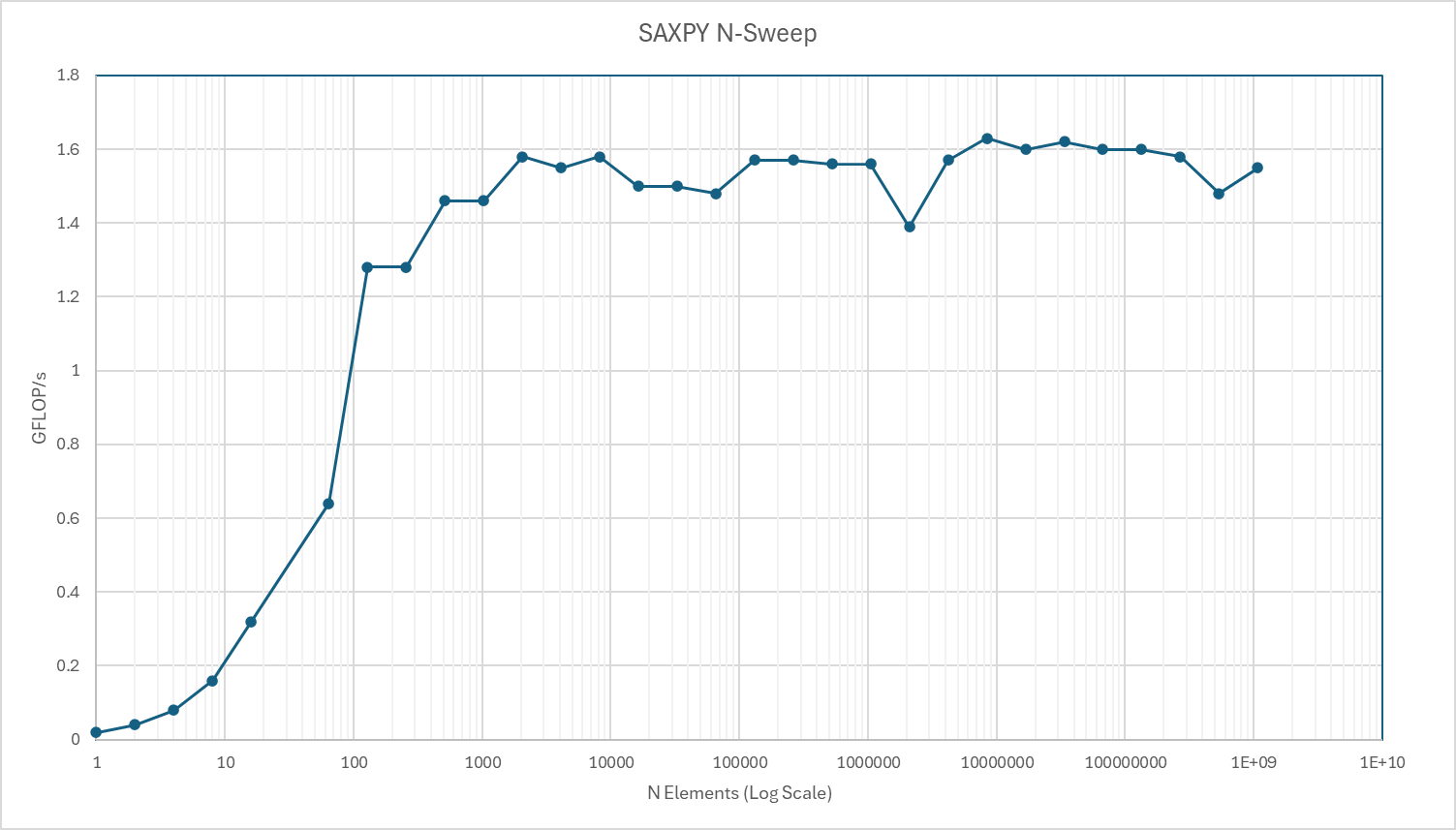
Note the use of the arguments ‘-mavx2,’ ‘-mfma.’ These arguments tell gcc that Intel AVX2 vector instructions are to be included, and the build will fail without them due to their presence in kernel.c. Currently there is no way to build an experiment without these arguments, so both scalar instructions as well as their vector counterparts will be present in the resulting binary, however vector functions are only used when explicitly called by the programmer. Additional optimization flags such as ‘-03’ may be used when compiling, however others such as ‘-fno-tree-vectorize’ may interfere with AVX implementation.

*Target System Specifications:*

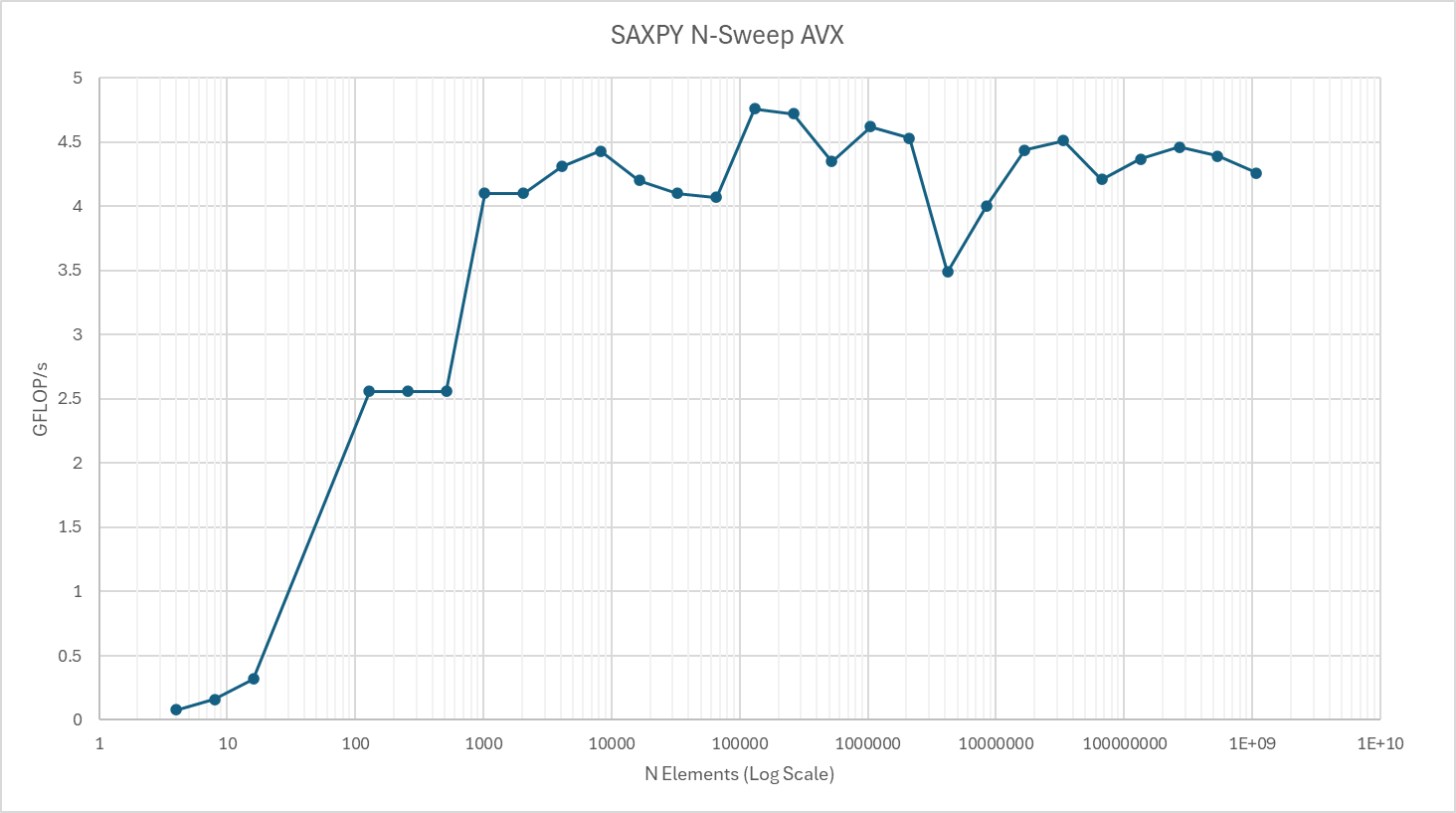
This project was built and run on a Windows 11 operating system. The system processor under analysis is an AMD Ryzen 7 7800X3D, utilizing Zen 4 architecture with a 3D L3 cache structure. The processor features a 4.2GHz base clock frequency with 8 cores, two threads per core. The available cache is 64kB L1 per core, 1MB L2 per core, and 96MB L3 shared. Due to the program’s use of Intel AVX instructions, it is important that the programmer first check that these instructions are supported on their intended target processor.

L1 to L3 Sweep

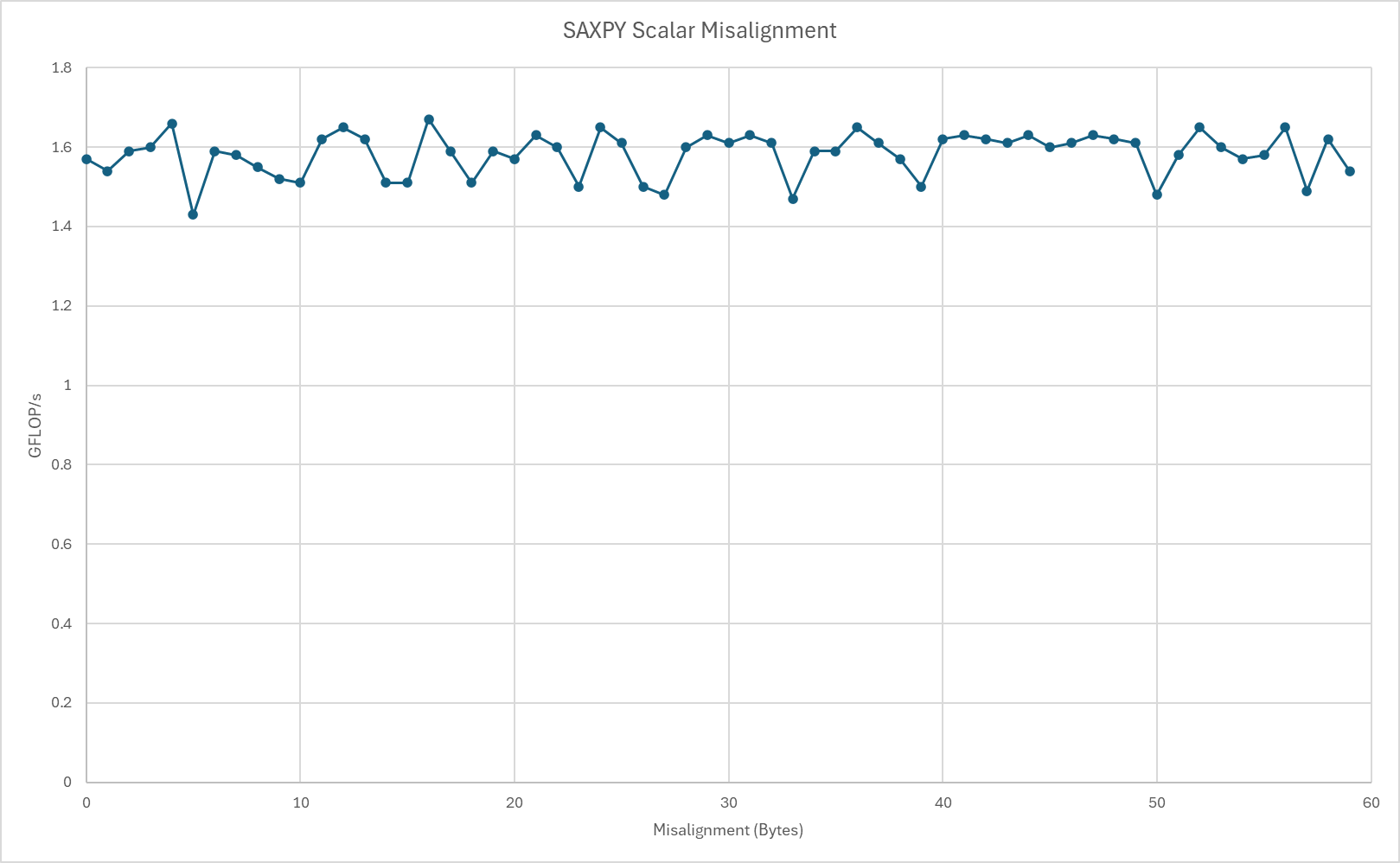
While there is no practical way to force a program to only utilize a certain cache level, we can ensure that most operations will fall within a certain level by fitting our dataset into the size of the desired cache. For example, if we wish to implement the SAXPY kernel on L1 for our system, we need two vectors, x and y, to fit in L1. Our given L1 size is 64kB per core, so if we use float32 data points (4B), then we have a maximum vector size of (64\*1000)/4/2 = 8,000 elements. In reality we should use even smaller vectors to ensure that all data stays within L1. This same methodology applies for L2 (between 64kB and 1MB), L3 (between 1MB and 96MB) and beyond into DRAM. We can create an experiment to sweep the number of elements in our vectors across a wide range to analyze how performance changes as our program crosses cache boundaries.

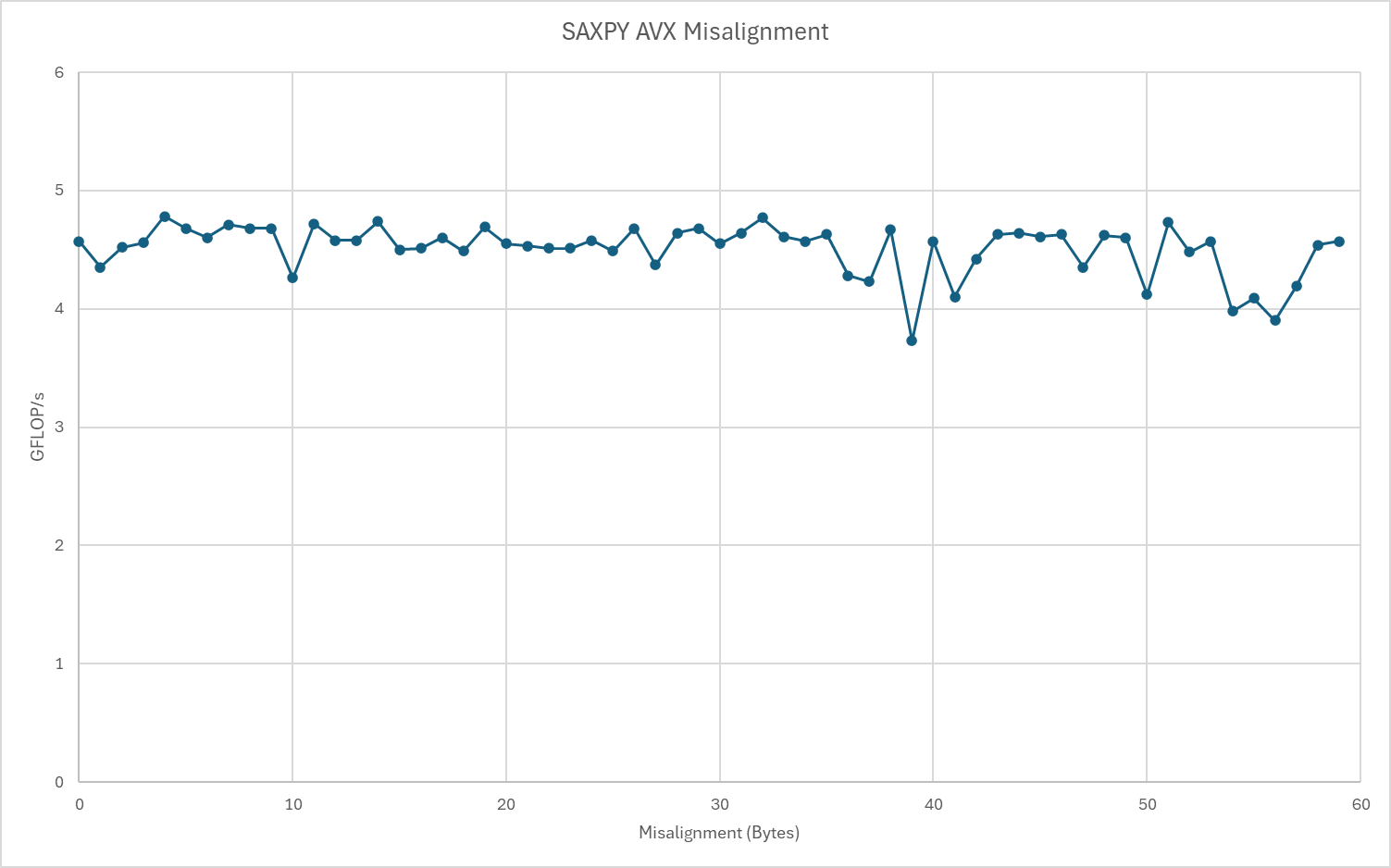
Fig. X

Performing the same data set sweep but this time implementing AVX instructions to stream multiple data points at once, we immediately see a clear speedup.

Fig. X

Data Misalignment Effects

Fig. X

Fig. X

In both the scalar and vectorized case, no appreciable effect occurred from increased data misalignment save for maybe a slight downward trend in average GFLOPS and increased irregularity in the vectorized case. This could be due simply to modern hardware systems having highly robust built-in alignment handling, or perhaps an improper implementation of misalignment. The possible increased impact on the vectorized performance would likely be due to an increased sensitivity of AVX instructions to misaligned data, requiring more overhead for error handling.