# EECE 5640 Homework 3

Emily Costa (costa.em@northeastern.edu)

September 27, 2021

**Question 1.**

The set of 3 single-threaded diverse benchmark programs that I selected focus on integer operations, floating-point operations, and memory bandwidth. The computational benchmarks come from the same library [6] that I found on Github while the memory bandwidth benchmark [8] is from a separate library. I chose these benchmark as I found them to give a wide and diverse scope of my computers performance and for exploring the benefits of various compiler optimizations. The following are descriptions of the benchmarks used in this study:

1. Performance-Evaluation-Benchmark (FLOPs): Run 100,000 loops of floating-point operations and measure performance using the floating-point operations per second (FLOPs) metric.
2. Performance-Evaluation-Benchmark (IOPs): Run 100,000 loops of integer operations and measure performance using the integer operations per second (IOPs) metric.
3. tinymembench: This benchmark measures peak bandwidth of sequential memory accesses. The metric I use to track performance is how many bytes can be copied per second. accesses

I ran each of the benchmarks with several compiler optimization flags and ran each benchmark and optimization combination a total of 5 times. The programs were compiled and run using the Ubuntu 20.04 OS on my personal computer which has a quad-core Intel® Core™ i7-8550U CPU @ 1.80GHz and 16 GB RAM. The following are observation and insights given by analyzing the performance of these benchmarks.

a. In Fig. 1 and 2, we observe that, when the benchmarks are ran with no optimizations (shown as O0), the performance variation is minimal. This is likely due to the ability of the processor to handle several applications simultaneously as it can run 8-threads in parallel. However, in Fig. 3, we observe that the variation in memory bandwidth is relatively high. This is likely due to unified memory leading to varying levels of congestion during the disjoint times the program was ran.

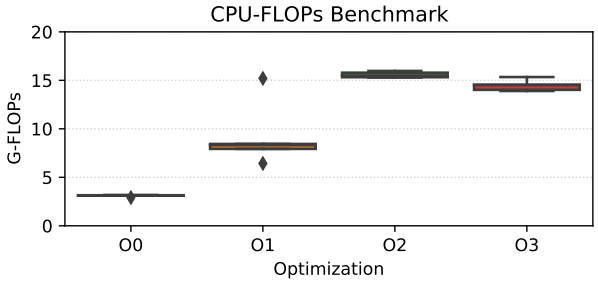


Figure 1: Shown is the performance of simple floating-point operations run in 100,000 loops.

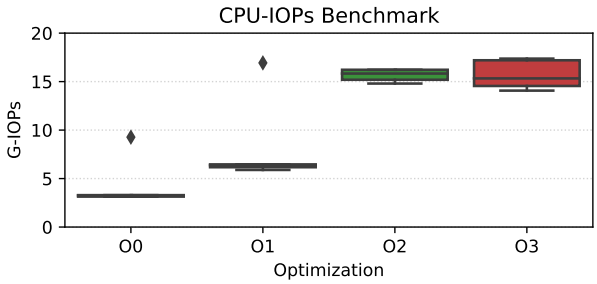


Figure 2: Shown is the performance of simple integer operations run in 100,000 loops.

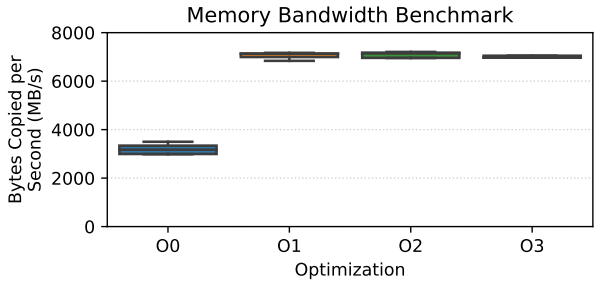


Figure 3: Shown is the performance gain (or not gain) of memory bandwidth as bytes are copied.

b. Next, I explored the compiler optimizations available to the compiler. In Fig. 1, we observe that performance increases as more optimization is done until the O3 flag is used and performance actually decreases. This is due to the fact that optimizations in O3 involve space-speed tradeoffs which slows down the memory-intensive floating-point operations. In Fig. 2, we observe this performance dropoff is much less during the O3 optimizations. This is due to integer operations being far less memory-intensive as floating-point. For these two benchmarks, the compiler optimizations significantly improved computational speed gains until memory usage outweighed it. In Fig. 3, we observe that performance improved after the O1 optimization flag but saw no gains with additional optimizations. This is due to memory access oriented optimizations being concentrated in the O1 optimization group (e.g. fstore-merging, freorder-blocks, etc.). Adding additional optimizations did not harm the performance, but are not necessary and the extra compilation time can certainly be avoided.

c. If I were to rewrite these programs to run using multi-threading, I would approach the floating-point and integer benchmarks similarly. I would build a thread structure to track the number of threads and blocks of computations that each thread should work on. Then, I would create threads to run the floating-point/integer operations in each chunk of data. This is relatively straightforward as the operations are independent of each other and have few dependencies. To implement threading in the memory bandwidth benchmark, I would do something similar to the computational benchmarks. However, I would be more cautious as I would need to program the threads to wait if another thread is accessing the same memory address. To make sure that does not happen, I would use a mutex lock. Additionally, the same data would not necessarily need to be read by both threads as the chip has cores that share an intermediate cache. Hence, the threads would not need to read the same memory address so perhaps programming the threads to return to looking through the cache when waiting at RAM would improve the performance.

```
(base) [costa.em@login-00 attempt1]$ source run_merge.sh
Time taken for execution: 279.212000 milliseconds
Array is in sorted order
Time taken for execution: 216.004000 milliseconds
Array is in sorted order
Time taken for execution: 215.971000 milliseconds
Array is in sorted order
Time taken for execution: 211.185000 milliseconds
Array is in sorted order
Time taken for execution: 207.669000 milliseconds
Array is in sorted order
```

Figure 4: Example output of the multi-threaded merge sort algorithm.

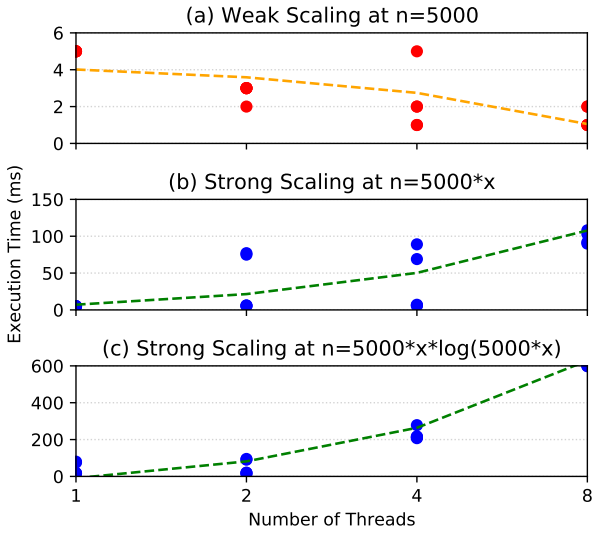


Figure 5: Weak and strong scaling of multi-threaded merge sort algorithm.

**Question 2.**

In the section, I select the merge sort sorting algorithm to sort 5,000 random integers with values 1-100,000. I use pthreads to generate a parallel version of your sorting algorithm. Originally, I developed code that was mainly my own but began having difficulties when running 8 threads so I ultimately found an alternative on the web. I selected and implemented code from a tutorial [4] that I followed. I ran this code on a node on the Discovery cluster. To ensure the sort was running, I ran a check after each sort, of which one of the outputs is shown in Fig. 4. I ran it 5 times for each configuration, or number of threads and array size combination, to increase the strength of my analysis.

a. First, I ran my program with 1, 2, 4, and 8 threads. In Fig. 5 (a), we observe that execution time of the sort decreases as the number of threads increases.

b. The main challenge of performing the threading was diagnosing a segmentation fault. This segmentation fault would not occur when using the gcc debugger. Eventually, I just completely redid the code through other code in a tutorial I found on the web. After that, I faced no challenges.

c. In Fig. 5 (a), we observe that the decreased execution time for more threads which is to be expected and the overhead is outweighed by the performance gains. However, when strong scaling is done, (b)

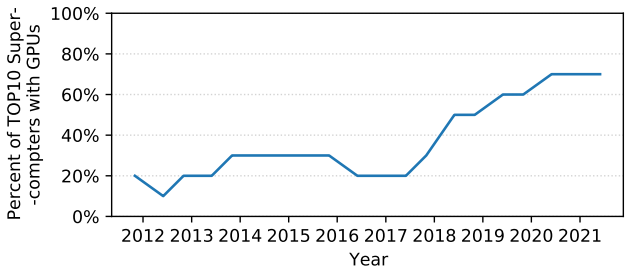


Figure 6: The number of GPU-based supercomputers in the TOP10 list increased over the previous decade.

shows that performance worsens as the data and threads scale. To rule out that this is because the time complexity of merge sort (n*logn) does not scale at the same rate as my linear strong scaling, I added additional benchmarks to scale the data at n*logn in (c). We observe that, in fact, scaling with the algorithm's time complexity does not change what we observed previously. Hence, I believe that the poor scaling is due to the overhead of the threads.

## Question 3.

Information on a Discovery login node:

a. CPU model: Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz

b. Cache memory hierarchy:

| Cache | Size | Associativity | Line Size |
|---|---|---|---|
| L1I | 32768 | 8 | 64 |
| L1D | 32768 | 8 | 64 |
| L2 | 1048576 | 16 | 64 |
| L3 | 14417920 | 11 | 64 |

c. Bandwidth and latency of the network interconnect: 10 Gb Ethernet or HDR100 InfiniBand interconnect running at 100 Gbps

d. Linux version: 3.10.0-1160.25.1.el7.x86_64

## Question 4.

One trend I noticed in the Top500 list is a higher number of systems with GPUs or GPU-like accelerators. In order to access this, I collected historical data of the top 10 ranked supercomputers in the last decade (2011-2021). As shown in Fig. 6, the percent of these computers has significantly increased over just this decade. This confirms the trend that supercomputer architects are increasingly relying on GPUs to deliver an exceptional amount of power and land systems on the TOP500 list.

One note is that the Cori supercomputer at NERSC was originally designed and built without any GPUs. However, an additional 18 cabinets with GPUs were added in 2018. NERSC recently added a new system that contains GPUs, Perlmutter, as 5 in the TOP500 list. This is one example of facilities shifting to GPU-based high-performance computers.

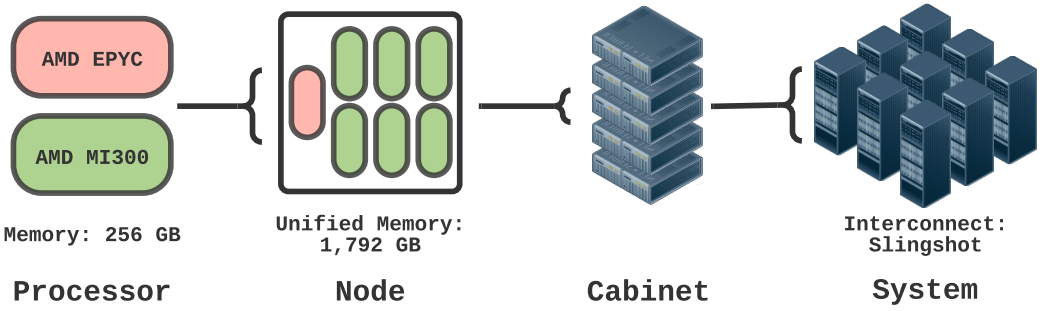


Figure 7: The number of GPU-based supercomputers in the TOP10 list increased over the previous decade.

This trend leads me to design not only a GPU-based supercomputer, but one with a uniquely high ratio of GPUs to CPUs. From my previous work with national laboratories, I know that the facilities participating in the Exascale Computing Project (ECP) are seeking to increase the GPU to CPU ratio to deliver more power with more efficiency. This is evident by tracking the architecture of systems ranked in the TOP10 from the Oak Ridge Leadership Computing Facility (OLCF)[7]. A former TOP10 system at OLCF, Titan had a GPU to CPU ratio of 1:1 and indicated a pivotal period for GPU-based supercomputers. Currently the 2 supercomputer in the TOP500 list, Summit, has a GPU to CPU ratio of 3:1. This upcoming year, we will see that ratio increase to 4:1 with the new Frontier supercomputer which is expected to be in the TOP10. As improved technology emerges, I anticipate it will enable HPC systems to further increase the GPU to CPU ratio. My design is optimistic with a GPU to CPU ratio of 6:1.

Another trend I noticed is an increased use of processors designed by AMD. In the past year alone, the use of AMD EPYC CPUs has quadrupled in the TOP500 list [5]. Additionally, as AMD ROCm software [2] enables applications to be ported and run on AMD GPUs, I anticipate this TOP500 takeover by AMD to include GPUs on GPU-based systems. This is due to AMD processors delivering exceptional performance as a lower cost while fulfilling contracts with computing facilities in a timely manner. While Intel is still used for some new TOP10 supercomputers, such as Aurora, I am mainly interested in this new AMD trend that is expected to impact the TOP10 soon with systems such as Frontier. Therefore, in my architecture diagram, I use AMD processors.

In my design, I will also be optimistic on how much memory the system can have. While AI-oriented GPU-based systems address scalable machine learning applications' massive compute consumption, a memory capacity to match is necessary. Though many software techniques exist to reduce memory usage, increasing memory while reducing cost is another desired trend in the TOP500. Additionally, I will use a Slingshot network interconnect as it has dragonfly topology, adaptive routing, and low latency. As I am using AMD processors, the processor interconnect will use AMD's Infinity Fabric with intra-node memory coherence.

My design is shown in Fig. 7. To have some fun, I use theoretical AMD Instinct MI300 GPUs that infers improvement on the new HPC-oriented MI200 design. Each node will have 6 MI300 GPUs and 1 AMD EPYC CPUs. The CPU will have 64 cores*tomorrow (9/21)

-Sara with 128 thread and 256 GB memory. I chose this based on the best performing CPUs in the AMD EPYC 7002 Series Processors [1] line. My MI300 GPUs will have 256 GB of memory, double of MI200s. I believe this to be realistic as MI200 quadruples the memory. Each node will have a shared memory access space between the CPU and GPUs.

**Question 5.**

In the current listing for the Green500 list, the differences I noticed for the Top500 are the types of processors, chip design, and system scale. Unlike the Top500, every computer in the top 10 of the Green500 list all

use accelerators to maximize efficiency. In fact, the top computer in the Green500, the MN-3, is built for matrix operations, which is similar to a GPU. Additionally, most of the chips were designed by AMD or custom designed for efficiency. Though AMD has yet to overcome Intel in the top 10 in Top500, it has more computer on the top 10 for Green500. According to AMD [3], some of the factors that reduced the power consumption of its EPYC processors are 14nm process technology, increased instructions per clock (IPC), new power management techniques, AMD's Infinity Fabric, and power optimized circuity. These factors all make AMD chips more attractive to energy-conscious customers and system designs. Finally, I noticed the unusual distribution of systems scales. Obviously, the to be at the top of the Top500, you need a very large scale computing system. However, this is not the case for be on top of the Green500. In fact, 8 out of the top 10 in the Green500 were also in the top 10 of the Top500. However, newer systems, such as Perlmutter, are displaying care about energy efficiency. I suspect more top supercomputers will follow Perlmutter's lead as computer architects become more conscious of the monetary and environmental cost of high-performance computing systems.

## 6. References


[1] AMD. *AMD EPYC™ 7002 Series Processors*. URL: https://www.amd.com/en/processors/epyc-7002-series.

[2] AMD. *Welcome to AMD ROCm™ Platform*. URL: https://rocmdocs.amd.com/en/latest/.

[3] Nathan Brookwood. *EPYC: A Study in Energy Efficient CPU Design*. URL: https://www.amd.com/system/files/documents/The-Energy-Efficient-AMD-EPYC-Design.pdf.

[4] Malith Jayaweera. *Parallel Merge Sort with Pthreads*. URL: https://malithjayaweera.com/2019/02/parallel-merge-sort/.

[5] Dylan Martin. *AMD Quadrupled EPYC's Top 500 Supercomputer Share In A Year*. 2021. URL: https://www.crn.com/news/components-peripherals/amd-quadrupled-epyc-s-top-500-supercomputer-share-in-a-year.

[6] Arihant Raj Nagarajan. *Performance-Evaluation-Benchmark*. URL: https://github.com/arihant15/Performance-Evaluation-Benchmark/blob/master/CPU/CPUBenchmark.c#L203.

[7] OLCF. *Frontier*. URL: https://www.olcf.ornl.gov/frontier/.

[8] Siarhei Siamashka. *tinymembenchmark*. URL: https://github.com/ssvb/tinymembench.