CPSC 524 Homework 1

Jonas Katona

September 21, 2023

1 Overall

For codes and scripts dealing with each exercise in particular, I will explain these in their respective sections below, but before doing that, I will describe my general environment/tools. Finally, I use the term "we" in this report in a manner which is inclusive for the reader, and it is *not* an indication that I have worked on this assignment with anybody. I attest that all of the words in this report are my own, and that all of the scripts provided (unless otherwise indicated, e.g., for the `timing` routines) were written entirely on my own.

1.1 Makefile

After writing the C codes for Exercise 1 (`exercise_1.c` and `test_div.c`) and Exercise 2 (`exercise_2.c`), we can compile these all at once using the makefile entitled `Makefile`. `Makefile` was partially based on the default makefile provided in `/gpfs/gibbs/project/cpsc424/shared/assignments/assignment1` but modified to run multiple compiler options at once. The targets provided in `Makefile` are listed below.

- `pi_$n`: Compiles `exercise_1.c` (π approximation benchmarking) with compiler option `n`, as defined as variable `CFLAG$n`.
- `div_$n`: Compiles `test_div.c` (division benchmarking) with compiler option `n`, as defined as variable `CFLAG$n`.
- `triad`: Compiles `exercise_2.c` (vector triad kernel benchmarking) with compiler option 5, i.e., `CFLAG5` (which is the one which the problem set recommends for real codes).
- `clean`: Erases all executables and object files which could be generated by `Makefile`.
- `all`: Runs all targets in `Makefile`.

Of course, the targets `pi_$n`, `div_$n`, and `triad` are only present for internal organizational purposes; we ultimately only need to directly make the targets `all` and `clean`.

1.2 Slurm build

The Bash script `build.sh` which I sent to Slurm deserves more explanation. `build.sh` is partially based on the sample batch script provided in `/gpfs/gibbs/project/cpsc424/shared/assignments/assignment1`, but I have added some significant changes to it. At first, we start in `build.sh` by running `make clean` to reset any prior builds and then run all builds via `make all`. Then, we have all our executables ready

to be ran. Since Slurm returns one's job as a text file, I organized the results of the executables using headers and what-not and according to the exercise in question and the compiler used (for Exercise 1). I sent the script to Slurm by running `sbatch build.sh` in the directory `/home/cpsc424_jek82/project/assignments/jek82_ps1_cpsc424`. The result after submitting the job to Slurm can be found in `slurm-ps1.out-26150633` (this was generated from my last Slurm job) as follows:

```
/home/cpsc424_jek82/project/assignments/jek82_ps1_cpsc424
r918u05n01
rm -f pi* div* triad exercise*.o test_div.o
icc -o pi_1 -std=c99  exercise_1.c timing.o
icc -o pi_2 -g -O0 -fno-alias -std=c99 -fp-model precise exercise_1.c timing.o
icc -o pi_3 -g -O0 -fno-alias -std=c99 exercise_1.c timing.o
icc -o pi_4 -g -O3 -no-vec -no-simd -fno-alias -std=c99 exercise_1.c timing.o
icc -o pi_5 -g -O3 -xHost -fno-alias -std=c99 exercise_1.c timing.o
icc -o div_1 -std=c99  test_div.c timing.o
icc -o div_2 -g -O0 -fno-alias -std=c99 -fp-model precise test_div.c timing.o
icc -o div_3 -g -O0 -fno-alias -std=c99 test_div.c timing.o
icc -o div_4 -g -O3 -no-vec -no-simd -fno-alias -std=c99 test_div.c timing.o
icc -o div_5 -g -O3 -xHost -fno-alias -std=c99 test_div.c timing.o
icc -o triad -g -O3 -xHost -fno-alias -std=c99 exercise_2.c timing.o dummy.o

---RESULTS---

***Results for icc compiler option 1:***

Pi approximation benchmark:
The approximated value of pi is 3.141592644203946.
|pi_exact-pi_approx|=0.000000009385847.
sin(pi_approx)=0.000000009385847.
Wallclock time elapsed for numerical integration was 0.696505 s.
MFlops needed for numerical integration was 7178.698645 MFlops.

Division benchmark:
div=0.000000003141593.
Wallclock time elapsed for N divisions was 0.180835 s.
MFlops needed for numerical integration was 5529.902687 MFlops.

***Results for icc compiler option 2:***

Pi approximation benchmark:
The approximated value of pi is 3.141592649747010.
|pi_exact-pi_approx|=0.000000003842783.
sin(pi_approx)=0.000000003842784.
Wallclock time elapsed for numerical integration was 2.869177 s.
MFlops needed for numerical integration was 1742.660085 MFlops.

Division benchmark:
div=0.000000003141593.
Wallclock time elapsed for N divisions was 2.273424 s.
```

```
MFlops needed for numerical integration was 439.865172 MFlops.

***Results for icc compiler option 3:***

Pi approximation benchmark:
The approximated value of pi is 3.141592649747010.
|pi_exact-pi_approx|=0.000000003842783.
sin(pi_approx)=0.000000003842784.
Wallclock time elapsed for numerical integration was 2.868650 s.
MFlops needed for numerical integration was 1742.980172 MFlops.

Division benchmark:
div=0.000000003141593.
Wallclock time elapsed for N divisions was 2.273577 s.
MFlops needed for numerical integration was 439.835559 MFlops.

***Results for icc compiler option 4:***

Pi approximation benchmark:
The approximated value of pi is 3.141592649747010.
|pi_exact-pi_approx|=0.000000003842783.
sin(pi_approx)=0.000000003842784.
Wallclock time elapsed for numerical integration was 1.467979 s.
MFlops needed for numerical integration was 3406.043380 MFlops.

Division benchmark:
div=0.000000003141593.
Wallclock time elapsed for N divisions was 1.147521 s.
MFlops needed for numerical integration was 871.443733 MFlops.

***Results for icc compiler option 5:***

Pi approximation benchmark:
The approximated value of pi is 3.141592652107960.
|pi_exact-pi_approx|=0.000000001481833.
sin(pi_approx)=0.000000001481833.
Wallclock time elapsed for numerical integration was 0.679443 s.
MFlops needed for numerical integration was 7358.967728 MFlops.

Division benchmark:
div=0.000000003141593.
Wallclock time elapsed for N divisions was 0.143558 s.
MFlops needed for numerical integration was 6965.824429 MFlops.

***Results for vector triad benchmarking***
9 4155.622568990171203
19 5193.618285863734855
40 8702.285108205638608
85 10867.656052522668688
```

```
180 10722.558974114808734
378 11466.062816146051773
794 11711.854287445614318
1667 6056.260174376440773
3502 6048.697557790098472
7355 6079.853984328789920
15447 6072.269537878891242
32439 4080.848622628652265
68122 1612.825752457982162
143056 1611.068751112581594
300419 1611.089271724287300
630880 1605.804349365285361
1324849 1333.188177587673408
2782184 1010.324889019605394
5842587 907.753244385307767
12269432 862.870098396331173
25765808 843.908098138772061
54108198 835.181741266711470
113627216 830.515721504229191

***CPU architecture***
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              48
On-line CPU(s) list: 0-47
Thread(s) per core:  1
Core(s) per socket:  24
Socket(s):           2
NUMA node(s):        2
Vendor ID:           GenuineIntel
CPU family:          6
Model:               85
Model name:          Intel(R) Xeon(R) Platinum 8268 CPU @ 2.90GHz
Stepping:            7
CPU MHz:             3900.000
CPU max MHz:         3900.0000
CPU min MHz:         1200.0000
BogoMIPS:            5800.00
L1d cache:           32K
L1i cache:           32K
L2 cache:            1024K
L3 cache:            36608K
NUMA node0 CPU(s):   0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46
NUMA node1 CPU(s):   1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
```

(Yes, some of the lines above are too long, but these lines are not relevant for what will be actually used in the report, so if you must, you can check them directly by looking at `slurm-ps1.out-26150633`.) I will list what the compiler options designated above are in

Section 2 (Exercise 1).

Finally, to extract the vector triad benchmarking results (MFlops vs. N) for plotting, I wrote the Bash script `write_tmp.sh` which uses `awk` to extract the lines in `slurm-ps1.out-26150633` corresponding to the data for MFlops vs. N and pipes them into `tmp.txt`. Then, one can open up MATLAB from the command line by running `matlab`, and in MATLAB run the aptly-named script `plt.m` which takes the data in `tmp.txt` and saves the resultant plot of the data in `plt.pdf`,[1] which is shown in Figure 1.

As for the text editor I used while working in the Grace OOD virtual desktop to write all my scripts, I used Emacs, which can be accessed by running `emacs` from the command line.

1.3 C compiler and modules

Regardless of the compiler options, running `icc --version` (after loading the Intel compiler module using `module load intel/2022b`) gives the C compiler I used:

```
icc (ICC) 2021.7.1 20221019
Copyright (C) 1985-2022 Intel Corporation.  All rights reserved.
```

and running `which icc` gives the location:

```
$DIR$/intel-compilers/2022.2.1/compiler/2022.2.1/linux/bin/intel64/icc
```

where `DIR=/vast/palmer/apps/avx2/software`. As for MATLAB, I loaded that by running `module load MATLAB/2023a` and then opened it up by just running `matlab`. This is MATLAB R2023a version 2, and running `which matlab` gives the location:

```
$DIR$/MATLAB/2023a/bin/matlab
```

Finally, we have the module list after running `module list`:

```
Currently Loaded Modules:
  1) StdEnv                                    (S)
  2) GCCcore/12.2.0
  3) zlib/1.2.12-GCCcore-12.2.0
  4) binutils/2.39-GCCcore-12.2.0
  5) intel-compilers/2022.2.1
  6) numactl/2.0.16-GCCcore-12.2.0
  7) UCX/1.13.1-GCCcore-12.2.0
  8) impi/2021.7.1-intel-compilers-2022.2.1
  9) imkl/2022.2.1
 10) iimpi/2022b
 11) imkl-FFTW/2022.2.1-iimpi-2022b
--More--
 12) intel/2022b
--More--
 13) MATLAB/2023a
```

That is a lot of modules!

[1] Originally, I had intended to create a Bash script which would both extract the vector triad benchmarking results from `slurm-ps1.out-*` and then pipe these directly into a Python plotting script via `sys.stdin`, but I had issues getting the Grace install of Python3 to use `matplotlib`. I do not think that MATLAB has something like `sys.stdin` readily built in, and while it has something similar to `input()` in Python, I had issues getting it to work with two columns. Hence, I had to save `tmp.txt` as an intermediate step.

2 Exercise 1

First, let us briefly describe the C codes. They are relatively self-explanatory due to my (hopefully, sufficiently detailed) in-line comments, but I will still give a general overview of how they work.

2.1 Codes

2.1.1 exercise_1.c

We start by importing the necessary header files for the reasons specified. I defined `M_PI` (the exact value of π up to 20 decimal places, generally built-in for C) directly just in case importing the header file `math.h` failed to retrieve it (which tends to happen sometimes, depending on your compiler and the built-in implementation of `math.h`.), but of course it is ultimately unnecessary as there are other methods (which will be discussed) to check the precision of our approximated π. After that, we define `main()` and declare some initial variables to be used later and take a command line input for N, the number of quadrature points used to approximate the given integral.[2] We write the expression to be summed in turns of $\Delta x = 1/N$ (represented as `dx` in the code) to ensure that we have less divisions to deal with upon each iteration (i.e., we only need to divide by N once).

Afterwards, we run the quadrature scheme, which is timed appropriately using `timing` from the provided `timing.h`. Of course, since the integral provided approximately gives $\pi/4$, we must multiply the result by `4.0` to get π. The script then returns the following:

(1) The approximated value of π;

(2) The absolute difference between the approximated value of π and the exact value of π, `M_PI`;

(3) Sine evaluated at the approximate value of π, which we should expect to be roughly equal to the above value for (2) (and it is in all cases) since $\sin x = \sin x_0 + (\cos x_0)(x - x_0) + \mathcal{O}\left(|x - x_0|^2\right)$ for $|x - x_0| \ll 1$ and $\sin \pi = 1 + \cos \pi = 0$;

(4) The wallclock time, evaluated with the help of the provided `timing` routine from `/gpfs/gibbs/project/cpsc424/shared/utils/timing`; and

(5) The MFlops needed for the numerical integration scheme, the computation of which has been written directly as a comment within the code.

(1)-(3) were reported up to 15 decimal places; (4)-(5) were reported using the default of 6.

2.1.2 test_div.c

This script was partially based on `exercise_1.c`, and hence, the workflow is the same, except we replace a few things. Instead of running the quadrature scheme, we want to run a floating-point divide operation. In this case, because this problem deals with π, we just divide π (the exact value up to 20 decimal places, `M_PI`) by i at each ith iteration,[3] where

[2] I could have defined N to be one billion within the code, but I was not sure a priori how long that might take to run so I decided to allow one to specify it for debugging purposes.

[3] We could have let π be one, but I wanted to ensure that the compiler would not automatically evaluate this as a floating-point reciprocal, since in practice, $x/y \neq x \times (1/y)$ in floating-point arithmetic. This is also why I change the number we are dividing by upon each iteration, just to make sure that the compiler does not reuse previously computed values.

of course we allow user input for the number of iterations N. (Although, for the final build script, we let N be 1 billion as asked for.) Furthermore, the script only returns (4) and (5) above, although before these things, it also returns the final value of `div`, which should in theory be π/N.

2.2 Discussion

All in all, as shown by the results above, my script was able to estimate π within 7 or 8 decimal places regardless of the compiler used. I reckon that increasing N more at this point might not help (and if anything might make things worse), as now the error in the approximate π is likely dominated by accumulation of floating-point errors.

2.2.1 Compilers

For Problem 1, we ran our results using five different Intel compiler options. At first, we will address these relative to the MFlops (performance) and wallclock time needed to complete each program. The results can be found above, but here they are in a more neat format:

1. (π) Wallclock time: 0.696505 s; performance: 7178.698645 MFlops; (division) Wallclock time: 0.180835 s; performance: 5529.902687 MFlops;
2. (π) Wallclock time: 2.869177 s; performance: 1742.660085 MFlops; (division) Wallclock time: 2.273424 s; performance: 439.865172 MFlops;
3. (π) Wallclock time: 2.868650 s; performance: 1742.980172 MFlops; (division) Wallclock time: 2.273577 s; performance: 439.835559 MFlops;
4. (π) Wallclock time: 1.467979 s; performance: 3406.043380 MFlops; (division) Wallclock time: 1.147521 s; performance: 871.443733 MFlops;
5. (π) Wallclock time: 0.679443 s; performance: 7358.967728 MFlops; (division) Wallclock time: 0.143558 s; performance: 6965.824429 MFlops.

The numbering above corresponds to the compiler option used; these options are listed below:

1. `-std=c99`
2. `-g -O0 -fno-alias -std=c99 -fp-model precise`
3. `-g -O0 -fno-alias -std=c99`
4. `-g -O3 -no-vec -no-simd -fno-alias -std=c99`
5. `-g -O3 -xHost -fno-alias -std=c99`

(**Note:** Options 3-5 correspond to the ones suggested on the problem set.) We will talk about the differences in performance between the π-estimation and division codes in the next sub-subsection, but first, let us talk about the overall differences in performance across disparate compilers. All in all, the greatest performance in terms of both wallclock time and MFlops was option 5, with option 1 just the slightest bit behind (by a relative difference of under 3% for both time and MFlops). Option 4 was next; its performance on the π script was half as good as for options 1 and 5 (roughly twice the wallclock time, half the MFlops) and a bit over a tenth as good for the division script. Finally, the worst performers were

options 2 and 3, both of which exhibited almost an identical level of performance in terms of time and MFlops.

First off, for the worst performing compilers, it should be noted that adding option `-fp-model precise` appeared to have no change to the code. Using the value `precise` with the option `-fp-model` "disables optimizations that are not value-safe on floating-point data."[4] However, I am realizing that adding this option is entirely redundant, since letting option `-O` have a value of 0 disables *all* compiler code optimizations, including those which would not be value-safe for floating-point data. And despite all this, the approximated value of π from option 5 was still the most precise out of all the options. Why should we expect this to be the case? Would we not need to do some business with the option `-fp-model` to improve the floating-point precision?

The answer lies in `-O` and/or `-xHost`. Let us review the possible values for option `-O`.[5] In summary, `-O0` disables all code optimizations, and I could see that this would probably only be useful for easy debugging since it is very clear what the original code is doing and one does not need to parse through some potentially complicated auto-optimization rules. We did not use `-O1` (I should have used it), but this one apparently introduces slight optimizations to help improve cases where, e.g., the code size is very large, when one needs to resolve a large amount of logical branches, and when there is large data flow across multiple sources which could be improved by introducing more data locality. None of these cases apply, so I would not really expect this to get drastically different results from options 2 and 3.

While it *looks* like we did not use `-O2`, we actually did implicitly in option 1, since `-O2` is the default; this partially explains the differences in precision for our π approximation. Namely, `-O2` introduces a laundry list of optimizations for speed, such as vectorization, loop optimizations (e.g., loop unrolling and variable renaming), and interprocedural optimization (e.g., inlining and forward substitution). Many of these are rather standard optimizations (some of which were actually covered in lecture), but `-O3` introduces several more aggressive optimization methods which *might* in some specific cases actually slow down the code, such as collapsing if statements, prefetching, and **floating-point error optimization**. In particular, the Intel documentation for option `-O` states that, while `-O3` might give worse performance than `-O2` in some cases, it is "recommended for applications that have loops that heavily use floating-point calculations and process large data sets," which is certainly relevant for both the π and division scripts. In particular, the floating-point optimizations explain the improvement in precision from option 1 to options 4 and 5. Furthermore, it appears that most of the performance gain was incurred when we went from `-O0` to `-O2` rather than from `-O2` to `-O3`.

However, both compilers 4 and 5 used option `-O3`; what explains the large difference in performance? One difference between these compilers is the option `-xHost`, which enables optimizations specific to the processor being used. In particular, it enables the highest instruction set available for the processor: in particular, the optimal versions and uses of Intel AVX (advanced vector expressions) and SSE (streaming SIMD Extensions). However, the performance gain from compiler 1 to compiler 5 was minimal despite the fact that compiler 1 did not use `-xHost`, which could either indicate that `-O2` already ensures the highest level of vectorization available (in which case the small gain in performance is due to `-O3` optimizations) or that whatever higher instruction set `-xHost` activates only leads to a small performance gain (again, at most 3% in this case).

[4] https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/fp-model-fp.html

[5] https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/o-001.html

Therefore, the remaining culprits are `-no-vec` and `-no-simd`, which in combination totally turn off both AVX and SIMD. In particular, `-no-vec` turns off auto-vectorization of source code entirely (including including vectorization of array notation statements[6]) while `-no-simd` "disables vectorization of loops that have SIMD pragmas," thereby disabling the effects of AVX and SIMD libraries offered by `-O2` and `-O3`. Intel AVX increases the width of the SIMD registers from 128 bits to 256 bits, thereby in theory increasing our performance by at least twice as many flops (as we see when comparing options 4 and 5), while these SIMD registers enable single instructions to be applied concurrently to multiple operands in parallel within a single node, which essentially parallelizes and vectorizes computational procedures. In particular, Intel SSE provides these for many common arithmetic operations on single- and double-precision floating-point numbers. These hardware optimizations are especially useful for `exercise_1.c` and `test_div.c`, as both of these source codes consist of the same operations being applied to different numbers a very large number of times, and we can see how important they were in our performance results.

Finally, let us move onto the remaining options. Specifiying option `-g` just "tells the compiler to generate a level of debugging information in the object file," which should not impede our performance by much unless we are either severely constrained by memory (since this increases the size of the object file) and/or if the compiler has an unusual amount of debugging information to report. Option `-fno-alias` tells the compiler to not assume any aliasing to be present in the program, which should not really affect the performance but it could in some cases affect how the program is run (or if it runs). Finally, option `-std` merely tells the compiler to compile to a provided C language standard, which makes sense as we are running our codes on a Linux shell (or at least I am).

2.2.2 Latency of floating-point division

We see from the outputted info on the CPU architecture that the CPU I used had a clock speed of 3.9 GHz. The latency is defined as the number of clock cycles per instruction. With the given clock speed, we can then get the number of clock cycles by multiplying 3.9 GHz by the wallclock time and then multiplying by 1 billion to get things in the right units. Then, the number of instructions in all cases that were ran were 1 billion, since N was 1 billion and the code consisted of N divisions. Thus, in any case, the latency (in units of clock cycles per instruction) is equal to 3.9 times the wallclock time in seconds.

I wrote my own floating-point division benchmark code because I was not sure if the π benchmark code would actually estimate the latency of the divide operation well, but did it in the end? It turns out that the answer is more complicated than it may seem and depends entirely on the compiler. Assuming that the performance of the pi calculation is dominated by the cost of division operations (such that we only consider the divisions when computing the latency), we have the following latencies in units of clock cycles per instruction:

1. (π) 2.7163695; (division) 0.7052565;
2. (π) 11.1897903; (division) 8.8663536;
3. (π) 11.187735; (division) 8.8669503;
4. (π) 5.7251181; (division) 4.4753319;
5. (π) 2.6498277; (division) 0.5598762.

[6] https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/vec-qvec.html

For no choice of compiler are the estimated latencies within 20% of each other, although they are considerably closer for the "bad" compilers (i.e., those which do not use AVX or SIMD) vs. the ones that use AVX and SIMD. Perhaps it is reasonable to assume that the difference in latency for the "bad" compilers between the π and division programs could be attributed to the other operations in the π program (there are four more floating-point operations there for every iteration, after all), but the fact that this difference is so much larger for the compilers that used AVX/SIMD demands an explanation. It is reasonable to suspect that this is largely due to AVX/SIMD. Much of the decrease in latency across different compilers is due to the fact that the code has been vectorized—the operation(s) that occur upon each iteration in both `exercise_1.c` and `test_div.c` is (or are) being applied to multiple data points at once, thereby decreasing the latency incurred when having to reapply this operation (or these operations) separately to multiple data points. Vectorization would cause the latency to come less so from the division operation in particular and be more equally spread out across the different floating-point operations, since the out-sized complexity of reapplying division over and over again (vs. reapplying "easier" floating-point operations like addition or multiplication) becomes less prominent. Of course, we see that for compiler options 1 and 5, the difference in latency between the π and division scripts is not exactly a factor of 5, but that might still be expected as division should still be more expensive even when vectorized. Rather, the key here is that the division operations will not dominate the performance of the overall program as much.

3 Exercise 2

3.1 exercise_2.c

We start `exercise_2.c` by including the necessary header files, including the provided dummy header file `dummy.h` that contains the dummy function `dummy` we use to ensure that the kernel gets executed when we repeat the computation. In particular, we include `math.h` to get the `floor` function. From here, we define `main()` and initialize our variables, including allocating memory for the large arrays `a`, `b`, `c`, and `d` using `malloc`. We use `malloc` to allocate `max_size` doubles for each of these arrays because we know *exactly* how large these arrays will need to be at most ($N = \lfloor 2.1^{25} \rfloor$ =`max_size`), and for optimization purposes, it makes sense to reuse these arrays even as the number of elements N increases (to keep the memory addresses and preserve some data locality), but only use a subset of their elements.

After preallocating `a`, `b`, `c`, and `d`, we loop through $N = \lfloor 2.1^3 \rfloor, \lfloor 2.1^4 \rfloor, \ldots, \lfloor 2.1^{25} \rfloor$. At the start of each iteration, we initialize the first N elements of `a`, `b`, `c`, and `d` with random floating point numbers in $[0, 100]$ using the suggested functions on the problem set (`rand` and `RAND_MAX`) and by looping through the first $1, \ldots, N$ elements of each. This is where I am now realizing that my code could have been written better, even though what I did is technically still correct. First off, for the vector triad kernel as written, every element of `a` needs to be replaced anyways, and so there is no need to initialize it with random floating point numbers; we just need to preallocate the memory for it. Of course, this does *not* affect how we benchmark things, as the timing routine does not include the initialization step; rather, the entire code just took longer to run than it needed to. Secondly, as I realized after going through the compiler options in more detail, the Intel compiler options `-O2` and `-O3` already include automatic loop unrolling as an optimization step, so there was no need for me to explicitly write out the 4-way unrolling as I did for the random initialization.

From here, we use the code fragment provided to time the vector triad kernel. This works as follows: We repeat the vector triad kernel operation multiple times to ensure that we get

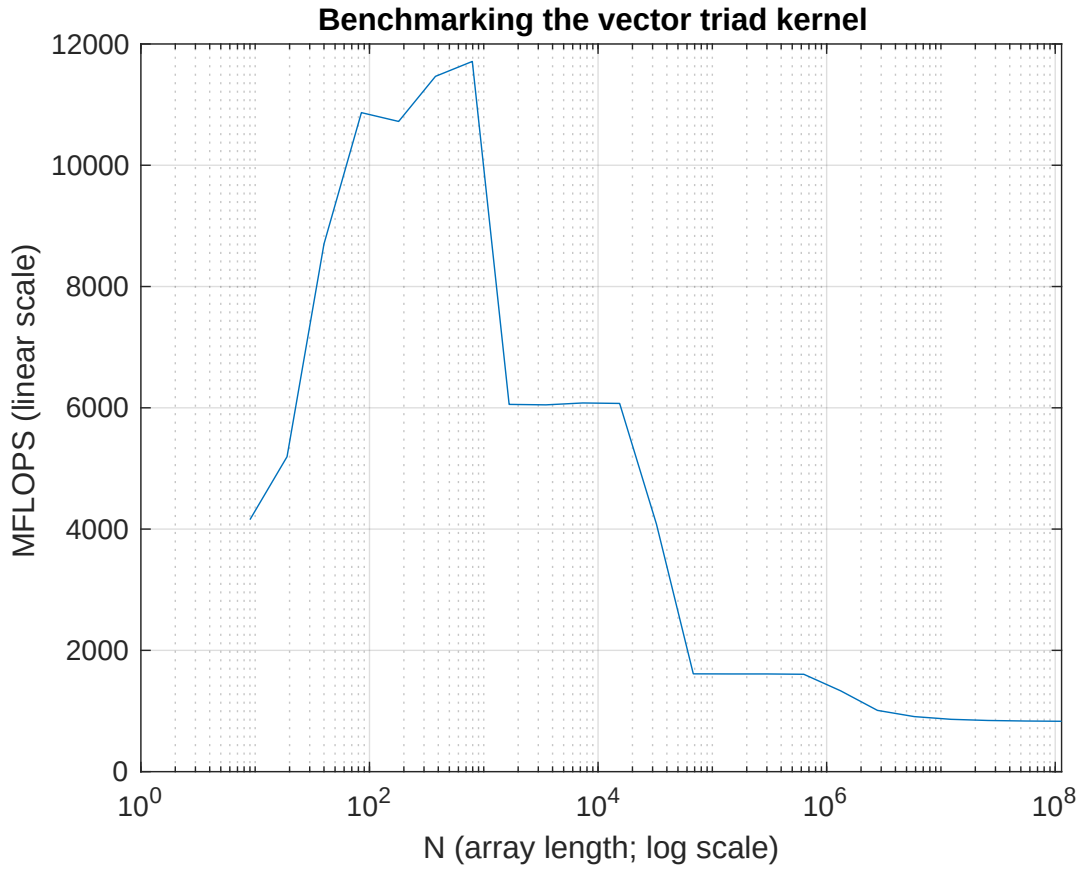

Figure 1: The figure created by running `plt.m` in MATLAB. It shows the performance in MFlops (in linear/normal scale) vs. N (number of array elements in logarithmic scale) for the vector triad kernel in C, ran with the compiler options `-g -O3 -xHost -fno-alias -std=c99` to ensure optimal efficiency (or at least as optimal as we can get with only one core and one compute node).

an "average," roughly consistent timing. This is done as follows. We continue to double the number of times we repeat this operation until the total wallclock time these repetitions take is at least 1 second. Then, we divide the final number of repetitions, `repeat`, by two because it is multiplied by two at the very end of all the repetitions each time— if the timing is over one second, then we are done, but that is checked after we have already multiplied the repetitions by two. Finally, at the end of this all, we print the MFlops (the computation of which is explained as a comment directly in the code) for the program and free up the memory which was allocated for `a`, `b`, `c`, and `d`.

3.2 Discussion

See Figure 1 for the results of the C program outlined in the previous subsection presented as MFlops vs. N. As has been mentioned multiple times in lecture, a key factor in parallelism and scalability is the amount of data being utilized; it is the principle of "use it or lose it." This applies in many ways, such as using as much of the chunks as possible in a cache line or making sure that all of the CPUs are being used. Hence, Figure 1 shows an initial increase

in the MFlops with N because of the parallel overhead; the CPU is not being fully used. As N increases initially, this overhead becomes increasingly less and less important, and we still have not exhausted any of our caches.

Memory remains a non-issue until after $N = 794$ elements and going to $N = 1667$, during which there is a large drop in performance. To understand this drop, let us see how much memory must be allocated in these two cases. There are four arrays of doubles which we allocate memory for, and each double requires 8 bytes to be stored. Thus, for running the vector triad kernel with $N = 794$ and $N = 1667$ elements, we require $4 \times 794 \times 8$ bytes = 25408 bytes = 25.408 kB and $4 \times 1667 \times 8$ bytes = 53.344 kB, respectively (setting aside other variables that have been declared, as they are negligible in comparison). As we can see from the CPU information shown on page 4 of this report from `slurm-ps1.out-26150633`, the L1 cache contains 32 kB,and hence, somewhere between $N = 794$ and $N = 1667$, the L1 cache is exhausted and so the program now needs to deal with moving memory from the L2 cache, which explains the loss in performance.

For a bit as we increase N, the performance does not change much—that is because the latency of moving data from the L2 cache dominates the performance, but this does not change by much, provided that the L2 cache has not been exhausted. The next notable drop in performance occurs from $N = 15447$ to $N = 32439$ and finally saturates by $N = 68122$. As before, this can be explained by the fact that these cases require $4 \times 15447 \times 8$ bytes = 494.304 kB, $4 \times 32439 \times 8$ bytes = 1038.048 kB, and $4 \times 68122 \times 8$ bytes = 2179.904 kB, respectively, to preallocate the arrays. The L2 cache for our CPU contains 1024 kB, which lies in between 494.304 kB and 1038.048 kB, but note that we do not see the full performance decrease until *after* $N = 32439$ since the total size of the L1 and L2 caches is $1024 + 32$ kB=1056 kB¿1038.048 kB. Hence, for $N = 32439$, the CPU can for the most part still utilize both the L1 and L2 caches with very little overflow onto the L3 cache (which is slower than both the L1 and L2 caches).

Finally, the next decrease occurs once $N \sim 10^6$, i.e., starting after $N = 630880$ (so first visible at $N = 1324849$) and then slowing down a bit once $N = 5842587$. Yet again, this happens because the entire CPU cache memory becomes saturated at that point, and so the CPU is forced to get it from the lower-level memory (which is generally much more expensive, as it is no longer in the caches!!). This is also why, after $N = 630880$, the decrease in MFlops with N does not really saturate anymore and just continues to decrease (albeit, rather slowly)—after all, the number of cache misses will have to increase in absolute terms. In particular, $N = 630880$ and $N = 1324849$ require $4 \times 630880 \times 8$ bytes = 20188.16 kB and $4 \times 1324849 \times 8$ bytes = 42395.168 kB, respectively, to preallocate the arrays, and the L1, L2, and L3 caches consist of $36608 + 1024 + 32$ kB=37664 kB of memory in total, which is less than 42395.168 kB.