15-418/618, Fall 2024
Assignment 1
Exploring Multi-Core and SIMD Parallelism

Event	Registered students	Waitlist students
Assigned:	Fri., Aug. 30	Fri., Aug. 30
Due:	Wed., Sep. 11, 23:59	Fri., Sep. 6, 23:59
Last day to handin:	Sat., Sep. 14, 23:59	Fri., Sep. 6, 23:59

Overview

In this assignment you will modify and experiment with code designed to exploit the two main forms of parallelism available on modern processors: the multiple cores that can execute programs independently, and the SIMD vector units that allow each processor to perform some of its arithmetic and memory operations on vectors of data. (You will see some of the effects of Intel hyperthreading, as well, where each core can execute the instruction streams of two programs simultaneously.)

You will also gain experience measuring and reasoning about the performance of parallel programs, a challenging, but important, skill you will use throughout this class. This assignment involves only a small amount of programming, but a lot of analysis!

This is an individual project. All handins are electronic. Your submission will consist of the code files you have modified, as well as a single document reporting your findings on the 5 problems described below. You may use any document preparation system you choose, but the final result must be stored as a single file in PDF format, submitted to Gradescope. More details on how to submit this information is provided at the end of this document.

Before you begin, please take the time to review the course policy on academic integrity at:

http://www.cs.cmu.edu/~418/academicintegrity.html

Getting started

You will need to run code on the machines in the Gates cluster for this assignment. Host names for these machines are `ghc`X`.ghc.andrew.cmu.edu`, where X is between 26 and 46. These machines contain eight 3.2 GHz Intel Core i7 processors (although dynamic frequency scaling can take them to 3.8 GHz when the chip decides it is useful and possible to do so). Each core in the processor can execute AVX2 vector instructions, supporting simultaneous execution of the same operation on multiple data values (8 in the case of single-precision data). For the curious, a complete specification for this CPU can be found at
https://ark.intel.com/products/92985/Intel-Xeon-Processor-E5-1660-v4-20M-Cache-3_20-GHz.
You can log into these machines in the cluster, or you can reach them via `ssh`.

We will grade your analysis of code run on the Gates machines; however for your own interest, you may also want to run these programs on other machines. To do this, you will first need to install the Intel SPMD Program Compiler (ISPC) available at: https://ispc.github.io/downloads.html. Feel free to include your findings from running code on other machines in your report as well, just be very clear in your report to describe the machine(s) you used.

ISPC is needed to compile many of the programs used in this assignment. ISPC is currently installed on the Gates machines in the directory `/afs/cs.cmu.edu/academic/class/15418-s24/public/ispc-v1.18.1-linux/bin`. **You will need to add this directory to your system path.** This can be done by adding the line

```
export PATH=/afs/cs.cmu.edu/academic/class/15418-s24/public/
ispc-v1.18.1-linux/bin:$PATH
```

to the end of your `~/.bashrc` file.

We will distribute the assignment starter code via a repository hosted on GitHub. Clone the assignment 1 starter code using:

```
git clone https://github.com/cmu15418f24/asst1.git
```

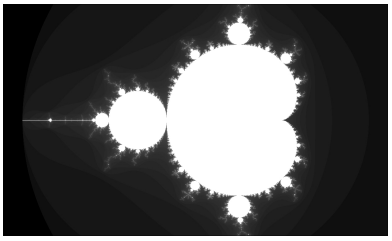
1 Problem 1: Parallel Fractal Generation Using Pthreads (15 points)

Build and run the code in the `prog1_mandelbrot_threads` directory of the Assignment 1 code base. This program produces the image file `mandelbrot-serial.ppm`, which is a visualization of a famous set of complex numbers called the Mandelbrot set. As you can see in the images below, the result is a familiar and beautiful fractal. Each pixel in the image corresponds to a value in the complex plane, and the brightness of each pixel is proportional to the computational

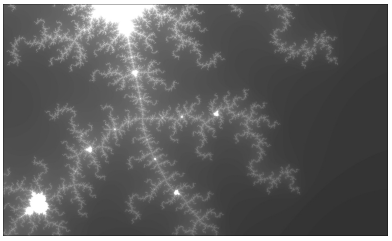
cost of determining whether the value is contained in the Mandelbrot set—white pixels required the maximum (256) number of iterations, black ones only a few iterations, and gray pixels were somewhere in between. (See function `mandel()` defined in `mandelbrot.cpp`.) You can learn more about the definition of the Mandelbrot set at http://en.wikipedia.org/wiki/Mandelbrot_set.

Use the command option "`--view` V" for V between 1 and 6 to get the different images. You can click the links below to see the different images on a browser. Take the time to do this—the images are quite striking.

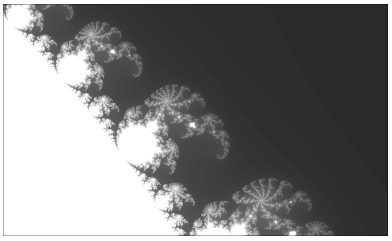
View 1
1× magnification



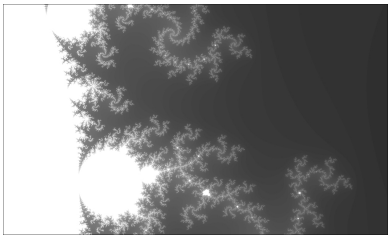
View 2
66× magnification



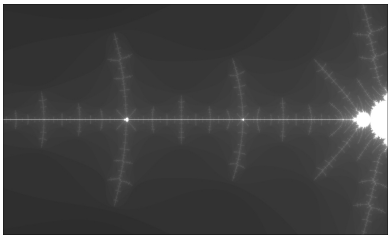
View 3
50× magnification



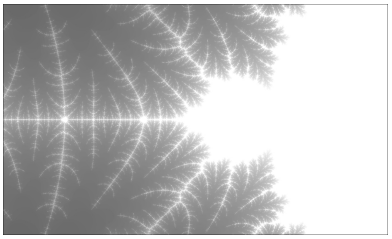
View 4
50× magnification



View 5
50× magnification



View 6
500× magnification



Your job is to parallelize the computation of the images using Pthreads. The commandline option "`--threads` T" specifies that the computation is to be partitioned over T threads. In function `mandelbrotThread()`, located in `mandelbrot.cpp`, the main application thread creates $T-1$ additional thread using `pthread_create()`. It waits for these threads to complete using `pthread_join()`. Currently, neither the launched threads nor the main thread do any computation, and so the program generates an error message. You should add code to the `workerThreadStart()` function to accomplish this task. You will not need to use of any other Pthread API calls in this assignment.

What you need to do:

1. Modify the code in `mandelbrot.cpp` to parallelize the Mandelbrot generation using two cores. Specifically, compute the top half of the image in thread 0, and the bottom half of the image in thread 1. This type of problem decomposition is referred to as *spatial decomposition* since different spatial regions of the image are computed by different processors.

2. Extend your code to utilize T threads for $T \in \{2, 4, 8, 16\}$, partitioning the image generation work into the appropriate number of horizontal blocks. You will need to modify the code in function `workerThreadStart`, to partition the work over the threads.

   Note that the processor only has 8 cores, but each core supports two hyper-threads. Also, the images have 900 rows, and so you must handle the case where the number of rows is not evenly divisible by the number of threads. In your write-up, produce a graph of speedup compared to the reference sequential implementation as a function of the number of cores used for view 1. Is speedup linear in the number of cores used? In your writeup hypothesize why this is (or is not) the case? (You may also wish to produce a graph for some of the other views to help you come up with an answer.)

3. To confirm (or disprove) your hypothesis, measure the amount of time each thread requires to complete its work by inserting timing code at the beginning and end of workerThreadStart(). How do your measurements explain the speedup graph you previously created?

4. Modify the mapping of work to threads to improve speedup to almost $8\times$ on the first two views of the Mandelbrot set (if you're close to $8\times$ that's fine, don't sweat it). You may not use any synchronization between threads. We expect you to come up with a single work decomposition policy that will work well for all thread counts; hard coding a solution specific to each configuration is not allowed! (Hint: There is a very simple static assignment that will achieve this goal, and no communication/synchronization among threads is necessary.) In your writeup, describe your approach and report the final 16-thread speedup obtained. Also comment on the difference in scaling behavior from 4 to 8 threads versus 8 to 16 threads.

What you need to turn in:

1. Your report should contain the graphs, analyses, and answers specified above.

2. Your report should describe the decomposition strategy you used to maximize speedup.

3. The archive file you submit will contain your version of the file `mandelbrot.cpp`. This file should contain the best performing code you created. Any modifications you made should follow good coding conventions, in terms of indenting, variable names, and comments.

2 Problem 2: Vectorizing Code Using SIMD Intrinsics (20 points)

Take a look at the function `clampedExpSerial()` in `prog2_vecintrin/functions.cpp` of the Assignment 1 code base. The function raises `values[i]` to the integer power given by `exponents[i]` for all elements of the input array and clamps the resulting values at 4.18. The function computes x^p based on the technique known as *exponentiation by squaring*. Whereas the

usual technique of multiplying together p copies of x requires $p-1$ multiplications, iterative squaring requires at most $2\log_2 p$ multiplications. For $p = 1000$, exponentiation by squaring requires less than 20 multiplications rather than 999. In Problem 2, your job is to vectorize this piece of code so that it can be run on a machine with SIMD vector instructions.

We won't ask you to craft an implementation using the SSE or AVX vector intrinsics that map to real vector instructions on moderns CPUs. Instead, to make things a little easier, we're asking you to implement your version using 15-418's "fake vector intrinsics" defined in `CMU418intrin.h`. The `CMU418intrin` library provides you with a set of vector instructions that operate on vector values and/or vector masks. (These functions don't translate to real vector instructions; instead we simulate these operations for you in our library, and provide feedback that makes for easier debugging.) As an example of using the 15-418 intrinsics, a vectorized version of the `abs()` function is given in `functions.cpp`. This example contains some basic vector loads and stores and manipulates mask registers. Note that the `abs()` example is only a simple example, and in fact the code does not correctly handle all inputs. (We will let you figure out why.) You may wish to read through the comments and function definitions in `CMU418intrin.h` to know what operations are available to you.

Here are a few hints to help you in your implementation:

1. Every vector instruction is subject to an optional mask parameter. The mask parameter defines which lanes have their output "masked" for this operation. A 0 in the mask indicates a lane is masked, and so its value will not be overwritten by the results of the vector operation. If no mask is specified in the operation, no lanes are masked. (This is equivalent to providing a mask of all ones.) Your solution will need to use multiple mask registers and various mask operations provided in the library.

2. You will find the `_cmu418_cntbits()` function to be helpful in this problem.

3. You must handle the case where the total number of loop iterations is not a multiple of SIMD vector width. We suggest you test your code with `./vrun -s 3`. You might find `_cmu418_init_ones()` helpful.

4. Use `./vrun -l` to print a log of executed vector instruction at the end. Use function `addUserLog()` to add customized debug information in log. Feel free to add additional `CMU418Logger.printLog()` to help you debug.

The output of the program will tell you if your implementation generates correct output. If there are incorrect results, the program will print the first one it finds and print out a table of function inputs and outputs. Your function's output is after "`output = `", which should match with the results after "`gold = `". The program also prints out a list of statistics describing utilization of the 15418 fake vector units. You should consider the performance of your implementation to be the value "Total Vector Instructions". "Vector Utilization" shows the percentage of vector lanes that are enabled.

What you need to do:

1. Implement a vectorized version of `clampedExpSerial()` as the function `clampedExpVector()` in file `functions.cpp`. Your implementation should work with any combination of input array size N and vector width W.

2. Run `./vrun -s 10000` and sweep the vector width over the values $\{2, 4, 8, 16, 32\}$. Record the resulting vector utilization. You can do this by changing the defined value of `VECTOR_WIDTH` in `CMU418intrin.h` and recompiling the code each time. How much does the vector utilization change as W changes? Explain the reason for these changes and the degree of sensitivity the utilization has on the vector width. Explain how the total number of vector instructions varies with W.

3. Extra credit: (1 point) Implement a vectorized version of `arraySumSerial()` as the function `arraySumVector()` in file `functions.cpp`. Your implementation may assume that W is a factor of the input array size N. Whereas the serial implementation has $O(N)$ span, your implementation should have at most $O(N/W + \log_2 W)$ span. You may find the `hadd` and `interleave` operations useful.

What you need to turn in:

1. Your report should contains tables giving the vector utilizations and total vector instructions for the different values of W.

2. Your report should contain the analyses and answers to the questions listed above.

3. If you did the extra credit problem, state in the report whether or not your code passed the correctness test.

4. The archive file you submit will contain your version of the file `functions.cpp`. This file should contain the best performing code you created. Any modifications you made should follow good coding conventions, in terms of indenting, variable names, and comments.

3 Problem 3: Parallel Fractal Generation Using ISPC (15 points)

The code for this problem is in the subdirectory `prog3_mandelbrot_ispc`. Now that you're comfortable with SIMD execution, we'll return to parallel Mandelbrot fractal generation. As in Problem 1, Problem 3 computes a Mandelbrot fractal image, but it achieves even greater speedups by utilizing the SIMD execution units within each of the 8 cores.

In Problem 1, you parallelized image generation by creating one thread for each processing core in the system. Then, you assigned parts of the computation to each of these concurrently executing threads. Instead of specifying a specific mapping of computations to concurrently executing

threads, Problem 3 uses ISPC language constructs to describe independent computations. These computations may be executed in parallel without violating program correctness . In the case of the Mandelbrot image, computing the value of each pixel is an independent computation. With this information, the ISPC compiler and runtime system take on the responsibility of generating a program that utilizes the CPUs collection of parallel execution resources as efficiently as possible.

You will make a simple fix to Problem 3, which is written in a combination of C++ and ISPC. (The error causes a performance problem, not a correctness one.) With the fix, you should observe performance that is over twenty times greater than that of the original sequential Mandelbrot implementation from `mandelbrotSerial()`.

3.1 Problem 3, Part 1. A Few ISPC Basics (7 of 15 points)

When reading ISPC code, you must keep in mind that, although the code appears much like C/C++ code, the ISPC execution model differs from that of standard C/C++. In contrast to C, multiple program instances of an ISPC program are always executed in parallel on the CPU's SIMD execution units. The number of program instances executed simultaneously is determined by the compiler (and chosen specifically for the underlying machine). This number of concurrent instances is available to the ISPC programmer via the built-in variable `programCount`. ISPC code can reference its own program instance identifier via the built-in `programIndex`. Thus, a call from C code to an ISPC function can be thought of as spawning a group of concurrent ISPC program instances (referred to in the ISPC documentation as a gang). The gang of instances runs to completion, then control returns back to the calling C code.

As an example, the following program uses a combination of regular C code and ISPC code to add two 1024-element vectors. As discussed in class, since each instance in a gang is independent and performs the exact same program logic, execution can be accelerated via SIMD instructions.

A simple ISPC program is given below. First, the C program, which calls the ISPC-generated code:

```
----------------------------------------------------------------
C program code: myprogram.cpp
----------------------------------------------------------------
const int TOTAL_VALUES = 1024;
float a[TOTAL_VALUES];
float b[TOTAL_VALUES];
float c[TOTAL_VALUES]

// Initialize arrays a and b here.
. . .

sum(TOTAL_VALUES, a, b, c);
```

```
// Upon return from sumArrays, result of a + b is stored in c.
```

The function `sum()` called by the C code is generated by compiling the following ISPC code:

```
-----------------------------------------------------------------
ISPC code: myprogram.ispc
-----------------------------------------------------------------
export sum(uniform int N, uniform float* a,
           uniform float* b, uniform float* c){

   // Assumption programCount divides N evenly.
   for (int i=0; i<N; i+=programCount){
      c[programIndex + i] = a[programIndex + i] + b[programIndex + i];
   }
}
```

The ISPC program code above interleaves the processing of array elements among program instances. Note the similarity to Problem 1, where you statically assigned parts of the image to threads.

However, rather than thinking about how to divide work among program instances (that is, how work is mapped to execution units), it is often more convenient, and more powerful, to instead focus only on the partitioning of a problem into independent parts. ISPCs `foreach` construct provides a mechanism to express problem decomposition. Below, the `foreach` loop in the ISPC function `sum2()` defines an iteration space where all iterations are independent and therefore can be carried out in any order. ISPC handles the assignment of loop iterations to concurrent program instances. The difference between `sum()` and `sum2()` below is subtle, but very important. `sum()` is imperative: it describes how to map work to concurrent instances. The `sum2()` function below is declarative: it specifies only the set of work to be performed.

```
-----------------------------------------------------------------
ISPC code:
-----------------------------------------------------------------
export sum2(uniform int N, uniform float* a,
            uniform float* b, uniform float* c){

   foreach (i = 0 ... N){
      c[i] = a[i] + b[i];
   }
}
```

Before proceeding, you are encouraged to familiarize yourself with ISPC language constructs by reading through the ISPC walkthrough available at http://ispc.github.io/example.html.

The example program in the walkthrough is almost exactly the same as Problem 3's implementation of `mandelbrot_ispc()` in `mandelbrot.ispc`. In the assignment code, we have changed the bounds of the `foreach` loop to yield a more straightforward implementation.

What you need to do:

1. Compile and run the program `mandelbrot.ispc`. The ISPC compiler is configured to emit 8-wide AVX2 vector instructions. What is the maximum speedup you expect given what you know about these CPUs? Why might the number you observe be less than this ideal? *Hint*: Consider the characteristics of the computation you are performing. What parts of the image present challenges for SIMD execution? Comparing the performance of rendering the different views of the Mandelbrot set may help confirm your hypothesis.

   We remind you that for the code described in this subsection, the ISPC compiler maps gangs of program instances to SIMD instructions executed on a single core. This parallelization scheme differs from that of Problem 1, where speedup was achieved by running threads on multiple cores.

3.2 Problem 3, Part 2: ISPC Tasks (8 of 15 points)

ISPC's SPMD execution model and the `foreach` mechanism facilitate the creation of programs that utilize SIMD processing. The language also provide the `launch` mechanism to utilize multiple cores in an ISPC computation, via a lightweight form of threading known as *tasks*.

See the `launch` command in the function `mandelbrot_ispc_withtasks()` in the file `mandelbrot.ispc`. This command launches multiple tasks (2 in the starter code). Each task defines a computation that will be executed by a gang of ISPC program instances. As given by the function `mandelbrot_ispc_task()`, each task computes a region of the final image. Similar to how the `foreach` construct defines loop iterations that can be carried out in any order (and in parallel by ISPC program instances), the tasks created by this launch operation can be processed in any order (and in parallel on different CPU cores).

What you need to do:

1. Run `mandelbrot_ispc` with the commandline option "`--tasks`." What speedup do you observe on view 1? What is the speedup over the version of `mandelbrot_ispc` that does not partition that computation into tasks?

2. There is a simple way to improve the performance of `mandelbrot_ispc --tasks` by changing the number of tasks the code creates. By only changing code in the function `mandelbrot_ispc_withtasks()`, you should be able to achieve performance that exceeds the sequential version of the code by about 20–22 times! How did you determine how many tasks to create? Why does the number you chose work best?

Note: Your code must correctly handle the case where the number of rows in the image is not divisible by the number of tasks.

3. Extra Credit: (1 point) What are differences between the Pthread abstraction (used in Problem 1) and the ISPC task abstraction? There are some obvious differences in semantics between the (create/join and (launch/sync) mechanisms, but the implications of these differences are more subtle. Here's a thought experiment to guide your answer: what happens when you launch 10,000 ISPC tasks? What happens when you launch 10,000 pthreads?

The smart-thinking student's question: Hey wait! Why are there two different mechanisms (`foreach` and `launch`) for expressing independent, parallelizable work to the ISPC system? Couldn't the system just partition the many iterations of `foreach` across all cores and also emit the appropriate SIMD code for the cores? *Answer*: Great question! And there are a lot of possible answers. We'll talk more in lecture.

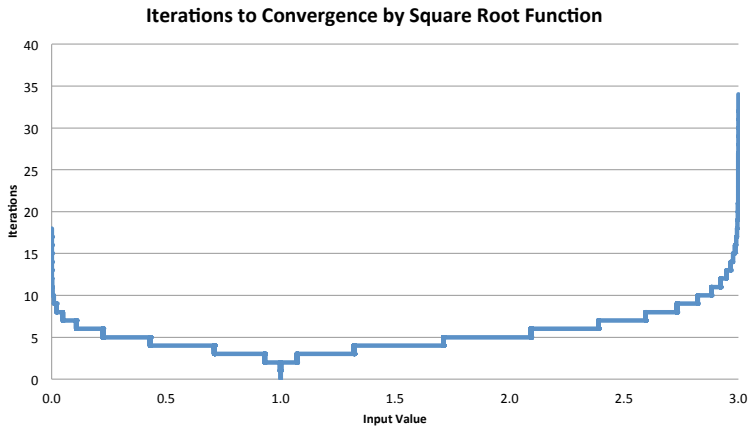
What you need to turn in:

1. Your report must contain answers to the questions listed above.
2. Your report should describe performance gains you get from both SIMD and threaded parallelism.
3. The archive file you submit will contain your version of the file `mandelbrot.ispc`. This file should contain the best performing code you created. Any modifications you made should follow good coding conventions, in terms of indenting, variable names, and comments.

4 Problem 4: Iterative Square Root (10 points)

The code for this problem is in the subdirectory `prog4_sqrt`. Problem 4 concerns the program `sqrt`, generated from an ISPC program that computes the square root of 20 million random numbers between 0 and 3. For value s, it uses the iterative Newton's method (named after Isaac Newton) to solve the equation $1/x^2 = s$. This gives a value $x \approx \sqrt{1/s}$. Multiplying x by s gives an approximation to $\sqrt{s}$.

The value 1.0 is used as the initial guess in this implementation. The graph below shows the number of iterations required for the program to converge to an accurate solution for values in the open interval $(0, 3)$. (The implementation does not converge for inputs outside this range). Notice how the rate of convergence depends on how close the solution is to the initial guess.



What you need to do:

1. Build and run `sqrt`. Report the ISPC implementation speedup for a single CPU core (no tasks) and when using all cores (with tasks). What is the speedup due to SIMD parallelization? What is the speedup due to multi-core parallelization?

2. Modify the function `initGood()` in the file `data.cpp` to generate data that will yield a very high relative speedup of the ISPC implementations. Describe why these input data will maximize speedup over the sequential version and report the resulting speedup achieved (for both the with- and without-tasks ISPC implementations). You can test this version with the commandline argument "`--data g`." Does your modification improve SIMD speedup? Does it improve multi-core speedup? Please explain why. (Optional: What is the theoretical maximum speedup, given a certain feature of modern CPUs?)

3. Modify the function `initBad()` in the file `data.cpp` to generate data that will yield a very low (less than 1.0) relative speedup for the ISPC implementation without tasks. Describe why these input data will cause the SIMD code to have very poor speedup over the sequential version and report the resulting speedup achieved (for both the with- and without-tasks ISPC implementations). You can test this version with the commandline argument "`--data b`." Does your modification improve multi-core speedup? Please explain why.

Notes and comments: When running your "very-good-case input", take a look at what the benefit of multi-core execution is. You might be surprised at how high it is. This is a hyper-threading effect.

What you need to handin:

1. Provide explanations and answers in your report.

2. The archive file you submit will contain your version of the file `data.cpp`.

5 Problem 5: BLAS `saxpy` (10 points)

The code for this problem is in the subdirectory `prog5_saxpy`. Problem 5 concerns implementations of the saxpy routine in the BLAS (Basic Linear Algebra Subproblems) library that is widely used (and heavily optimized) on many systems. The saxpy function computes the simple operation $\vec{r} = a\,\vec{x} + \vec{y}$ where a is a scalar, and $\vec{r}$, $\vec{x}$, and $\vec{y}$, are vectors of N single-precision floating-point values. (The word "saxpy" is an acronym for "single-precision a x plus y.") In program 5, $N = 20 \times 10^6$. Note that saxpy performs two math operations (one multiply, one add) for every three elements used. The saxpy function is a trivially parallelizable computation and features predictable, regular data access and predictable execution cost.

What you need to do:

1. Compile and run the `saxpy` program. The program will report the performance of ISPC (without tasks) and ISPC (with tasks) implementations of saxpy. What speedup from using ISPC with tasks do you observe? Explain the performance of this program. Do you think it can be substantially improved? (For example, could you rewrite the problem to achieve near linear speedup?) Please justify your answer. Take into consideration both the memory bandwidth performance and the floating-point arithmetic performance.

2. Extra Credit: (1 point) Note that `main.cpp` computes the total number of memory bytes used as `4 * N * sizeof(float)`, even though on each step, the saxpy program loads one value from vector `X`, one from `Y` and then writes one value to the result vector. (It holds the scalar value in a register.) Why is the computation in `main.cpp` correct? *Note:* Some students have gotten hung up on this question (thinking too hard) in the past. We expect a simple answer, but the results from running this problem might trigger more questions in your head.

3. Extra Credit: (1.5 points) Does your answer to part 1 make complete sense, given what you currently know about CPU and RAM? If not, what else might be at play here and why? You'll need to do some research online about what the performance could be bounded by.

4. Extra Credit: (2 points) Improve the performance of saxpy by reducing the memory requirement to `3 * N * sizeof(float)`. Write your code by modifying the function `saxpyStreaming()` in the file `saxpyStreaming.cpp`. *Hint*: You will need to make use of the Intel intrinsics that enable the "non-temporal memory hint."

What you need to turn in:

1. Your report should contain your observations and analyses.

2. Your answer for the first extra credit problem should be in the report.

3. If you did the second extra-credit problem, your report should contain data on the performance you achieved.

4. The archive file you submit will contain your version of the file `saxpyStreaming.cpp`. This will contain your solution to the second extra-credit problem.

Hand-in Instructions

First, you should submit the writeup/report on Gradescope – you should already be added to the 15-418/618 course.

In the home directory for the assignment, execute the command

```
make handin.tar
```

This will create an archive file with some of the files your modified for the different problems.

Second, please go to Autolab at:

https://autolab.andrew.cmu.edu/courses/15418-f24

and submit the file `handin.tar`.

This assignment is not autograded, and you will not receive immediate feedback upon submission in Autolab.

You can submit multiple times, but only your final submission will be graded, and the time stamp of that submission will be used in determining any late penalties.

If you are enrolled in the course (on SIO), but not registered on Autolab and/or Gradescope, please let the course staff know in a private post on Piazza.