15-418/618, Spring 2020
Assignment 1
Exploring Multi-Core, Instruction-Level, and SIMD Parallelism

Event	Registered students	Waitlist students
Assigned:	Mon., Jan. 13	Mon., Jan. 13
Due:	Wed., Jan. 29, 11:00 pm	Wed., Jan. 22, 11:00 pm
Last day to handin:	Sat., Feb. 1	Wed., Jan. 22

Overview

In this assignment you will modify and experiment with code designed to exploit the three main forms of parallelism available on modern processors: the multiple cores that can execute programs independently, the multiple functional units that can operate in parallel, and the SIMD vector units that allow each processor to perform some of its arithmetic and memory operations on vectors of data.

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, named `report.pdf`. Make sure the report includes your name and Andrew Id. 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 47 and 86. Each of these machines has an eight-core, 3.0 GHz Intel Core i7 processor (although dynamic frequency scaling can take them

to 4.7 GHz when the chip decides it is useful and possible to do so). Each core 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/content/www/us/en/ark/products/191792/intel-core-i7-9700-processor-12 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: `ispc.github.com/`. 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 `/usr/local/depot/ispc/bin/`. You will need to add this directory to your system path.

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

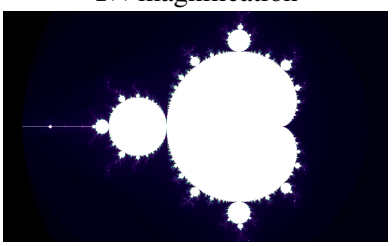
```
git clone https://github.com/beckmann-cmu/15418-asst1-s20.git
```

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

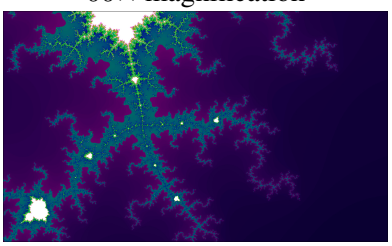
Build and run the code in the `prob1_mandelbrot_threads` directory of the Assignment 1 code base. This program produces the image file `mandelbrot-v`V`-serial.ppm`, where V is the view index. This image is a visualization of a famous set of complex numbers called the Mandelbrot set. As you can see in 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, dark ones only a few iterations, and colored pixels were somewhere in between. (See function `mandel()` defined in `mandelbrot.cpp`.) You can learn more about the definition of the Mandelbrot set at `en.wikipedia.org/wiki/Mandelbrot_set`.

Use the command option "`--view` V" for V between 0 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 0 is not shown—it is all white.)

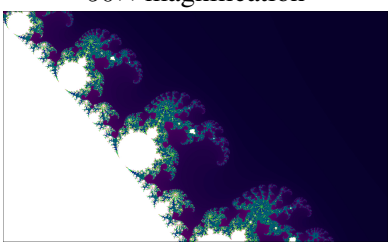
View 1
1× magnification



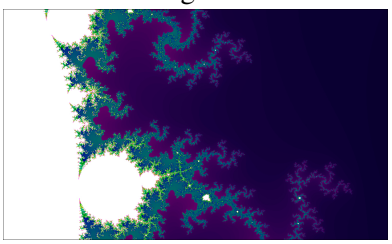
View 2
66× magnification



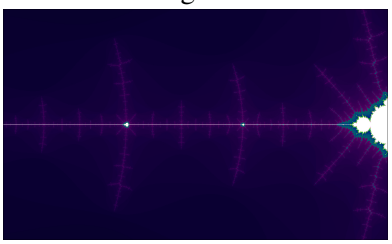
View 3
50× magnification



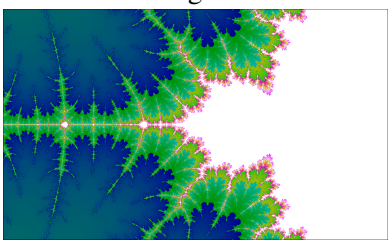
View 4
50× magnification



View 5
50× magnification



View 6
500× magnification



Your job is to parallelize the computation of the images using Pthreads. The command-line 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 has 8 cores. Also, the active images have 599 rows (with another row added to detect array overrun), 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 views 0, 1, and 2. Is speedup linear in the number of cores used? In your writeup hypothesize why this is (or is not) the case?

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 8× on view 0 and almost 8× on views 1 and 2 (if you're close to 8× 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.

As a bonus (to you, not extra credit), once you have your threaded code running properly, you can run our interactive visualization tool of the Mandelbrot set. You will find this quite addictive. The program is implemented as the file `mviz.py` in the main assignment directory. Invoke it using, as command-line arguments, the command line you give to run the `mandelbrot` program. For example, you might give the command

```
linux> ../mviz.py ./mandelbrot -t 16
```

When the program starts, it will display the list of single-character keyboard commands you can use to zoom and pan around the set. You will notice the fractal *self similarity* property, where common patterns keep occurring as you zoom deeper. You will also find that speedup you get from threading can greatly improve the interactive response.

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 `prob2_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 15.418. 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 instrinsics 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 instrinics" 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 command-line option `-l` to print a log of executed vector instruction at the end. Insert calls to function `addUserLog()` in your vector code to add customized debugging information to the log.

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.[1] 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.

[1] If it finds a mismatch in row 599, that indicates that your program overran the image array.

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: Using Instruction-level Parallelism in Fractal Generation (20 points)

Now that you have seen how SIMD execution can operate on multiple data values simultaneously, we will explore how these principles can speed up a conventional C++ program. In this case, we will exploit the parallel execution capability provided in the multiple, pipelined functional units of an out-of-order processor. This exercise will give us the chance to analyze how well a program makes use of these units and how this limits program performance.

You will want to refer to material in *Computer Systems: A Programmer's Perspective, third edition* (CS:APP3e) that you perhaps did not study carefully in whatever version of 15-213 you took. In particular, you will find Sections 3.11 (floating-point code) and Sections 5.7–5.10 (out-of-order execution and instruction-level parallelism) to be helpful. Copies of the book are on reserve in the Sorrell's Library. Don't try to get by with an earlier edition of the book.

All of the code for this problem is provided in the directory `prob3_mandelbrot_ilp`. Your job will be to measure performance and understand it. Compile and run the code on view 0. This is a useful benchmark, since all of the iterations hit the maximum limit of 256, yielding a minimum of overhead. You will find that the reference implementation, as given by the function `mandel_ref` requires around 9.75 clock cycles per iteration.

Figure 1 shows the assembly code generated for the inner loop of `mandel_ref`, with some comments added to help you understand how the integer and floating-point registers are used. Add more annotation to this code to describe how it relates to the original C++ code. (A copy of the code is provided in the file `mandel_ref_loop.s`.) **Explain:** How is the expression `2.f * z_re * z_im` is implemented?

Create a data-flow diagram indicating the data dependencies formed by successive iterations of the loop, similar to Figure 5.14(b) of CS:APP3e. You need not show the integer operations, since these do not affect the program performance.

The Gates-cluster machines are based on what Intel labels its "Coffee Lake" microarchitecture. This is very similar to Haswell microarchitecture described in Section 5.7.2 of CS:APP3e, but with slightly more and faster functional units. You can find out more about the microarchitecture in

```
        # This is the inner loop of mandel_ref
        # Parameters are passed to the function as follows:
        #   %xmm0: c_re
        #   %xmm1: c_im
        #   %edi:  count
        # Before entering the loop, the function sets registers
        # to initialize local variables:
        #   %xmm2: z_re = c_re
        #   %xmm3: z_im = c_im
        #   %eax:  i = 0

.L123:
        vmulss  %xmm2, %xmm2, %xmm4
        vmulss  %xmm3, %xmm3, %xmm5
        vaddss  %xmm5, %xmm4, %xmm6
        vucomiss        .LC0(%rip), %xmm6  # Compare to 4.f
        ja      .L126
        vaddss  %xmm2, %xmm2, %xmm2
        addl    $1, %eax
        cmpl    %edi, %eax   # Set condition codes for jne below
        vmulss  %xmm3, %xmm2, %xmm3
        vsubss  %xmm5, %xmm4, %xmm2
        vaddss  %xmm3, %xmm1, %xmm3
        vaddss  %xmm2, %xmm0, %xmm2
        jne     .L123
```

Figure 1: Assembly code for inner loop of function `mandel_ref`

en.wikichip.org/wiki/intel/microarchitectures/coffee_lake

Coffee Lake shares the same CPU core design as Intel's previous two server processors, called Kaby Lake and Skylake. You can read about the details of the core design here (search for "Skylake"):

www.agner.org/optimize/microarchitecture.pdf

Pages 151–152 of this document describe the functional units (the `wikichip.org` link has diagrams). In particular, the processor has two units for floating-point arithmetic. These are fully pipelined, able to start new operations on every clock cycle. For floating point, unit 0 can perform addition, multiplication, and dvision, while unit 1 can only perform addition and multiplication, and unit 5 can do floating-point addition. Both operations have a latency of 4 clock cycles (except for addition in unit 5, which takes 3 cycles). A separate functional unit can perform the floating-point comparison required by the `vucomiss` instruction.

Based on these latencies and the data-flow diagram you have created, determine the latency bound for function `mandel_ref`. How close does the measured performance come to reaching this bound?

It is difficult to see how to speed up the individual iterations through increased instruction-level parallelism. However, it is possible to increase the level of activity in the loop by processing multiple candidate points. This will enable increasing the throughput of the processing to yield better performance than is dictated by the latency bound.

The file `mandelbrot.cpp` contains a macro `MANDEL_BODY` and its instantiations to generate the functions `mandel_par`U for U ranging from 1 to 8. The idea is to process U points in parallel, using a style of programming similar to SIMD execution. We represent the multiple data with arrays of size U, and write conventional code that uses looping to process all U elements. As with SIMD, the loop exits only when all points have completed their iterations, using a bit vector to determine which elements should be updated. The compiler automatically unrolls all of these loops, and so the generated code is equivalent to what would be obtained by explicitly writing the computation for all U elements. (You can see the generated code in the file `mandelbrot.s`, generated when you compiled the code for this problem.) The function `mandelbrotParallel` exploits this capability by computing U adjacent rows of the array in parallel.

Running the program on view 0, you will see that it reaches a peak performance of around 3.8 clock cycles per iteration for $U = 5$. (That is, each execution of the loop requires around 19 cycles, but it performance an iteration for five points.) That's an improvement over the original performance, but perhaps not quite as significant as one may hope.

Considering the floating-point operations, what is the highest throughput bound imposed by the functional units? You should consider the bounds imposed by multiplication, addition, and the two in combination. How close does the measured performance come to reaching this bound?

If you could modify the generated assembly code, do you think you could lower the throughput bound? **Extra Credit:** (1 point) Modify the C++ code for both the reference and the parallel versions to (slightly) improve the performance of the parallel version, while still passing the functionality tests. You may alter the computed function, but only in a way that demonstrates how a better compiler could generate faster code.

As a bonus, you can run the interactive visualization tool `mviz.py` of the ILP-enhanced version the Mandelbrot computation with $U = 5$. Run with the command line:

```
linux> ../mviz.py ./mandelbrot
```

What you need to turn in:

1. An annotated version of the assembly code for the main loop of `mandel_ref`.
2. Data-flow diagram showing the loop-carried dependencies among the floating-point values.
3. An analysis of the latency bound of `mandel_ref` and how the measured performance compares.
4. An analysis of the throughput bound for the parallel versions, and how the measured performance compares.
5. Ideas for how a compiler could generate code that runs faster on this particular microarchitecture.
6. (Extra credit.) A demonstration of how the compiler could generate code that runs faster.

4 Problem 4: Parallel Fractal Generation Using ISPC (20 points)

The code for this problem is in the subdirectory `prob4_mandelbrot_ispc`. Now that you're comfortable with SIMD execution, we'll return to parallel Mandelbrot fractal generation. As in Problem 1, Problem 4 computes a Mandelbrot fractal image, but it achieves even greater speedups by utilizing the SIMD execution units within each of the 8 cores. We will also see that the ILP-enhancement techniques used in Problem 3 can, in principle, be applied to ISPC code.

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 4 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.

4.1 Problem 4, Part 1. A Few ISPC Basics (5 of 20 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)
{
    // Assumes 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.com/example.html. The example program in the walkthrough is almost exactly the same as Problem 4'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 AVX 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.

4.2 Problem 4, Part 2: Combining instruction-level and SIMD parallelism (7 of 20 points)

The floating-point functional units in the Gates cluster processors are fully vectorized. They can perform 8-wide additions and multiplications on single-precision data. Thus, the same techniques we used to exploit instruction-level parallelism can be combined with the SIMD execution targeted by ISPC.

The function `mandel_par2` provides two-way parallelism of the Mandelbrot computation. (Unfortunately, the ISPC compiler cannot handle the small loops and arrays we used in Problem 3, and so this code must explicitly duplicate each line of code. It must also pass the point data as separate scalar values, rather than as arrays.) Your job is to modify the function `mandelbrot_ispc_par2` to make use of this parallel form. As we did in Problem 3, your code should process two rows per pass. You should make use of the ISPC `foreach` construct, but modifying it to only do half as many passes. You also should handle the case where the height of the array is not a multiple of two.

How much speedup does this two-way parallelism give over the regular ISPC version? Does it vary across different inputs (i.e., different `--views`)? When is it worth the effort?

4.3 Problem 4, Part 3: ISPC Tasks (8 of 20 points)

ISPC's SPMD execution model and the `foreach` mechanism facilitate the creation of programs that utilize SIMD processing. The language also provides 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 (always 4 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 with dimensions `BLOCK_WIDTH` × `BLOCK_HEIGHT`. 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 command-line 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. The starter code always spawns 4 tasks. This works for the image size, `BLOCK_WIDTH`, and `BLOCK_HEIGHT` we have given you, but it will not work in general. Modify the calculation of `taskCount` before launching tasks to always work correctly, for any image and block size.

   Note: Your code must correctly handle the case where the block width or height does not evenly divide the image width or height.

3. There is a simple way to improve the performance of `mandelbrot_ispc --tasks` by increasing the number of tasks the code creates. By only changing values of `BLOCK_WIDTH`, and `BLOCK_HEIGHT`, you should be able to achieve performance that exceeds the sequential version of the code by about 20–22 times! What region sizes and shapes work best? (Do you prefer tall, wide, or square blocks?)

4. **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?

Some of you may be thinking: "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?"

Great question! Glad you asked! There are a lot of possible answers; we'll talk more in lecture.

As a bonus, you can run the interactive visualization tool `mviz.py` of the ISPC versions of the Mandelbrot computation. Run with either the command line:

```
linux> ../mviz.py ./mandelbrot_ispc
```

or

```
linux> ../mviz.py ./mandelbrot_ispc -t
```

You will see how the speedup improves interactivity.

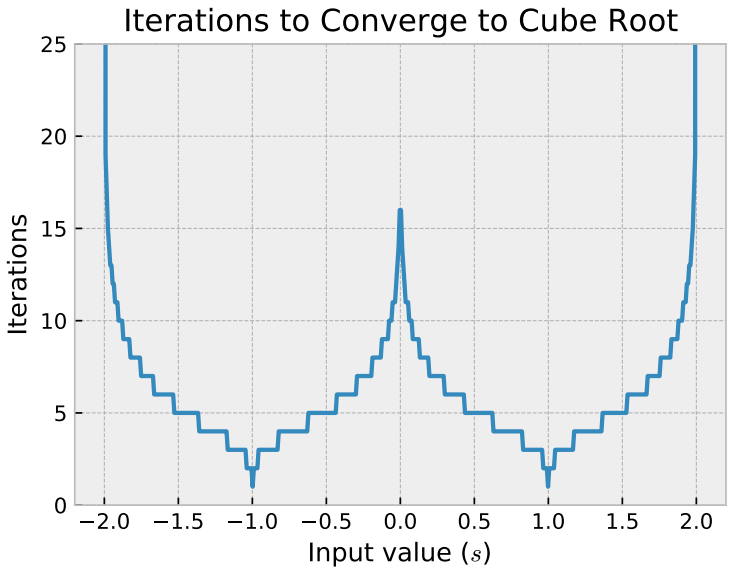
What you need to turn in:

1. Your report must contain answers to the questions listed above.
2. Your report should describe performance gains you get from SIMD, SIMD+ILP, and threaded parallelism.
3. Your answer to the extra credit question, if any.
4. 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.

5 Problem 5: Iterative Cubic Root (10 points)

The code for this problem is in the subdirectory `prob5_cuberoot`. Problem 5 concerns the program `cuberoot`, generated from an ISPC program that computes the cube 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 $1/x^3 = s^2$. This gives a value $x \approx s^{-2/3}$. Multiplying x by s gives an approximation to $s^{1/3}$.

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 $(-2, 2)$. (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 `cuberoot`. 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. (You may generate any valid data.) 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 command-line argument "`--data g`." Does your modification improve SIMD speedup? Does it improve multi-core speedup? Please explain why.

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 of the ISPC implementations. (You may generate any valid data.) 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 command-line 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.

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`.

Hand-in Instructions

As mentioned, you should have your report in a file `report.pdf` in the home directory for the assignment. (Make sure the report includes your name and Andrew Id.) In this directory, execute the command

```
make handin.tar
```

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

Completing the handin then depends on your status in the class:

- **If you are a registered student**, you should have a course Autolab account. Go to Autolab at:

  https://autolab.andrew.cmu.edu/courses/15418-s20

  and submit the file `handin.tar`.

  Important: You should also submit the PDF version of your report to Gradescope. You can do so at:

  https://www.gradescope.com/courses/78440

- **If you are on the waitlist**, you will not have an Autolab account. Instead, you should run the provided program `submit.py` as follows:

  ```
  ./submit.py -u  AndrewID
  ```

  where *AndrewID* is your Andrew ID.

In either case, 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.