15-418: Assignment 1:
Analyzing Program Performance on a Quad-Core CPU

Due: Tues Jan 31, 11:59PM

1 Overview

In this assignment you will analyze the performance of four simple parallel programs. These programs should help you develop an understanding of the two primary forms of parallel execution present in a modern multi-core CPU: (1) SIMD execution on a single core and (2) parallel execution using multiple cores. 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!

2 Environment Setup

You will need to run code on the machines in GHC 5205 for this assignment. These machines contain quad-core 3 GHz Intel Xeon processors.[1] Each Xeon core can execute SSE4 instructions which describe simultaneous execution of the same operation on four single-precision data values. For the curious, a complete specification for this CPU can be found at `http://ark.intel.com/products/39718/Intel-Xeon-Processor-W3520`.

Note: We will grade your analysis of code run on the 5205 machines, however for kicks, you may also want to run these programs on your own machine. Those of you with machines that support AVX instructions should be able to observe significant speedups. Further, running these programs on a CPU with hyper-threading enabled can give some interesting results. Feel free to include your findings in your report.

To get started:

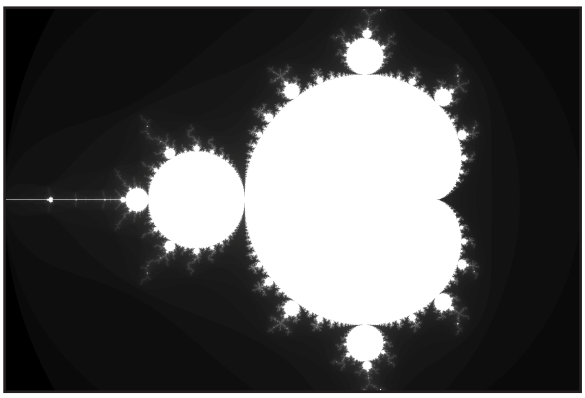
1. The Intel SPMD Program Compiler (ISPC) is needed to compile many of the programs used in this assignment. ISPC is currently installed on the Gates 5205 lab machines in `/usr/local/ispc`. You will need to add this directory to your system path.
2. Download the Assignment 1 starter code from the course directory:
   `/afs/cs.cmu.edu/academic/class/15418-s12/assignments/asst1.tgz/`

3 Assignment

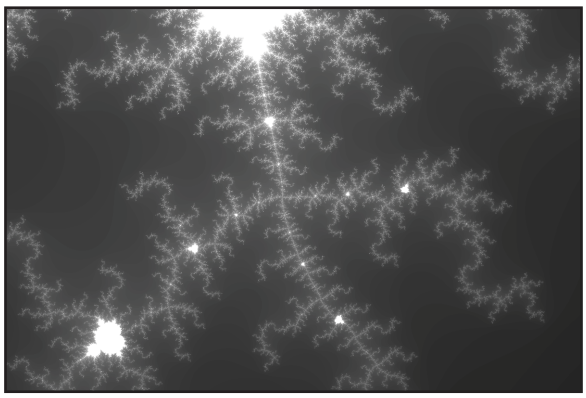
3.1 Program 1: Parallel Fractal Generation Using Pthreads (20 pts)

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`[2], which is a visualization of a famous set of complex numbers called the Mandelbrot set. As you can see in the images shown in Figure 1, the result is a famil-

[1] Note: Hyper-threading is disabled on these machines so Linux reports them four physical processing cores (not eight).
[2] To view images remotely, use `ssh -Y` and the `display` command.



View 1



View 2
(66x zoom)

Figure 1: Two visualizations of the Mandelbrot set produced by Program 1. The cost of computing each pixel is proportional to its brightness. When running programs 1 and 2, use the command option `--view 2` to set output to be view 2.

iar 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. (See function `mandelbrotSerial()` defined in `mandelbrotSerial.cpp`.) You can learn more about the definition of the Mandelbrot set at `http://en.wikipedia.org/wiki/Mandelbrot_set`.

Your job is to parallelize the computation of the image using pthreads. Starter code that spawns one additional thread is provided in the function `mandelbrotThread()` located in `mandelbrotThread.cpp`. In this function, the main application thread creates another additional pthread using `pthread_create`. It waits for this thread to complete using `pthread_join`. You will not need to make use of any other pthread API calls in this assignment.

What you need to do:

1. Modify the starter code to parallelize the Mandelbrot generation using two processors. 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 regions of the image are computed by different processors.
2. Extend your code to utilize 2, 3, and 4 threads, partitioning the image generation work accordingly. In your write-up, produce a graph of speedup as a function of the number of cores used. Is speedup linear in the number of cores used? 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 achieve to improve speedup to at least 3.5× on both views of the Mandelbrot set. You may not use any synchronization between threads. (Hint: There is a very simple static assignment that will achieve this goal. No communication/synchronization among threads is necessary.). In your writeup, describe your approach and report the 4-thread speedup obtained.

3.2 Program 2: Parallel Fractal Generation Using ISPC (20 pts)

Program 2 also implements parallel Mandelbrot fractal generation, but it achieves even greater speedups by utilizing both the Xeon CPU's four cores and the SIMD execution units within each core.

In Program 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.[3]

Instead of specifying a specific mapping of computations to concurrently executing threads, Program 2 uses ISPC language constructs to describe *independent computations*. These computations *may* be executed in parallel without violating program correctness (and indeed they will!). 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 CPU's collection of parallel execution resources as efficiently as possible.

You will make a simple fix to Program 2 which is written in a combination of C++ and ISPC (the error causes a performance problem, not a correctness one). With the correct fix, you should observe performance that is over *ten times greater* than that of the original sequential Mandelbrot implementation from `mandelbrotSerial()`.

3.2.1 Part 1: A Few ISPC Basics (10 pts)

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 we discussed in class, since each instance in a gang is independent and performing the exact same program logic, execution can be made more efficient via implementation using SIMD instructions.

A simple ISPC program is given below:

```
--------------------------------------------------------------------------
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

sumArrays(TOTAL_VALUES, a, b, c);

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

[3]Since pthreads were one-to-one with processing cores in Program 1, you effectively assigned work explicitly to cores.

```
--------------------------------------------------------------------------
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 Program 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. ISPC's `foreach` construct provides a mechanism to express problem decomposition. Below, the `foreach` loop in the ISPC function `sum` 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 example 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 Program 2'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. Read the ISPC documentation on *Choosing A Target Vector Width*: `http://ispc.github.com/perfguide.html#choosing-a-target-vector-width`.

   Notice in the Makefile we have configured ISPC with `--target=sse4-x2`, telling it to generate SSE instructions, but generate gangs of eight program instances. We've found leads to the best performance on the 5205 machines. *However, it does mean you must reason about the resulting programs as a eight-wide SIMD programs with respect to instance divergence.* (Note: this target should be changed to `avx-x2` if you want to try the programs out on a CPU with support for AVX instructions.)

2. Compile and run the program `mandelbrot_ispc`. What is the maximum speedup you expect given

what you know about the Xeon CPUs in GHC 5205? Why might the number you observe be less than this ideal? (Hint: consider the characteristics of the computation your are performing? Describe the parts of the image that 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 Program 1, where speedup was achieved by running threads on multiple cores.

3.2.2 Part 2: ISPC Tasks (10 pts)

ISPC's SPMD execution model and mechanisms like `foreach` facilitate the creation of programs that utilize SIMD processing. The language also provides an additional mechanism utilizing multiple cores in an ISPC computation. This mechanism is launching *ISPC tasks*.

See the `launch[2]` command in the function `mandelbrot_ispc_withtasks`. This command launches two tasks. 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 parameter `--tasks`. What speedup do you observe? 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 over ten times! How did you determine how many tasks to create? Why does the number you chose work best?
3. *Extra Credit: (2 points)* What are differences between the pthread abstraction (used in Program 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 tasks? What happens when you launch 10,000 pthreads?

The thinking man's (or woman'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. Come to office hours.

3.3 Program 3: BLAS `saxpy` (10 pts)

Program 3 is an implementation of the `saxpy` routine in the BLAS (Basic Linear Algebra Subproblems) library that is widely used (and heavily optimized) on many systems. `saxpy` computes the simple operation $C = \alpha A + B$, where A, B, and C are vectors of N elements (in Program 3, N=20 million). Note that `saxpy` performs two math operations (one multiply, one add) for every three elements used (two loads, one store). `saxpy` is a trivially parallelizable computation and features predictable, regular data access and predictable execution cost.

What you need to do:

1. Compile and run `saxpy`. The program will report the performance of a sequential, ISPC, and ISPC (with tasks) implementations of `saxpy`. What speedup do you observe? Explain the performance of this program. Do you think it can be improved?
2. *Extra Credit: (up to 5 points)* Improve the performance of `saxpy`. We're looking for significant speedup here, not just a few percentage points. If successful, describe how you did it (the course staff is curious) and what a best-possible implementation on these systems might achieve.

3.4 Program 4: Iterative `sqrt` (10 pts)

Program 4 is an ISPC program that computes the square root of 20 million random numbers between 0 and 3. It uses a fast, iterative implementation of square root that uses Newton's method to solve the equation $(1/x^2 - S) = 0$. The value 1.0 is used as the initial guess in this implementation. Figure 2 shows the number of iterations required for `sqrt` to converge to an accurate solution for values in the $(0, 3)$ range. (The implementation does not converge for inputs outside this range). Notice that the speed of convergence depends on the accuracy of the initial guess.

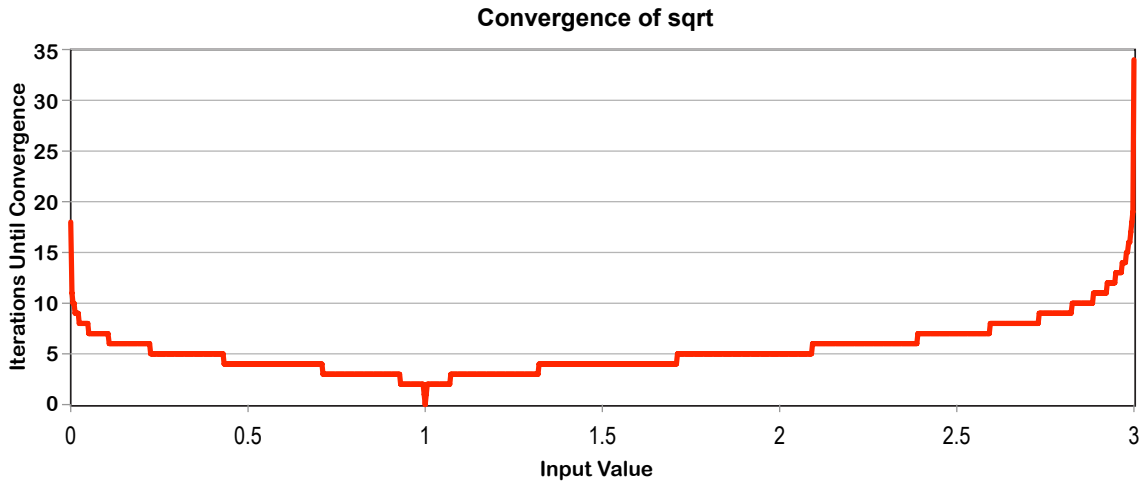


Figure 2: Convergence of `sqrt` on the range 0-3 with starting guess 1.0.

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 contents of the array `values` to improve the relative speedup of the ISPC implementations. Describe a very-good-case input and report the resulting speedup achieved (for both the with and without tasks ISPC implementations). Does your modification improve SIMD speedup? Does it improve multi-core speedup? Can you explain?
3. Construct a very-bad-case input for `sqrt`. Describe this input and report the resulting relative performance of the ISPC implementations. What is the reason for the loss in efficiency?

3.5 Hand-in Instructions

Hand-in directories have been created at:
`/afs/cs.cmu.edu/academic/class/15418-s12/handin/asst1/<ANDREW ID>/`.

Copy your `asst1/` directory into your hand-in directory. The written answers should be in `asst1/writeup.pdf`. **When handed in, all code must be compilable and runnable!** We should be able to `make` and execute your programs without manual intervention.

3.6 Resources And Notes

- Extensive ISPC documentation and examples can be found at `http://ispc.github.com/`.
- Specs for the Xeon CPUs in GHC 5205 are available at:
  `http://ark.intel.com/products/39718/Intel-Xeon-Processor-W3520`
- If you do wish to run these programs on an AVX-capable CPU, you will need to tell ISPC to generate AVX, rather than SSE instructions. Please see the Makefile comment about ISPC's `--target=avx-x2` compiler flag.
- Zooming into different locations of the mandelbrot image can be quite fascinating.