Intro HPC Serial Optimization

For this assignment, we were tasked with optimizing a serial weighted 5-point stencil program. In order to reach the ballpark times, I implemented four main optimizations: compiler choice, data access, streamlining the pipeline, and vectorization. All times were taken on Blue Crystal Phase 3. The default program started out with these times over 100 iterations compiled with gcc:

1024x1024	8.171282 s
4096x4096	346.630007 s
8000x8000	629.838448 s

Compiler

I started with the default program compiling with gcc. Gcc has various optimization flags you can set during compile time, so I tested some of these. All optimizations I tested were run on the 1024x1024 image iterated 100 times. With the -O2 flag, the program produced a time of 6.802239 s, while with the -O3 flag, I got 6.801548 s. Both have shaved over a second off the original time, but the -O3 flag performed marginally better. This is because -O3 enables more optimization options within the compiler such as -finline-functions and -funswitch-loops. I then tried looking at different compilers other than gcc. So I compiled using icc, the intel compiler (with no optimization flags), providing a significant time increase with 3.389847 s. Icc works better as it compiles the code specifically for the intel architecture, which bcp3 uses with its sandybridge processors. With a better compiler choice, I then tested the different optimization flags: -O2, -O3, and -Ofast. They produced these times respectively:

-O2 flag	3.389510 s
-O3 flag	6.815138 s
-Ofast flag	2.157853 s

From this it’s clear that -Ofast is the best optimization flag for time. This option disregards all strict standards compliance and enables the aggressive optimization flags. Strangely, -O3 produced a time twice that of the -O2 flag even though more optimizations are enabled with -O3. This happens due to one or more of the flags within -O3 being detrimental. However, -Ofast was the fastest.

Data Access & Streamline Pipeline

Next, I looked into optimizing the code itself. I used a profiler (gprof) to analyze the code and see where the biggest bottleneck was. The result was that the stencil method was taking over 99% of the time of the program. This method accessed each pixel in the image column by column, applied the weighting, then put that new pixel into a new image. The main bottleneck then was how it accessed the pixels. To solve this, I switched the for loops around so the pixels are accessed row by row. This change produced a time of 0.098972 s, already hitting the ballpark time. Accessing row by row is faster because of how the data is loaded into the cache. When a pixel is accessed, some data ahead and behind it in the image array is also loaded into cache. Scanning row by row means we can take advantage of what is stored in the cache for faster data access. By switching the datatypes from double to float also means more pixels can be stored in the cache due to a 1 byte decrease in memory per pixel. This produced a time of 0.096735 s.

Even with these changes however, the profiler still showed stencil as the biggest use of time. When the pixel is accessed, the method runs a series of conditionals to check if the pixel is on the edge of the image. Conditionals in a for loop break the pipeline in the processor, which takes up time. Therefore I had to remove the if statements. I first tried padding the outside of the image with 0's, so no pixel in the image would be on the edge. However, this produced a slower time of 0.115291 s. This is because the time saved by removing the conditionals doesn't outway the extra time accessing a bigger image. So my next attempt was to hard code the edges and corners. This worked a lot better with a time of 0.093961 s.

Vectorization

The stencil method was still taking up a lot of time, so the next optimization I did was vectorization. Firstly, I restricted the pointers in stencil so they couldn't alias. Then with icc, I simply needed to add the -vec -xHost flag when compiling. This made a time of 0.090674 s. Vectorization works by allowing the same for loop instruction to operate on multiple data points at the same time. In the stencil program, multiple pixels are being operated on simultaneously. With this, I could be confident that I have reached an efficient serial program. The times for the different sizes for 100 iterations are shown:

1024x1024	0.090674 s
4096x4096	2.479880 s
8000x8000	8.480290 s