Assignment 1: Analyzing Program Performance on an Eight-Core CPU

Program 1: Parallel Fractal Generation Using Pthreads

Q1

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 spatial regions of the image are computed by different processors.

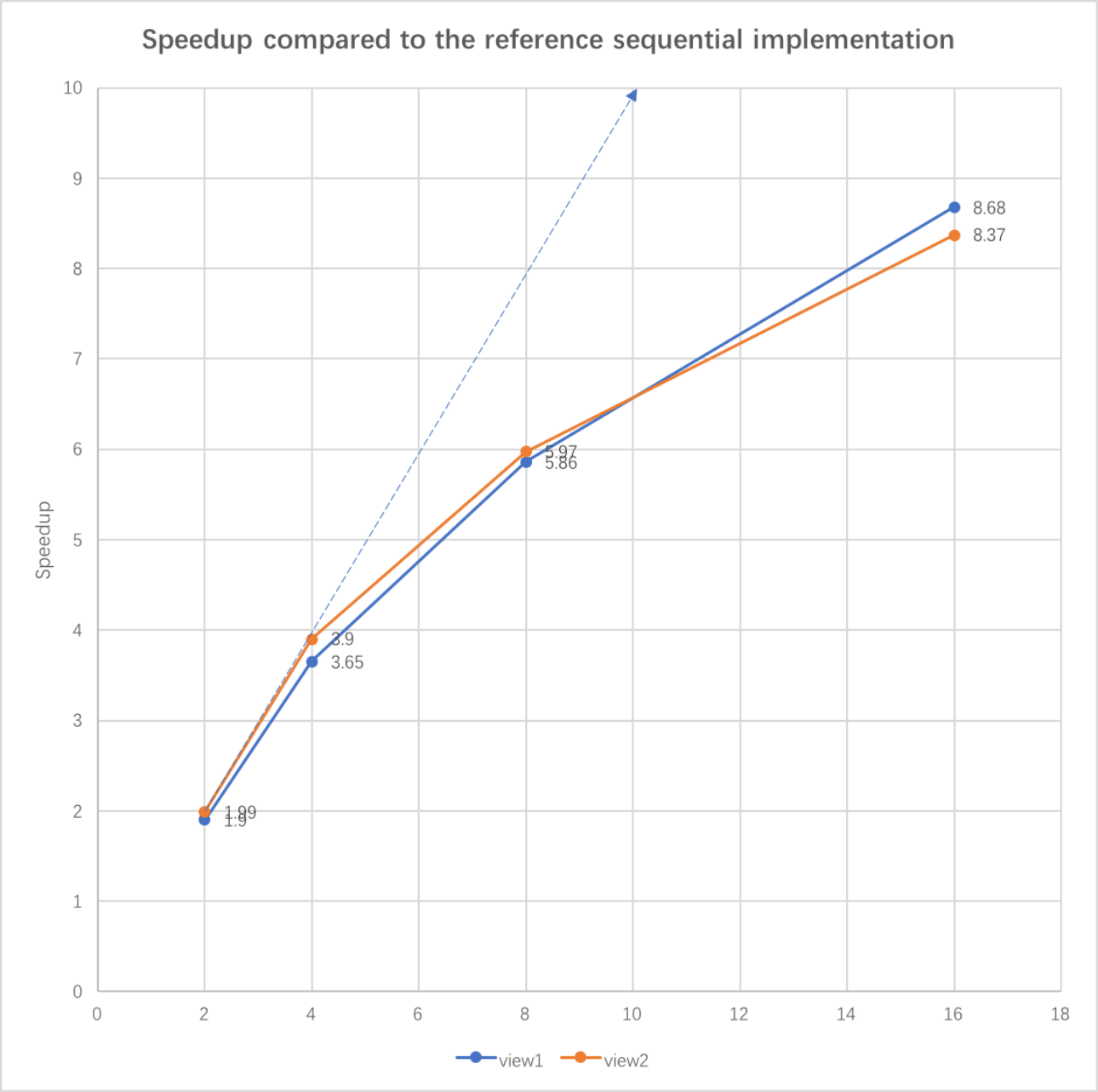
Answer

Spatial decomposition into top half and bottom half. See in the source code.

Q2

Extend your code to utilize 2, 4, 8, and 16 threads, partitioning the image generation work accordingly. Note that the processor only has 8 cores but each core supports two hyper-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 VIEW 2 to help you come up with an answer.)

Answer



No, it is far and far away from linear as we can see from the blue line in the figure above.

I think there must be some extra overhead when executing multiple threads such as unbalanced computaion cost for each thread, communication cost or content switch latency. So, it can never be linear in the number of cores.

Q3

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?

Answer

```
Round 1:
Thread 0 run time: 55.341430 ms
Thread 1 run time: 91.689474 ms
Thread 2 run time: 106.188526 ms
Thread 3 run time: 106.812619 ms

Round 2:
Thread 0 run time: 55.839582 ms
Thread 1 run time: 66.282260 ms
Thread 2 run time: 69.522140 ms
Thread 3 run time: 71.892980 ms

Round 3:
Thread 0 run time: 55.286306 ms
Thread 1 run time: 55.666321 ms
Thread 3 run time: 57.575151 ms
Thread 2 run time: 57.705027 ms

Round 4:
Thread 0 run time: 55.280241 ms
Thread 1 run time: 55.275218 ms
Thread 3 run time: 55.298766 ms
Thread 2 run time: 55.907081 ms

Round 5:
Thread 0 run time: 55.262136 ms
Thread 1 run time: 55.276773 ms
Thread 2 run time: 55.305860 ms
Thread 3 run time: 55.319156 ms
[mandelbrot thread]:            [55.369] ms
Wrote image file mandelbrot-thread.ppm
                                (3.54x speedup from 4 threads)
```

From the result above, each core completes their task for different duration, so there is some ALU that may not work while some others is still busy, the performance of cores are not fully utilized.

Q4

Modify the mapping of work to threads to achieve to improve speedup to at **about 8.5x on both views** of the Mandelbrot set (if you're close to 8.5x that's fine, don't sweat it). You may not use any synchronization between threads. We are expecting 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 vs 8 to 16 threads.

Anwser

At last the final speedup for 16 threads is 8.68X, here is my strategy. Since mandelbrot has "connect" property which means it is continuous everywhere, so we can assume that there is very small difference(The number of white pixel and black pixel) between adjcent columns since it changes continuously. So we can also assume that two adjcent columns have very similar computation cost. From this assumption, we can then assign each column to every core one by one and in that situation all cores can have a very close computaion blance, which means we can better utilize multiple cores when computing mandelbrot. The performance benefit form 4 to 8 per thread is 0.5X per thread, however, The performance benefit form 8 to 16 per thread is 0.33875X per thread, as we can see the performance benefit from adding more threads is decreasing as we add more and more thread, I think it is because there exists some overhead such as commnication or scheduling process will slowdown the performance. What's more, the imbalance load among different thread will also affect the speedup and after we create threads more than the number of threads that CPU can handle simultaneously (In this case we have 8-core CPU with 16 threads' hyper threading), the performance will not increase or even decrease.

Program 2: Vectorizing Code Using SIMD Intrinsics

Q1

Implement a vectorized version of `clampedExpSerial` in `clampedExpVector` . Your implementation should work with any combination of input array size (`N`) and vector width (`VECTOR_WIDTH`).

Answer

See in source code.

Q2

Run `./myexp -s 10000` and sweep the vector width from 2, 4, 8, to 16. Record the resulting vector utilization. You can do this by changing the `#define VECTOR_WIDTH` value in `CMU418intrin.h` . Does the vector utilization increase, decrease or stay the same as `VECTOR_WIDTH` changes? Why?

Answer

```
Vector Width:              2
Vector Utilization:        80.501238%

Vector Width:              4
Vector Utilization:        73.577597%

Vector Width:              8
Vector Utilization:        69.923057%

Vector Width:              16
Vector Utilization:        68.121626%
```

Vector utilization decreases when VECTOR_WIDTH increases from 2 to 16, because more VECTOR_WIDTH means that there are more data computed at the same time, and not all of data will finish computing process at the same time. More data computed at the same time, the more chance we have that many vectors are actually not being used because they have finished. From math perspective, *vector utilization* is calculated by *utilized vector lanes* divided by *total vector lanes*, from the data above, we can see that while *utilized vector lanes* remain mostly the same when VECTOR_WIDTH increasing from 2 to 16, *total vector lanes* increases because of more loop iterations caused by more imbalance computation cost within vector.

Q3

Extra credit: (1 point) Implement a vectorized version of `arraySumSerial` in `arraySumVector`. Your implementation may assume that `VECTOR_WIDTH` 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 / VECTOR_WIDTH + log2(VECTOR_WIDTH))` span. You may find the `hadd` and `interleave` operations useful.

Answer

Use `_cmu418_hadd_float(x, x)` and `interleave_float(x, x)` to calculate, see in source code.

Program 3: Parallel Fractal Generation Using ISPC

Part 1

Q1

Compile and run the program mandelbrot ispc. **The ISPC compiler is currently configured to emit 4-wide SSE vector instructions. (although you're encouraged to experiment with changing that compile flag to emit AVX, please answer this question for the SSE configuration.)** 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? 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.)

Answer

1. What is the maximum speedup you expect given what you know about those CPUs:

   4, however the real speedup on machine is only 2.99X.

2. Why might the number you observe be less than this ideal?

   Because it is SIMD computation, so all the data need to be performed through the same instruction, however, there some data that can be calculated(diverged) very quickly after several iterations while some data which is within Mandelbrot set that will continue being calculated untill it reaches maximum iterations. So within the same core, there will be different load on different ALU, and some ALU might be idle while others being busy, this lead to the situation that 4-wide SSE vector cannot perform 400% speed up since it is not fully utilized.

Part 2

Q1

Run `mandelbrot_ispc` with the parameter `--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?

Answer

1. 4.33X speedup from ISPC-task over serial.
2. 1.45X speedup from ISPC-task over ISPC-nontask.

Q2

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 16-18 times! How did you determine how many tasks to create? Why does the number you chose work best?

Answer

I choose 100 tasks to run, it reaches at 17.87X speedup. To fully utilize all cores of CPU, we need at least assign 16 tasks to the CPU. However, when tasks is more than threads that a CPU can handle simultaneously, tasks are queued into a list, and thread can choose one from pool when it is idle. Assigning more tasks than the cores will improve performance because more tasks can divide

computation cost more evenly and thus reduce imbalances among threads.

Q3

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 ISPC tasks? What happens when you launch 10,000 pthreads?

Answer

1. According to user guide provided by Intel, every task will be assigned to different threads when running, I think the relationship between tasks and threads is just like the relationship between threads and processes. Thread is the container of tasks, and there might be multiple tasks on the same thread.
2. Also, since tasks within the same thread will never run concurrently, so these tasks actually do not need synchronization to access under parallel execution. However, each thread need synchronization because they can run concurrently. Thinking about executing 1000 tasks compared with 1000 threads, the synchronization between threads might be very complex because we need ensure data integrity among so many threads, however, with tasks we just need to synchronize those tasks that are assigned to different threads.
3. What's more, I think enough tasks can reach a better load balance because the work is divided into smaller one, without having to consider the complex synchronization like threads.

Program 4: Iterative `sqrt`

Q1

Build and run `sqrt`. Report the ISPC implementation speedup for 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?

Answer

1. 2.64X from ISPC-nontask over serial
2. 21.7X from ISPC-task over serial

Q2

Modify the contents of the array values to improve the relative speedup of the ISPC implementations. Describe a very-good-case input that maximizes speedup over the sequential version 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 (i.e., the benefit of moving from ISPC without- tasks to ISPC with tasks)? Please explain why.

Answer

Set the values in array to 2.9999f, which means all the elements have the same computation cost.

1. Yes, it improves the performance of SIMD.
2. No, it does not improve the performance of muti task.
3. if the data in the 4-wide vector are relatively close to each other, which means that they have the same computation cost, in that situation the ALU in the same core will seldom stay idle when performing SIMD instruction. So this type of data will utilize the advantage of SIMD most. Since it only improve the performance when executing SIMD program, it does not affect the multiple core performance over SIMD program without tasks since from the computation is well distributed by assigning 64 tasks.

Q3

Construct a very-bad-case input for `sqrt` that minimizes speedup for ISPC with no tasks. Describe this input, describe why you chose it, and report the resulting relative performance of the ISPC implementations. What is the reason for the loss in efficiency? **(keep in mind we are using the --target=sse4 option for ISPC, which generates 4-wide SIMD instructions)**.

Answer

For every 4 continous elements in array, put one element with very high computation cost(2.998f) with three elements that have very low computaion cost(1.0f for example).

In that case, the computaion cost for 1.0f will be unimportant for both serial version and ISPC version, so the main cost would be computing 2.998f. Since for each 4 elements there will be a high cost, the total number of loading 2.998f for serial and ISPC version is equal. In the most ideal case, oonly one ALU in the core will be utilized while other ALU will stay idle since 1.0f is very easy to compute its root, and there will be no speedup in ISPC compared with serial version. However, in fact, for this type of input, the ISPC version would be even worse, because SIMD it own has some overhead when being executed.

Q4

Extra Credit: (up to 2 points) Write your own version of the `sqrt` function manually using AVX intrinsics. To get credit your implementation should be nearly as fast (or faster) than the binary produced using ISPC when its compile flag is set to emit AVX (`--target=avx2-i32x8`). You may find the Intel Intrinsics Guide very helpful.

Answer

See in the source code

Program 5: BLAS `saxpy`

Q1

Compile and run `saxpy`. 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 improved?

Answer

```
ghc29.ghc.andrew.cmu.edu(8 core 3.2 GHz Intel Core i7):
[saxpy serial]:         [22.314] ms     [13.356] GB/s   [1.793] GFLOPS
[saxpy ispc]:           [21.269] ms     [14.012] GB/s   [1.881] GFLOPS
[saxpy task ispc]:      [7.668] ms      [38.865] GB/s   [5.216] GFLOPS
                                (2.77x speedup from use of tasks)
                                (1.05x speedup from ISPC)
                                (2.91x speedup from task ISPC)
```

```
ghc66.ghc.andrew.cmu.edu(quad-core 2.2 GHz Intel Core i7):
[saxpy serial]:         [29.178] ms     [10.214] GB/s   [1.371] GFLOPS
[saxpy ispc]:           [28.997] ms     [10.278] GB/s   [1.379] GFLOPS
[saxpy task ispc]:      [29.994] ms     [9.936] GB/s    [1.334] GFLOPS
                                (0.97x speedup from use of tasks)
                                (1.01x speedup from ISPC)
                                (0.97x speedup from task ISPC)
```

It shows that ISPC does not improve the performance over the serial version, however, ISPC with tasks increase the performance by nearly 3X. I test it on old GHC machines, we can find that new GHC can improve the performance when using ISPC task.

The reason why it cannot be improved by SIMD is that the original serial version:

```
for (int i=0; i<N; i++) {
        result[i] = scale * X[i] + Y[i];
    }
```

It has 3 memory refrencing for every element in the array including load x, load y and store to result. It spends most of its time wating on memory even if it can exchange to another thread when waiting for data, because it is bandwidth bounded. So, this program achieves low ALU utilization even it uses SIMD or muti-thread algorithm, the time it saves on computing is far less than the time used in waiting data. I think it can be improved by using `_mm256_stream_ps` instruction because `_mm256_stream_ps` uses a non-temporal memory hint.

Q2

Extra Credit: (1 point) Note that the total memory bandwidth consumed computation in `main.cpp` is `TOTAL_BYTES = 4 * N * sizeof(float);`. Even though `saxpy` loads one element from X, one element from Y, and writes on element to `result` the multiplier by 4 is correct. Why is this the case?

Answer

As shown above, every time we calculate `result[i] = scale * X[i] + Y[i]` first we need to load both `X` and `Y`, so we use 8 bytes here. However set value to `result` cost more than 4 bytes, because we need first read `result` from memory to cache, and then change the value of result in the case, and then `result` in the cache is writen back to memory, so set value to result actually use 8 bytes in total. So, for every element in the array, we will use 16 bytes(4 * sizeof(float)) to go through the bus.

Q3

Extra Credit: (points handled on a case-by-case basis) Improve the performance of `saxpy`. We're looking for a significant speedup here, not just a few percentage points. If successful, describe how you did it and what a best-possible implementation on these systems might achieve.

Answer

Use `_mm256_stream_ps` to avoid temporal memory use and use multi thread to increase throughput. See in source code.

Finally, I use 6 threads with `_mm256_stream_ps` on GHC machine, and reach at 4X speedup.