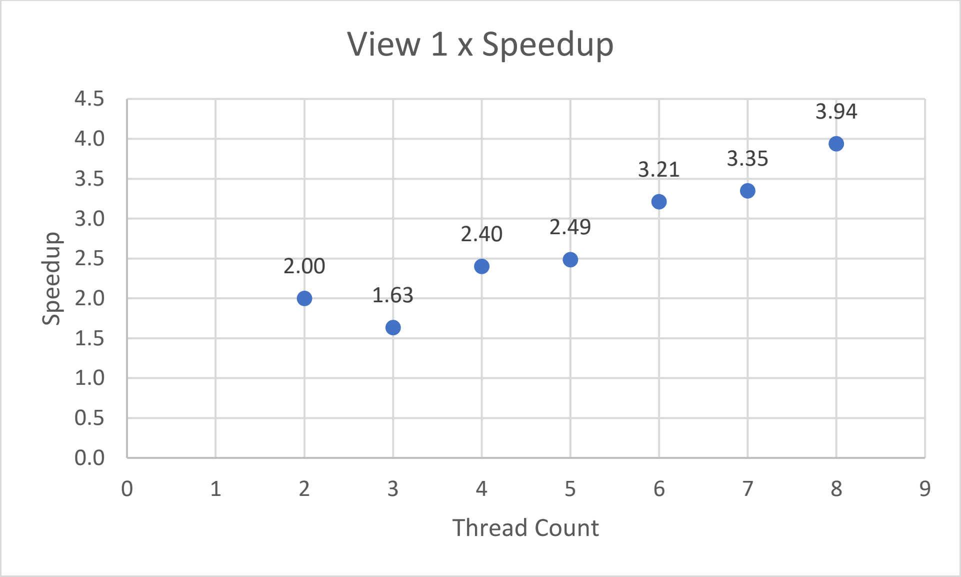
# Assignment 1 Writeup

Chenye Zhu (chenye@stanford.edu) working with Albert Wu (amhwu@stanford.edu)

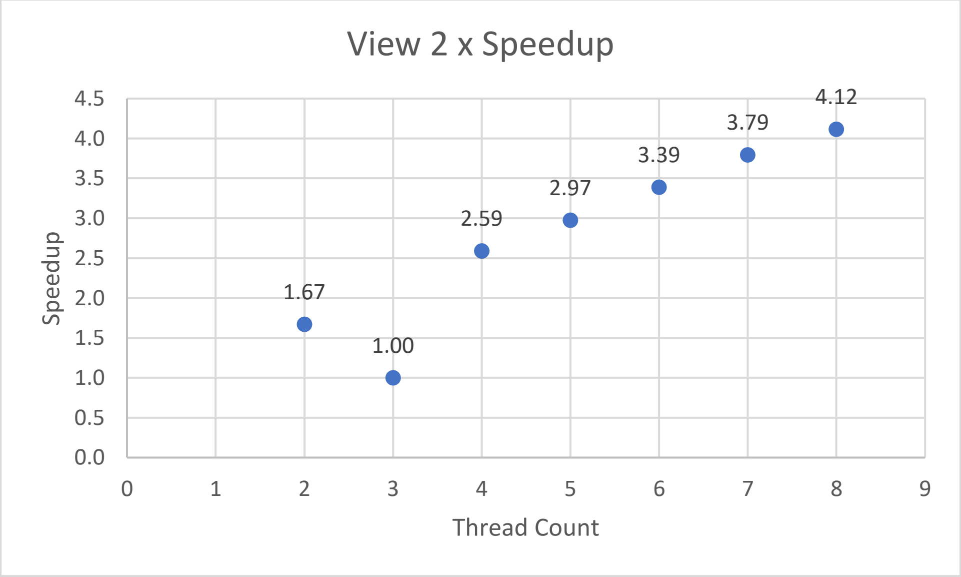
## Program 1 - Parallel Fractal Generation Using Threads

Uniformly splitting the graph horizontally into $n$ chunks, to be scheduled on $n$ cores, does not bring about linear speed up. In fact, moving from 2 cores to 3 cores hurt performance.

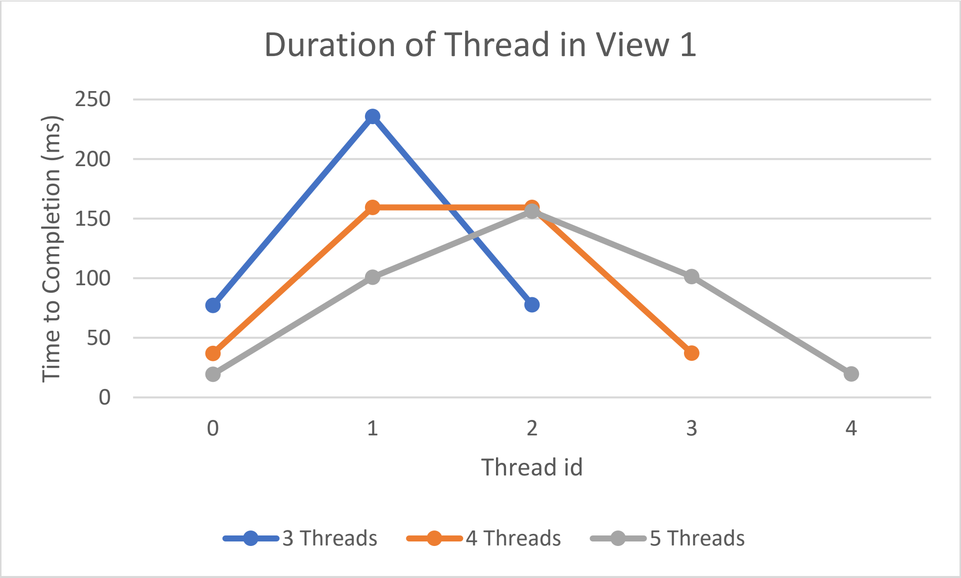
On View 1, we have following relationship



On View 2, we have this relationship

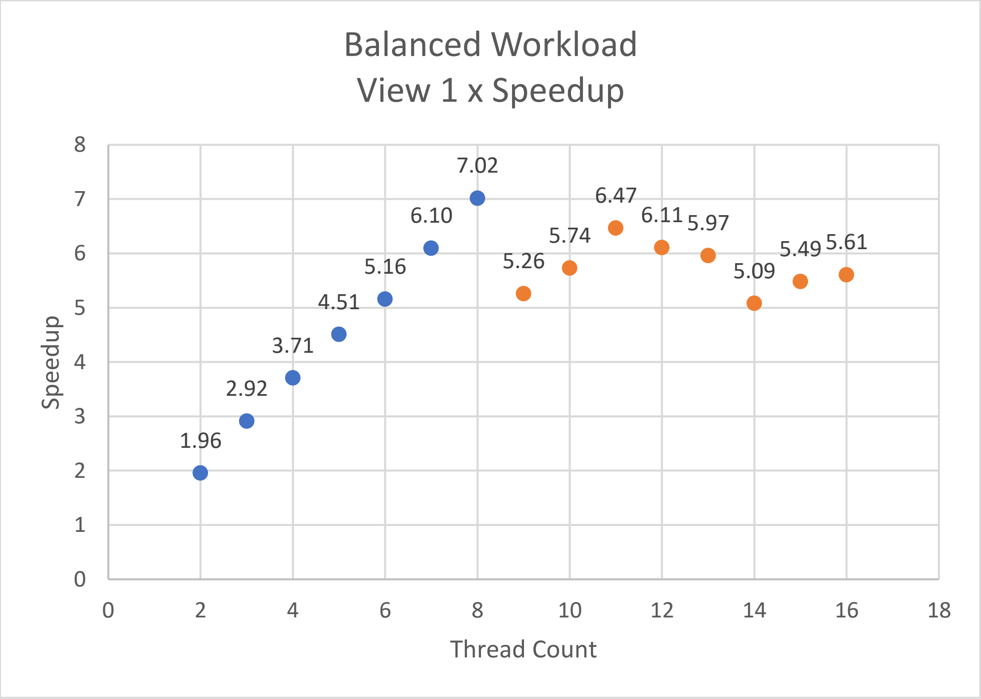


The phenomenon can be explained by uneven distribution of works among the threads. In View 1, chunks in the middle take more time to render (with significantly more elements); and in View 2, the top chunk contains more elements than the remaining chunks.



Regardless the number of threads used, in View 1, we see middle chunks takes significantly longer time to render and dominates total run time of the program.

To improve parallelism, we aim to rebalance workloads among the $n$ threads, so that each thread would wrap around the same time. First, we divide the picture into $n$ segments, with $n$ blocks in each segment horizontally, i.e., $n^2$ chunks in total. Then, each thread will take 1 block out of each of the $n$ segments. For instance, thread $i \in \{0, \cdots, n-1\}$ will take $\{i, n+i, n+2i, \cdots, (n-1)+i\}$ chunks. Notice that each thread still takes $\frac{1}{n}$ of the total work. And, since the work are selected from different segments in the graph, duration is roughly equal among threads. Now, we see a linear relationship between thread count $n$ and performance improvement.

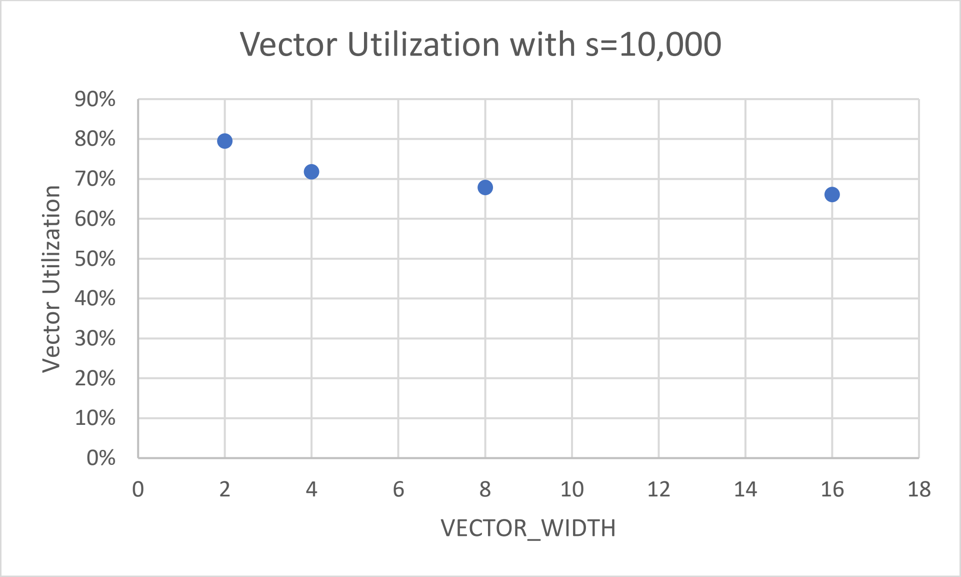


Further notice that adding more threads beyond 8 will not further boost the performance. Since Intel i7 is quad core, with 2 hardware hyper-threading, at any given moments the processor is saturated with 8 threads. Any more threads will have to wait for the first batch

## Program 2 - Vectorizing Code Using SIMD Intrinsic

### Clamped Exp Vector

Vector utilization decreases with larger VECTOR_WIDTH



Notice that in each batch, $y\_batch = y[start : start + VECTOR\_WIDTH]$, a lane $i$ in the batch is underutilized if $y[i] < max(y\_batch)$ for $max(y\_batch) - y[i]$ iterations. With large VECTOR_WIDTH, the chance of lane $i$ matched up with large $max(y\_batch)$ increases. Therefore, the overall utilization of the system decreases.

### Array Sum Vector

extra credit

First use *_cs149_vload_float* and *_cs149_vadd_float* to add the stream of windows.

Then, for *(int)log2(VECTOR_WIDTH)* times, use *_cs149_hadd_float* and *_cs149_interleave_float* to compute the addition.

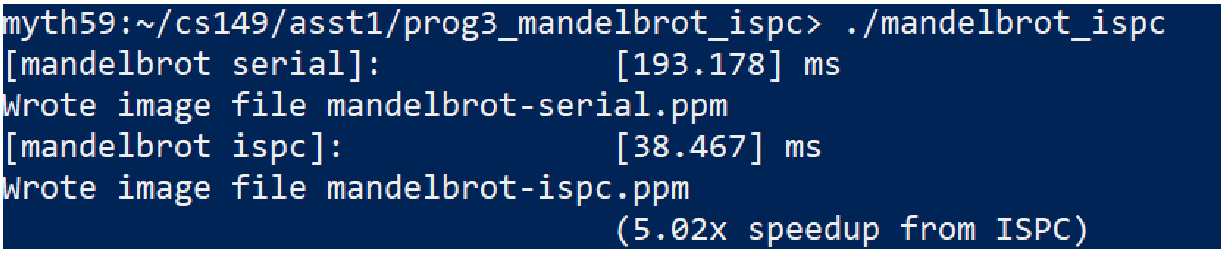
Finally, use *_cs149_vstore_float* to load into a float array and read out the first element.

## Program 3 - Parallel Fractal Generation Using ISPC

### Part 1. A Few ISPC Basics

Theoretically, with ISPC parallelism running a gang of *programCount* instances, the speed up should be *programCount* faster. Furthermore, since the ISPC compiler is currently configured to emit 8-wide AVX2 vector instructions, i.e., a single AVX2 instruction can operate on eight 32-bit floats at once, *programCount* ought to be close to 8.

However, the observed speedup is merely ~5. The difference comes from the underutilization of channels: within the same batch some units finish much faster than others and must be masked out in MSID operations.

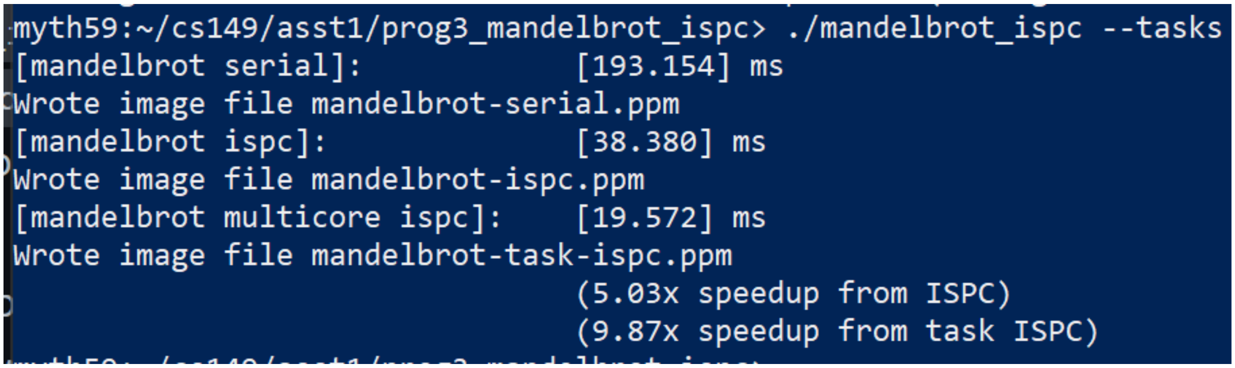


```
myth59:~/cs149/asst1/prog3_mandelbrot_ispc> ./mandelbrot_ispc
[mandelbrot serial]:            [193.178] ms
Wrote image file mandelbrot-serial.ppm
[mandelbrot ispc]:              [38.467] ms
Wrote image file mandelbrot-ispc.ppm
                                (5.02x speedup from ISPC)
```

### Part 2 - ISPC Tasks

Extra credit

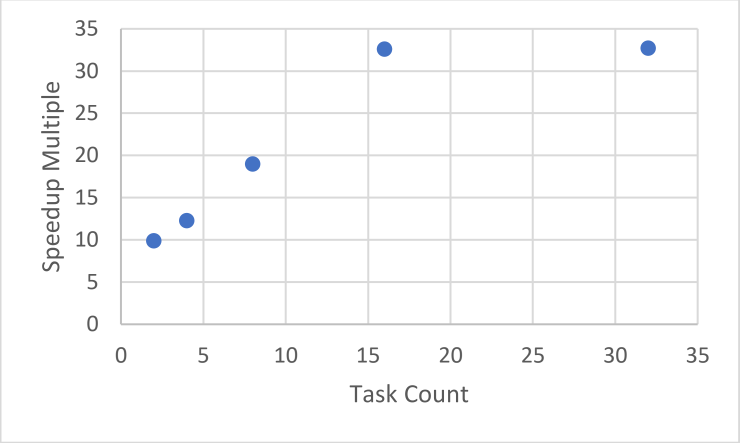
Tasks are roughly equivalent to parallelism through multithreading, with key differences to be discussed later in this part. With 2 task split ups, we get almost twice the speed up (~5x → ~10x). Compiler spawn up 2 gangs of ISPC instances, running all in parallel. Thus, the 2x faster speed up is expected.

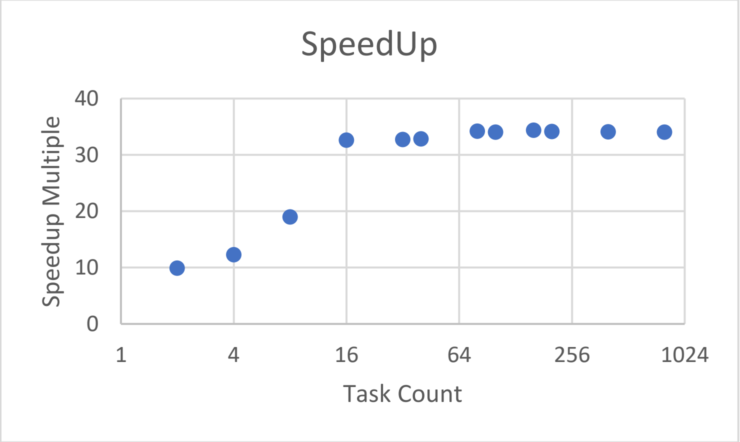


```
myth59:~/cs149/asst1/prog3_mandelbrot_ispc> ./mandelbrot_ispc --tasks
[mandelbrot serial]:            [193.154] ms
Wrote image file mandelbrot-serial.ppm
[mandelbrot ispc]:              [38.380] ms
Wrote image file mandelbrot-ispc.ppm
[mandelbrot multicore ispc]:    [19.572] ms
Wrote image file mandelbrot-task-ispc.ppm
                                (5.03x speedup from ISPC)
                                (9.87x speedup from task ISPC)
```

Speedup (almost) linearly increase with more tasks, up until 16 where the improvement plateaued around ~32x. More threads beyond that yields marginal improvement to ~34x.

- With 4 cores times 2 hyper-threading means at least 8 threads needed to utilize all hyper threads.
- 16 threads get another boost, thanks to the granularity of work delegated. Smaller chunks lead to better overall workload balance
- Notice that 4 cores times 8 SIMD machines points to a targeted speedup ratio above ~32x.
- Adding more tasks can only marginally benefit performance, as the parallelism is restricted by bottleneck resources, such as L3 cache.





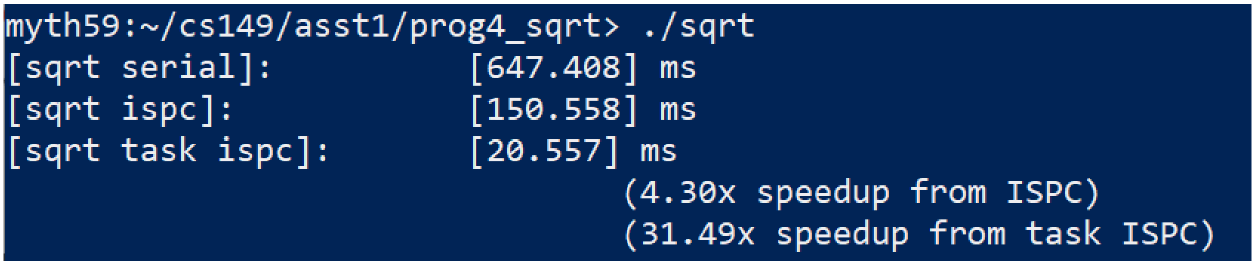
Tasks and threads are different level of abstraction. The flow goes as the programmer tells number of tasks ($N_1$) in the program, and compiler intelligently choose the number of threads ($N_2 \leq N_1$) to spawn up, and the $N_2$ threads handle $N_1$ tasks with *queues*. Specifically, if number of tasks $N_1$ is significantly larger than number of cores in the processor, the number of threads $N_2$ will be much less than $N_1$, and closer to core count.

Task incapsulates independent unit of work in a program, and therefore each task must be fully executed on 1 core and maps to one ISPC gang. However, since tasks ensure that independence of each other, different tasks can execute on different cores, or the same core. Such decision is made by the ISPC compiler targeting the specific machines.

Furthermore, there are only $N_1$ threads, and therefore much less context switch and disruptions between tasks, compared to the alternative solution of spawning up $N_2$ threads.

## Part 4 – Iterative SQRT

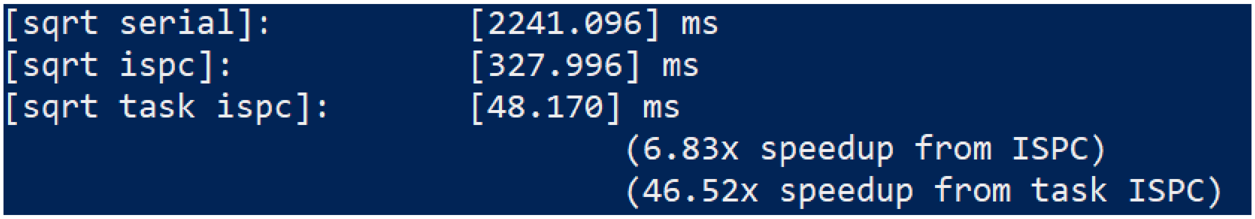
With ISPC, we observe ~4 times speedup from SIMD parallelization, and addition 8x speedup from multi-core parallelism.



```
myth59:~/cs149/asst1/prog4_sqrt> ./sqrt
[sqrt serial]:          [647.408] ms
[sqrt ispc]:            [150.558] ms
[sqrt task ispc]:       [20.557] ms
                                (4.30x speedup from ISPC)
                                (31.49x speedup from task ISPC)
```

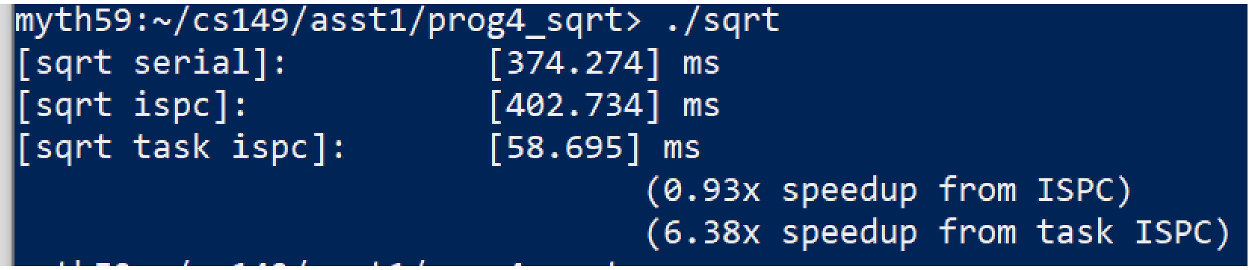
Maximum speedup comes from a list of equal values, **2.9999f (~3.f)** where each value requires max iterations to converge. It improves ISPC speedup, but not multi-core speedup.

- With quad core and 2 hyper-threading, only 8 threads can be handled in parallel. Thus, 8x for multi-core speedup is the plateau.
- With equal values, channel utilization in SIMD peaked at ~100%.
- At ~3.f each task takes max iterations, leading to maximum ratio between the computation's time and context switch cost. (Computation dominates)



```
[sqrt serial]:          [2241.096] ms
[sqrt ispc]:            [327.996] ms
[sqrt task ispc]:       [48.170] ms
                                (6.83x speedup from ISPC)
                                (46.52x speedup from task ISPC)
```

Minimum ISPC SIMD parallelism speedup comes from following distribution, every 8 items contains **seven 1.f** values and **one 2.9999f (~3.f)** value. Notice that with 1.f, value requires no iteration to converge, and with 2.9999f, value requires max iterations to converge. One eighth distribution means conditional execution leads to the worst utilization of the 8 channels in AVX2 SIMD ALUs: *only* one channel is active

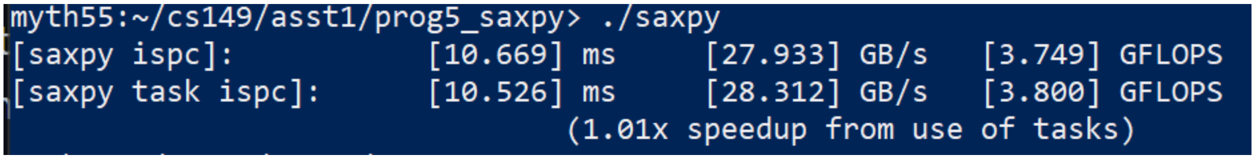


```
myth59:~/cs149/asst1/prog4_sqrt> ./sqrt
[sqrt serial]:          [374.274] ms
[sqrt ispc]:            [402.734] ms
[sqrt task ispc]:       [58.695] ms
                                (0.93x speedup from ISPC)
                                (6.38x speedup from task ISPC)
```

## # TODO Implement Vector SQRT

## Part 5 – BLAC saxpy

Extra Credit



```
myth55:~/cs149/asst1/prog5_saxpy> ./saxpy
[saxpy ispc]:           [10.669] ms     [27.933] GB/s   [3.749] GFLOPS
[saxpy task ispc]:      [10.526] ms     [28.312] GB/s   [3.800] GFLOPS
                                 (1.01x speedup from use of tasks)
```

Multi-core solution won’t help to much with the situation, as the bottle neck is reading data from memory, and significant delay introduced by cache misses. Since total memory bandwidth is limited and shared across parallelism, spawn more threads won’t help too much, as the new threads will be busy waiting for the data feed.

TOTAL_BYTES = 4 * N * sizeof(float), because for each iteration, the program loads 1 element from X, 1 element from Y. It writes 1 copy to cache, and 1 copy to memory, regardless of whether it is using write-through or write-back method. Therefore, the 2 reads and 2 writes yields four trips across the memory band.