CS149: Assignment 1 Solution:
Analyzing Program Performance on an Quad-Core CPU

These solutions pertain to the performance of ISPC 1.18.0 running on Myth.

Program 1

Question 1-3. The "obvious" strategy of parallelizing the Mandelbrot program by decomposing the image into equal-sized blocks (where each block consists of consecutive rows of the image), then assigning to each pthread does not achieve near-linear speedup. **The primary reason for this is work imbalance among the threads.** Conveniently, the image produced by the program is a visualization of the amount of work performed per pixel (brighter pixels = more work for that pixel). As you can observe from the image, threads assigned blocks near the vertical center of the image must do more work than threads assigned blocks near the top and bottom of the image. (Since there are more bright pixels in the center region). For example, when executing using four threads, threads 1 and 2 do more work than threads 0 and 3. This behavior can be diagnosed by timing the amount of time spent in each thread (as you are asked to do in the next part of the problem).

Question 4. The challenge in question 4 is to modify the assignment of image pixels to threads to ensure an even work balance among the threads. There are a number of ways to perform this assignment so that good balance is achieved, regardless of view. The simplest example is to interleave rows of the image among the threads (that is, thread `i` processes rows `j` of the image where `j % num_threads = i`. As a result, each thread is responsible for computing pixel values for rows distributed evenly throughout the image. An interleaved assignment of rows realizes about a 7.2× speedup on both views when using 8 threads. Note that speedup gained from 4 threads to 8 threads on this 4-core machine is less than 2 threads to 4 threads since 2-to-4 threads results in the program using more physical cores, but 4 to 8 only fills the extra execution context on the already used 4 cores.

Some student solutions hard-coded the number of rows assigned to each processor after observing properties of the Mandelbrot image. While these solutions achieved good speedup on one view, changing the view parameters modified the distribution of work over the image, reducing the performance of these hard-coded approaches (since the actual distribution no longer met the assumptions of the assignment). Hard-coded implementations that performed well only for view 1 received only partial credit since the assignment specifically asked you to come up with an assignment that worked well for both views (and preferably, for all views).

Program 2

Question 1. Since this was a programming question, there is no one *"correct"* solution. However, a few common problems in submitted solutions included:

- **Computing and using invalid masks:** The key challenge of this problem is determining how to set vector instruction masks to mimic the logic of each independent loop iteration in the sequential program. A commonly used correct approach used code that looked something like:

  `while (_cs149_cntbits(predicate_mask)).`

  Also, care must be taken to restore the while loop mask after exiting conditionals in the loop body.
- **Not handling non-multiple of SIMD-width array sizes:** If the array size is not a multiple of the SIMD vector width, the example code may read or write to out-of-bound array values. To avoid

this, solutions should initialize their mask with `_cs149_init_ones(min(N-i, VECTOR_WIDTH))` at the beginning of each gang.

Question 2. Assuming the input is held constant, as *vector width* increases, *vector utilization decreases* since there is an increased probability for SIMD execution divergence. Since exponents are uniform-randomly generated from 0 to 9, the expected value for *maximum exponent* within a gang (which determines number of *loop iterations*) increases as vector width increases.

Vector utilization is calculated by *utilized vector lanes* divided by *total vector lanes*. For a fixed input vector, *utilized vector lanes* remains the essentially same as the vector width is increased (it may increase slightly based on how mask operations are used and masked), *total (executed) vector lanes* is increased due to increased number of *loop iterations*. ***Note that reduced vector utilization* is also the key reason for why you will observe imperfect speedup in program 4.**

Program 3

Part 1, Question 1. The computers you ran your assignment on have quad-core processors capable of executing 8-wide AVX2 instructions. Therefore, the maximum speedup on a single core due to SIMD execution is eight. The ISPC implementation of the Mandelbrot program achieves about a 5× speedup for view 1 and a 4.2× speed up for view 2 on one core, which is less than this upper bound. **The primary reason for not achieving peak speedup is SIMD execution divergence**. In the Mandelbrot program, divergence occurs when ISPC program instances in the same gang require different number of iterations to compute pixel values. The most dramatic examples of SIMD divergence occur on the edges of the Mandelbrot fractal shape, where some instances perform only a few iterations of the while loop, and other instances in the gang execute many iterations. In terms of the Mandelbrot fractal program, this occurs when consecutive pixels in a row have significantly varying brightness. (You can observe ISPC speedup dropping on view 2 due to increased divergence.)

Program 2 is a compute-bound program with no data reuse. As a result, memory system effects on performance are negligible. Answers that discussed caching effects were not on target.

Part 2, Question 1-2. The provided code decomposes the image into two blocks and creates a ISPC task to process each block. Therefore, two tasks are created in total, and *each task* corresponds to a gang of program instances that run on *one core* of the quad-core CPU. To improve performance, the code should be modified to use all four cores of the CPU and, as was learned in Program 1, distribute work evenly to all the cores. The simplest way to achieve both of these goals is to divide the image into a large number of blocks, and create a task for each block. The blocks then get distributed to the cores dynamically by ISPC (the list of tasks is placed in a shared work queue). In program 2, dividing the image into a large large number of blocks (e.g., more than 40 or so) results in a reasonable work balance onto the eight hyper-threads available across the four cores, and an overall speedup approx. 32× that of the sequential C implementation. You certainly wanted to create at least eight tasks in this problem, so that each execution context (hyper-thread) on each core would have a task to work on. Making more, smaller granularity, tasks improves load balance.

Many students came up with a good solution to this problem (they created many tasks), but they arrived at their solution via incorrect reasoning. A common answer was to create 32 tasks, motivated by multiplying four cores per chip times the 8-wide SIMD instructions per core. While 32 tasks results in a reasonable workload distribution among the CPU's four cores, the SIMD width of the machine is not a relevant detail to the choice of task partitioning. **Remember, each ISPC task is implemented by a gang of program instances running on a single core. In ISPC, the purpose of partitioning the workload into tasks is to use all the hyperthreads on the CPU's cores. It is not to expose parallelism that enables SIMD execution.**

Small deductions were applied to answers that imprecisely mixed terminology associated with SIMD, threads, and ISPC tasks.

Program 4

Question 1. **sqrt** achieves a 4.3× speedup due to ISPC on one core. This speedup is less than the ideal value of eight because of SIMD execution divergence. (The number of iterations required for convergence is data dependent, and the starting code populates its input array with random numbers.) However, `sqrt` does demonstrate nearly perfect multi-core scaling. (Note you might have observed more than 4× faster performance with tasks due to hyperthreading.)

Question 2. To eliminate the inefficiency of SIMD execution divergence the input array should be modified *so all elements are the same value*. Ideally, this value should be a value that requires many iterations until convergence (a value very near three), thereby increasing the arithmetic intensity of the application. (If only a few iterations are required, the arithmetic intensity of the application decreases, and `sqrt`'s behavior tends towards that of `saxpy` and becomes bandwidth bound (see Program 5). Providing the program an input of all 2.999's improved single-core speedup to 6.6×.

Question 3. A worst-case input for `sqrt` is an input that results in significant *execution divergence*. One such input sets input array elements to 2.999 if their index modulo `programCount` (gang size) is 0, and 1.0 otherwise. Given this input, seven of the eight instances in a gang complete their work immediately, then do no useful work (their execution is masked out) while the single long running instance 0 completes. Note that the change in input values *only effects only the SIMD speedup* of the parallel `sqrt` implementation. It in fact drives the performance of the single-core implementation down below the sequential implementation (.92×) due to ISPC needed to emit extra instructions to manage divergent control! **However, the modified input values above does not reduce the near perfect multi-core speedup of the code as the amount of work per processing core remains well balanced.**

Some students provided an input of all 1's as a "bad case" input. This input did not create divergence, but made the problem increasingly bandwidth bound, which is the reason for the low speedup over around 1.8×. (Creating divergent execution resulted in lower single core performance than making the program bandwidth bound.)

Program 5

saxpy exhibits very low arithmetic intensity (high communication to computation ratio) because it performs only a single multiply-add operation for every 16 bytes read/written to memory. **saxpy** is not significantly more computationally intensive than **memcpy**. **As a result, performance does not significantly improve when adding SIMD execution or multi-core execution because the program is bandwidth bound.** More precisely, essentially no speedup from SIMD execution is observed, and only a very small speedup from multi-core execution (if any at all) is observed, suggesting that memory bandwidth is saturated by just a few cores.

Note that the 16 bytes of bandwidth per element processed (as opposed to 12) is correct. For each operation, the CPU needs to load the input arguments (2 floats), and store the output (1 float is read because the cache line being written to is read from memory, and then it is written when the cache line is evicted.)

There is no data reuse in `saxpy` (each element of the input and output arrays is touched exactly once). As a result, improving the caching behavior of `saxpy` is not an available optimization. There is no way to improve its arithmetic intensity via clever blocking strategies or loop reordering as some students suggested.

Note that the observed bandwidth (using ISPC with tasks) of approximately 25 GB/sec is around half the theoretical peak of the machine (38.4 GB/sec). Is it possible to write a program that does notably better? You might be able to, but you have to hack and use 16-byte aligned SSE/AVX loads/stores and special non-temporal store instructions that improves effective bandwidth by better use of the memory bus. (Not topics covered in this class.)

Program 6

kmeans is a serial implementation of the popular K-Means algorithm. There are 3 main steps to the algoithm:

1. Computing the cluster assignments: we first need to figure out which data points belong to which clusters, where each data point is assigned to the closest cluster centroid (based on Euclidean distance).
2. Computing the cluster centroids: once we have the assignments, we can recompute the cluster centroids by taking a mean over all data points assigned to a particular cluster.
3. Computing the cost: for each cluster we then compute the sum of the distances between the centroid and all datapoints and use this as the cost.

These steps are repeated until the change in cost for each cluster between successive iterations is less than some predefined threshold ϵ, i.e. $|prevCost[i] - currCost[i]| < \epsilon, \quad i = 0, ..., K-1$.

For this program, students are tasked with figuring out where the "hotspot" in the code is, and then how to improve it. To do this, students should insert timing code and figure out that the function for computing the cluster assignments is the dominant component of the workload (in terms of runtime) and then try to optimize this function. There are a few constraints on the problem, notably their changes cannot affect the functionality of the algorithm (it still must produce a correct result) and they may only parallelize one of the functions in the code (`dist`, `computeAssignments`, `computeCentroids`, or `computeCost`).

It's important to note that there are various axes of parallelism that one can exploit in this problem, and the performance effects of parallelizing over these axes is largely dependent on the problem parameters M, N, and K which are the number of data points, the dimensionality of the data points, and the number of clusters. In particular, it's helpful to remember that $K << N << M$. The dataset we give to students has $M = 10^6, N = 10^2, K = 3$.

Since `computeAssignments` takes up the majority of the algorithm's runtime, the solution is to parallelize this function. Students will need to identify they should paralellize over the M loop in the function. Upon closer inspection, they should also find that they can reorder the M and K loops which will lead to a better data access pattern and cut down on some other overheads. This also just leads to a more "parallel-friendly" formulation of the problem, as some students stumbled upon this optimization by thinking about how to break up the problem into chunks of independent work and/or code up the parallelization over the M loop cleanly.

With these two observations, it should be straightforward to statically assign loop iterations over the M dimension to threads. If students don't switch the loop order, they will not achieve the same performance speedup as the solution (1.8x vs 2.4x) due to additional overheads such as potentially having to repeatedly spawn and wait on threads, as well as a suboptimal data access pattern and poor cache behavior (since the starting code will loop over the large M array K times).

Threshold is set at 2.1x to give some headroom.

An alternative solution that was found was to "cache" and reuse the distance computations in `computeAssignments` and `computeCosts`. Implementations of this solution generally require storing the K distances for each of the M data points (instead of recomputing them). This should give >3x speedup.

Note: since this is a very open ended question, there are many other possible solutions. We are mainly looking for students to exercise good debugging skills and justify their approaches and design.