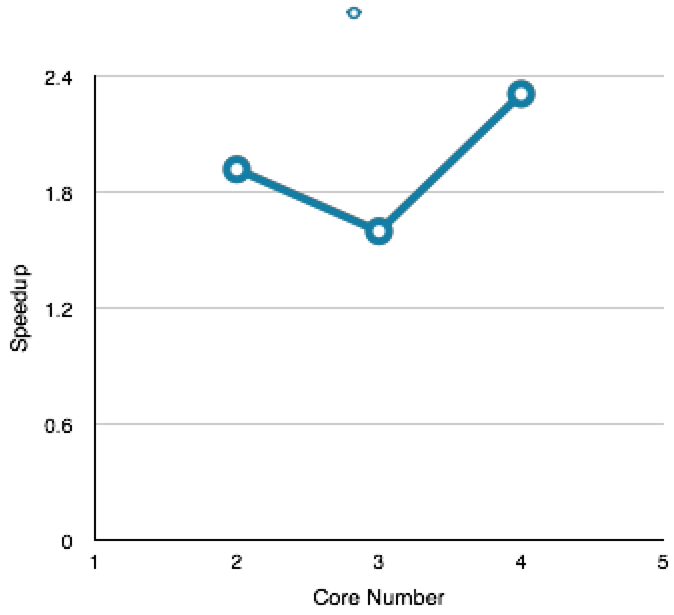
15618 Assignment 1
Name: Xiaoxiang Wu
Andrew ID: xiaoxiaw

Program 1

Core used	2	3	4
Speedup	1.92x	1.60x	2.31x



The speedup is not linear to the number of cores used. Because when we divide the Mandelbrot set into more than two parts, the workloads are distinct between different threads. And the overall performance is limited by the slowest thread.

(1)Divide into two parts:

Thread 0 takes 0.137956 seconds.
Thread 1 takes 0.138843 seconds.

Since the Mandelbrot set is symmetric between the upper and lower parts, each thread only need to process half of the original workload. So the speedup is about 2x.

(2) Divide into three parts:

Thread 0 takes 0.059064 seconds.
Thread 2 takes 0.060497 seconds.
Thread 1 takes 0.165328 seconds.

Thread 1 is responsible for the middle part of the set, which is much more expensive to compute. Since thread 1 has more workload than any one of the thread when the set is divided into two parts, it takes more time to finish. In addition, the overall performance of the program is limited by the slowest thread, so the speedup is less than when the set is divided into two parts.

(3)Divide into four parts:

Thread 0 takes 0.031285 seconds.
Thread 3 takes 0.031663 seconds.
Thread 1 takes 0.114764 seconds.
Thread 2 takes 0.115364 seconds.

Thread 1 and 2 are responsible for the middle parts of the set, which is more computationally expensive. However, compared to the case that the set is only divided into two parts, they still have less work to do. Although the speedup is still limited by the slowest thread 1 and 2, the speedup is slightly higher than using only two cores.

My algorithm to gain more speedup is to let the threads process rows by turn. For example, supposed we have three threads, thread 0 computes row 1, 4, 7, …, 3k + 1, thread 1 computes row 2, 5, 8, ... , 3k + 2 and thread 2 computes row 3, 6, 9, …, 3k + 3. The reason is that rows close to each other have similar computational costs so that we can divide the workloads more evenly. The final speedup is 3.49x on view 1 and 3.48x on view 2.

Program 2

Vector width	2	4	8	16
Vector utilization	85.2%	80.4%	77.9%	76.7%
Instructions number	167727	97075	52877	27592

The vector utilization decreases as the vector width increases.
Because if the vector has longer length, the lane that are empty may still be involved into computation for several times, and thus lower vector utilization.
For example, only when all elements' exponents become zero will the while loop terminate. So if there are more elements in the vector, those elements whose exponent quickly become zero still need to occupy the lane. For instance, consider 4 elements whose exponent are [1, 1, 2, 2]. If the vector length is only two, the two vectors would look like ** -> __ and ** -> ** -> __. However, if the vector length is four, the vector should look like ****-> __** -> ____. In this case, we have two more lanes that are not utilized.

3. EXTRA CREDIT
I use hadd to combine adjacent elements. After this, adjacent values will have same value. For example, {1, 2, 3, 4} -> {3, 3, 7, 7}. Then I use interleave to move all even-index elements

to the front of the array. However, I do this in place and do not use an extra vector to store the result, so the back half of the vector may not be what is expected, but we only care about the front half. For example, after interleaving {3, 3, 7, 7} -> {3, 7, 7, 7}, which should be {3, 7, 3, 7}. However, here we do not care about the back half. In the next iteration, {3, 7, 7, 7} -> {10, 10, 14, 14}. Now, the first value 10 is the result. Given a vector width, we only need to do log(vector width) iterations. And in the last iteration, we do not need to interleave, since the first element is already the result.

Program 3

Part 1

Since we are use 4-wide SSE vector, the maximum speedup should be 4x. However, the actual speedup is 3.73x in view 1 and 3.22x in view 2. The reason is that region with very low brightness only need very few iterations to compute. This will cause it to access memory too often. However, the memory bandwidth is not fast enough to feed data to the functional units. So some of the functional units remain idle when the computational cost is low. This can also be demonstrated in view 2, whose overall computational cost is lower than view 1, and thus gain a smaller speedup.

Part 2

1. When running with parameter --tasks, the speedup is 7.19x, which is a 1.92x speedup over the version of mandelbrot_ispc without tasks. Since we use two tasks to compute, this is a reasonable speedup we can gain from using tasks.

2. I create 800 tasks totally, one task for one row in the mandelbrot set, which achieves 15.95x speedup on view 1 and 13.02x speedup on view 2. It works best because although each task may take significant different time to finish. The core which finish a light workload task can continue to process the next task, without waiting for other cores. Although different core may process different number of tasks, they have almost the same execute time except that the last task may run for a long time, which is a rare case. So the overall performance is the best. Actually, by creating 400 tasks, we can get similar speedup. This shows that creating more tasks give only little overhead compared to a lot of threads.

3. EXTRA CREDIT:
The pthread abstraction is to give programmers interfaces to explicitly create threads and to assign which parts of the data each thread should deal with manually. However, the ISPC task abstraction defines a gang of program instances that are independent and can run parallelly without caring about the details about how the data are assign to each threads.

Talking about the implementation, when we launch 10,000 ISPC tasks, they will be enqueued to be run asynchronously. This can be implemented in the manner like thread pool, where we are able to reuse threads to execute the tasks. However, when we launch 10,000 pthreads, it will create 10,000 threads at once, which will impose a great burden to schedule them and suffer from frequent context switch. Also, creating and destroying such a large quantity of threads have very large overhead. In addition, the threads created by pthreads are not guaranteed to be independent, which may also lead to synchronization overhead.

4. Foreach and launch defines different level of parallelism. Foreach mainly make use of SIMD on one core, which tells the CPU that all these data can be computed independently. Launch will spawn gangs of tasks, which will make use of multiple cores. All these tasks can run independently on different cores. This gives the programmer more flexibility to decide if they want to distribute the work to multiple cores. For example, In some programs, it may be enough to just run on one core and more introduce some overheads when running on several cores.

Program 4

1.
2.65x speedup from ISPC, and 15.28x speedup from task ISPC.
The speedup from ISPC without task is due to SIMD parallelization, which is 2.65x. The speedup from task ISPC is due to multi-core parallelization, which should roughly be (15.28/2.65)x.

2.
When all values are very close to 3, such as 2.999f, the program can achieve maximum speedup over the sequential version, which is 3.79x from ISPC and 23.49x from task ISPC. It both improve SIMD and multi-core speedup. Originally, because the input involves relatively less iteration to converge, it needs to access memory frequently and lots of functional units are idle due to lack of memory bandwidth. However, we can see from the convergence of sqrt graph that it takes much more iterations to converge when the input values are close to 3. So functional units are involved into more computation and access the memory less frequently, which leverages the bottleneck of memory bandwidth. In this case, using SIMD and more cores can help to improve the speedup.

3.
The input contains 2.999f every four elements and the remaining elements are all 1. For example, the input looks like {2.999f, 1.f, 1.f, 1.f, 2.999f, 1.f, 1.f, 1.f, 2.999f, 1.f, 1.f, 1.f, ….}.

The ISPC implementation achieves 0.91x speedup. The reason for the loss of efficiency is that value 1 takes only several iterations to converge, while 2.999f takes much more iterations to converge, so the three lanes that contain 1.f are masked most of time while the lane containing 2.999f still need to run many iterations. In this process, lots of functional units fed by 1.f are actually not doing useful work, so the core is only about ¼ of its peak performance.

4. EXTRA CREDIT
I wrote the AVX program according to Intel Intrinsics Guide:
https://software.intel.com/sites/landingpage/IntrinsicsGuide/
One tricky part is that the AVX intrinsics the memory address must be aligned on a 32-byte boundary. I solved it by google and found memalign is what we want:
http://linux.die.net/man/3/memalign
Another tricky part is how to get the terminating condition of the while loop. I combine the _mm256_cmp_ps with _mm256_testz_ps.
cmpRes = _mm256_cmp_ps(error, threshold, _CMP_GT_OQ) will only give all zeros when all errors are not greater than thresholds.
_mm256_testz_ps(cmpRes, vecAllOne) will return 1 only when all sign bits of vector cmpRes are zeros.
So, combining them, we get the condition that as long as there are one error in the vector that is greater than threshold, we will return 0.
This is the final result:

```
[sqrt serial]:          [1045.810] ms
[sqrt ispc]:            [395.315] ms
[sqrt task ispc]:       [61.290] ms
[sqrt AVX]:     [251.010] ms
                                (2.65x speedup from ISPC)
                                (17.06x speedup from task ISPC)
                                (4.17x speedup from AVX)
```

Program 5

1.I saw a 0.99x speedup from use of tasks. This shows that spawning more tasks does not actually help improve the speed. The reason is that the computations are so simple that it has to access new elements from the memory too frequently. It is limited by memory bandwidth. However, it is very weird that the result shows that the memory bandwidth of this program is just about 10 GB/s, while the official specification shows that the maximum memory bandwidth is 25.6GB/s, which can be found at:
http://ark.intel.com/products/80814/Intel-Core-i7-4785T-Processor-8M-Cache-up-to-3_20-GHz
This shows that actually we are not make full use of the memory bandwidth. There should be somewhere that we can improve the program's performance. One possible problem is that we

are not actually reuse any of the elements, however, we still load them into cache, which will introduce more latency. In addition, we are actually have redundant access to the memory when writing the result. Since we have to first load the result into the cache, and then when the cache is full, it is write back to the memory, which is unnecessary. However, this should not lead to only 10GB/s memory bandwidth, while the idea one is 25.6 GB/s. I still cannot figure out a way how to explain this.

2. EXTRA CREDIT
As described in the previous question, when saxpy write one element into result, it actually load the result into cache and then at some later time be written back to memory. So it actually has 2 accesses to the memory when writing. So plus the two load operations, it accesses the memory 4 times. So total bytes are 4 * N * sizeof(float).

3. EXTRA CREDIT
When I try to solve this problem, I find that since we have unnecessary access to memory when writing result, maybe there is a way to bypass the cache and write directly to memory. So I search “bypass memory cache” on Google, and found this article: http://lwn.net/Articles/255364/. It shows that we can use _mm_stream intrinsics to bypass the cache. And then I try to find if there is a corresponding AVX intrinsic on Intel Intrinsic Guide: https://software.intel.com/sites/landingpage/IntrinsicsGuide/ and found _mm256_stream_ps. I tried it and finally got 1.34x speedup, which is still not satisfactory. So I was wondering if creating more threads can help to improve performance, but finally got 1.32x speedup. The reason maybe that like the original program, it is still limited by the memory bandwidth. So adding more threads does not help at all. We may also introduce some overhead when using more threads. Followed are the result:

```
[saxpy ispc]:           [27.526] ms    [10.827] GB/s [1.453] GFLOPS
[saxpy task ispc]:      [28.224] ms    [10.559] GB/s [1.417] GFLOPS
                                (0.98x speedup from use of tasks)
[saxpy stream]:         [20.548] ms    [14.503] GB/s [1.947] GFLOPS
                                (1.34x speedup from use of stream)
[saxpy stream thread]:          [20.841] ms    [14.300] GB/s [1.919] GFLOPS
                                (1.32x speedup from use of stream threads)
```

But the memory bandwidth is still far from 25.6GB/s, which is the best possible value. However, I have not figured out why we cannot improve the memory bandwidth to this idea value so far.