CS 475 Parallel Computing
Instructor: Matthew Meyn
Student: **Khuong Luu** (**luukh**)

Project 5

1. **What machine did you run the programs on**

- Processor: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
- Memory: 16GB, 2400Mhz
- Operating System: Ubuntu 16.04
- Kernel: 4.13.0-37-generic
- **Number of Cores: 4**
- Cache line size: 64 bytes
- **Compiler: Intel(R) C++ Intel(R) 64 Compiler for applications running on Intel(R) 64, Version 18.0.2.199 Build 20180210**

2. Data Tables

2.1. Results with Non-Vectorized

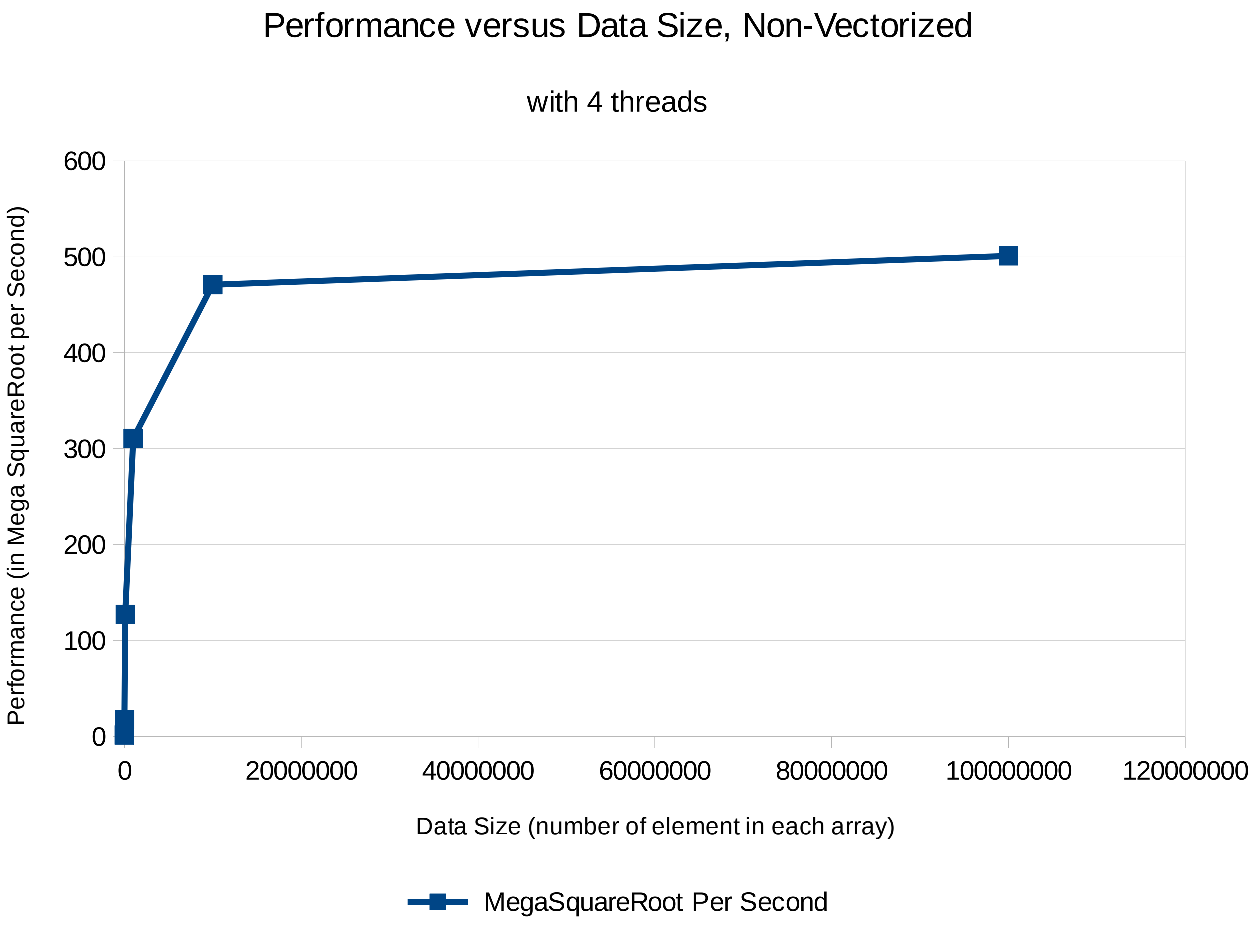
NUMT	ARRAYSIZE	MegaSquareRoot Per Second
4	1000	1.65
4	10000	17.89
4	100000	127.41
4	1000000	310.67
4	10000000	471.16
4	100000000	501.11
1	1000000	123.84
2	1000000	231.37
4	1000000	327.65

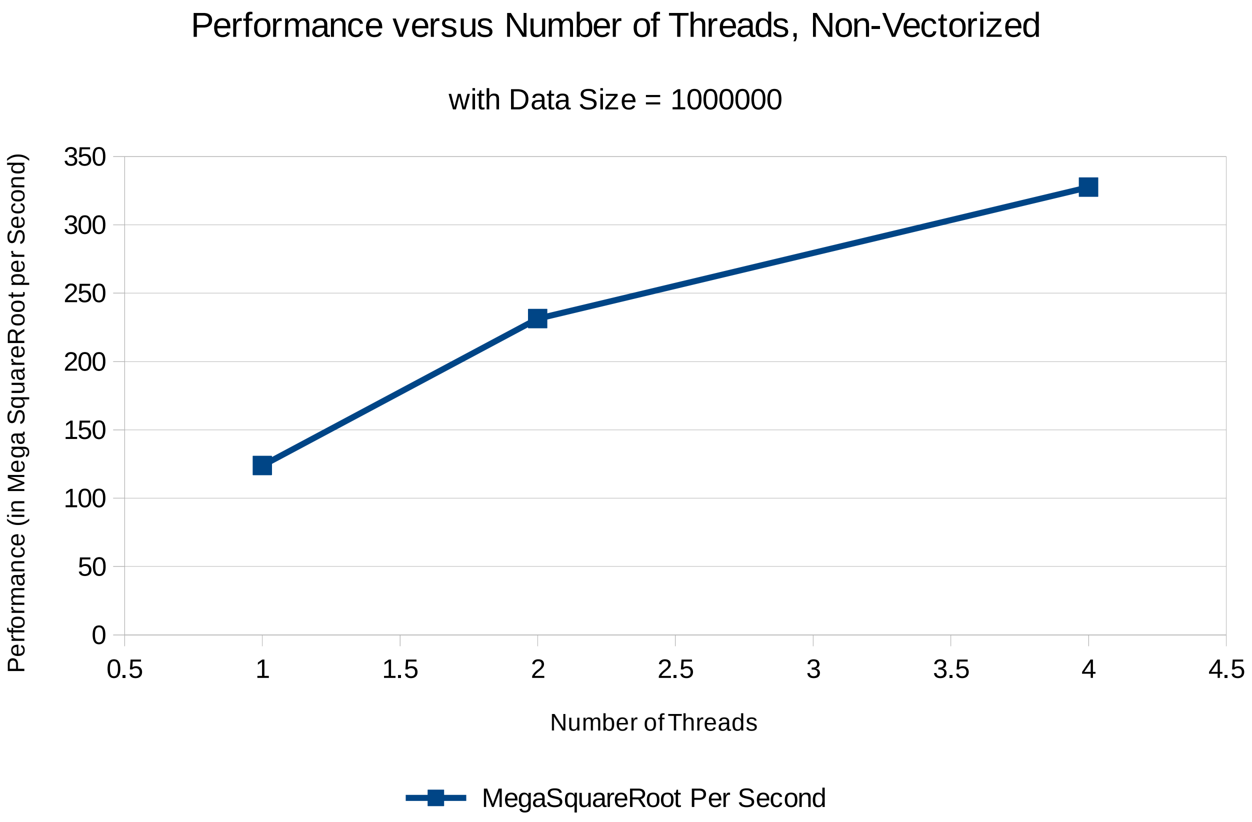
2.2. Results with Vectorized

NUMT	ARRAYSIZE	MegaSquareRoot Per Second
4	1000	1.7
4	10000	31.44
4	100000	103.72
4	1000000	524.68
4	10000000	892.86
4	100000000	973.04
1	1000000	227.38
2	1000000	379.95
4	1000000	520.06

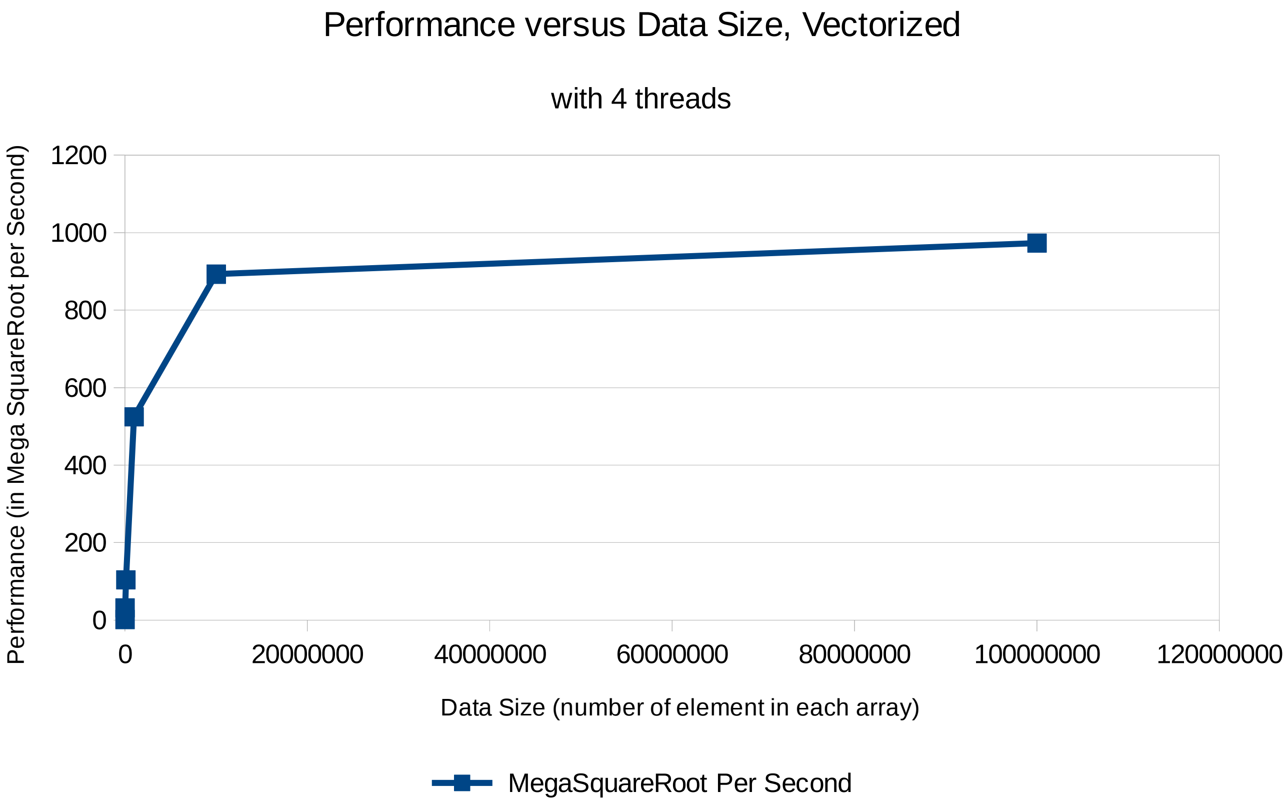
3. Graphs

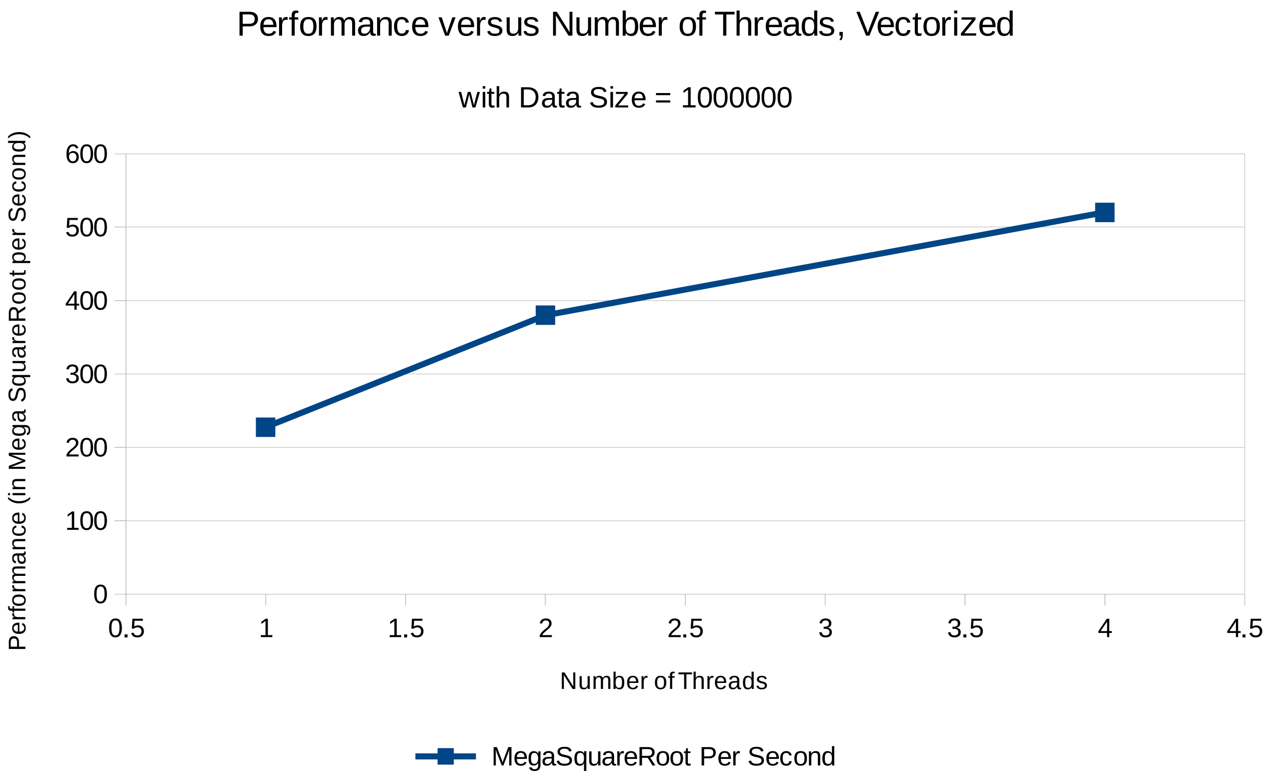
3.1. Non-Vectorized





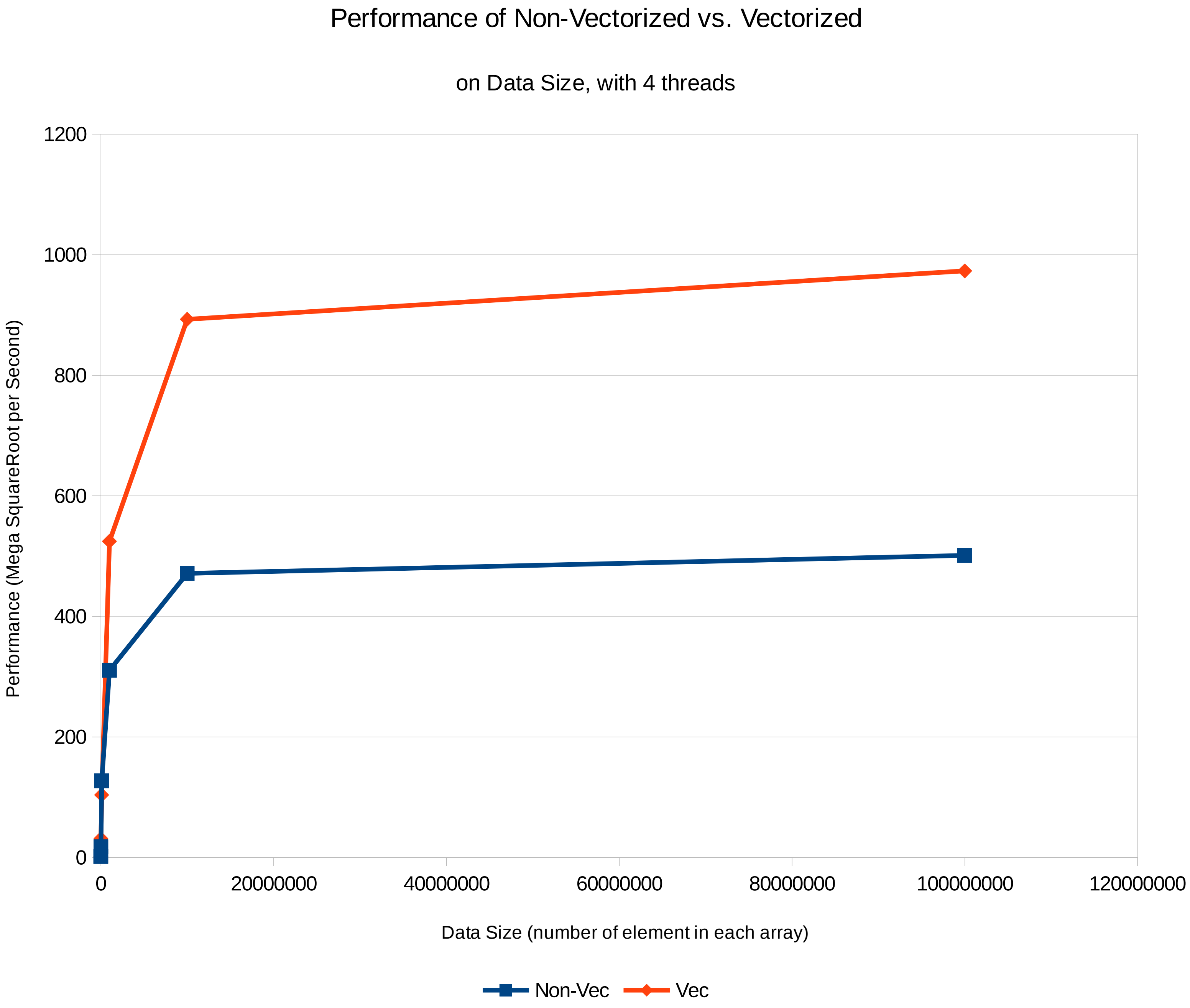
3.2. Vectorized

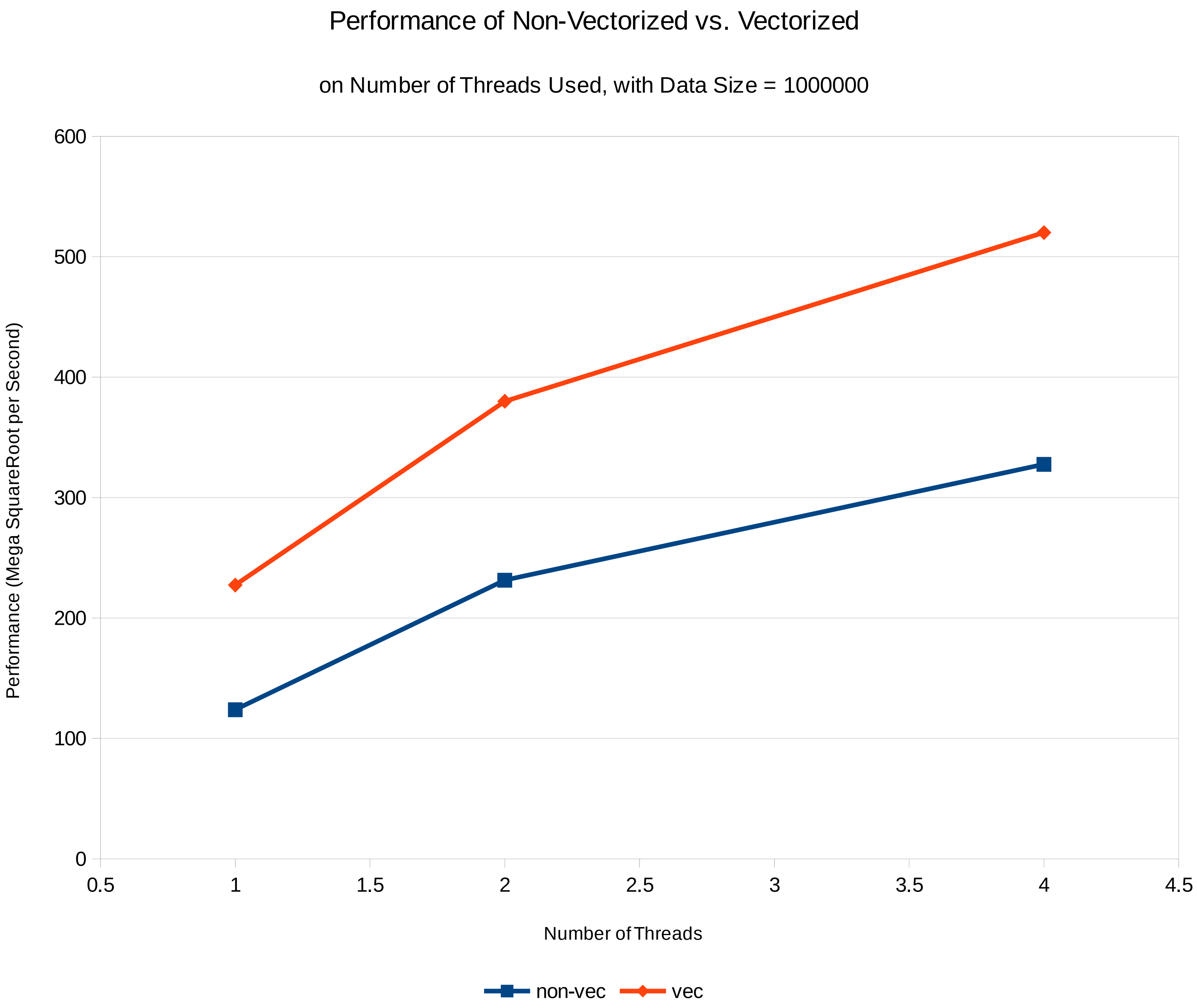




4. Patterns and Explanations

When I look back and forth between 2 graphs, one with non-vectorized and one with vectorized, **I honestly could not tell much notable difference between those 2 methods until I combine each of 2 graphs together** into the following graphs:





Beside the probably obvious pattern that **the vectorized method performs a lot better** than the non-vectorized method (to more accurate: around 1.8 times or 180% better), the most interesting and notable pattern I can now see after combining 2 graphs is that **the vectorized method <u>scale better</u> than the non-vectorized version on both data size and number of threads**.
That is, as we look at the trend of performance as I keep increasing the data size, the performance of vectorized method keep increasing and does not look like it's going to converge to soon while the performance of the non-vectorized method seems to start converging at ARRAY_SIZE = 1000000. For the increasing of number of threads, the curve of vectorized method, though already better than its counterpart in value, has the slope notably steeper than the slope of the curve of non-vectorized method.

Explanation:

The vectorized version perform better: This is because each thread in the vectorized program performs square root operation on multiple array element at the same time while each thread in the non-vectorized program performs the same square root operation but only on one array element at a time (iteratively). As a result, since each thread in vectorized version perform better, the overall performance of n-threads is better than the non-vectorized version.

The vectorized version scale better: This is because in vectorized program, we have another layer of parallel inside each threads (perform multiple operations at once instead of one operation at once) , the upper bound of speed up (compared to 1 threads) of vectorized program is higher than the non-vectorized program, and so the performance of the vectorized program converges later than the non-vectorized program. Take the above case as an example, while the non-vectorized program converges at around 500 Mega Square Root Per Seconds, the vectorized program, though not visualized, may converge at around 1.8 x 500 = 900 Mega Square Root Per Seconds (where 1.8 is how much times the vectorized program performs better than the non-vectorized program), and it converges at this value at a much bigger data size (array size) thus it’s the rationale for us to say the vectorized program scales better on data size.

5. What does that mean for the proper use of vectorized parallel computing?

Vectorized parallel computing work best with operation on memories that near each other on cache line. Thus, it is best to align data to the beginning of the cache line. In icpc, I use the -align flag to accomplish this. We can also accomplish this via the “aligned” clause in #pragma omp simd. More specific, my current Intel CPU has 64 bytes cache line and 32 bytes vector size, so the best alignment is 32 bytes, which is two vectors per one cache line.

Also, as each thread performs vectorized operation on <simdlength> element at once, where <simdlength> is the number of element the compiler can vectorize the operation to perform on. Thus, it’d be better to have each threads a large enough number of bytes to perform vectorized operations, at least 32 bytes in my computer.