1) (3 points) Suppose you are programming a processor with an add latency of 3 clock cycles and a multiply latency of 5 cycles. It is also given that this processor can complete one add and one multiply instruction every clock cycle, when instructions are fully pipelined. Consider the following loop:

```
for (i=0; i<m; i++) {

A[i] = B[i] * C[i] + D[i] + i;

}
```

1a) Assuming the program is executed as-is (i.e. no pipelining), that is the lower bound on execution time (in clock cycles) based on the math performed?

The code performed within the loop contains one multiply and two additions. In the case where instructions are not pipelines, this line of code would take 5+3*2 = 11 cycles. If we want to be more precise, we might want to include the incrementation of i. However, that might not be correct, as the incrementation of i is handled in parallel by the hardware and does not affect other operations very much. And so, the lower bound on execution time is (5+3*2)m = 11m.

1b) How can you exploit more instruction level parallelism in this program? What changes do you propose?

This program is an O(m) program with only one loop and so there are very little optimizations that can be done. Using SIMD operations is the most effective optimization method in this case. An SIMD operation allows us to perform multiple operations with only one command. SIMD operations are usually integrated with the hardware to enable efficient processing and exploit data level parallelism.

1c) Assuming you can pipeline the adds and multiplies, what would be the lower bound on execution time in clock cycles for the arithmetic?

Assuming the wind-up and wind-down times of the processor pipeline to be 4 cycles each, and considering that additions and multiplies each take one cycle when instructions are fully pipelined, we can infer that the lower bound on execution time is (1+1*2)*m + 4 + 4 = 3m + 8.

2) Tests performed on lab machine and ACI-I cluster. Lab machine specs: (Intel(R) Xeon(R) CPU E5-1620 v3 @ 3.50GHz, 15.5 GB RAM, 4 cores, 8 threads, Kernel Linux 2.6.32-696.18.7.el6.x86_64) .

Optimizations performed:

- Switched loop nest structure for maximum loop unrolling
- Simplified trigonometry formula: sin^2(x) - cos^2(x) = -(cos^2(x) - sin^2(x)) = - cos(2x)
- Replaced fmod with % as expression should return integer (as i_v contains i*i values and i is an integer)
- Precomputed -cos(2*d_val) outside the inner loop and stored in d_val
- No need to compute the size() of the vectors; declared i_N and I_R as global variables and use anywhere

I will go over each optimization.

The first optimization is maximum loop unrolling. Let's take a look at the main operations being performed inside the loop:
d_val = round(fmod(i_v[i],256)); // this should return an integer
v_mat[i][j] = v_s[i][j]*(sin(d_val)*sin(d_val)-cos(d_val)*cos(d_val));
We notice that v_mat and v_s cannot be affected by loop unrolling as they cannot be reused (their values change on each and every iteration of i and j). We notice that d_val only relies on index i. Instinctively, we can perform "classic" loop unrolling by adding v_mat[i][j+1], v_mat[i][j+2] lines and so on. This optimization would heavily impact running time and improve our elapsed time in a big way. However, after careful analysis, we notice that we want to use one d_val for one i (to cover all values of j for a specific i). That implies that we can invert the nest structure. The outer loop would increment i and the inner loop would increment j. We take d_val out of the inner loop. By doing that, we obtain the maximum loop-unrolling effect possible and achieve incredible timing.

The next optimization is fairly obvious if one has taken trigonometry. Trigonometric operations are known to be very heavy on the processing side and so we should always try to minimize them. I thought about using a lookup table but after careful consideration, I did not think it would bring any timing improvements. And so, I decided to simplify: sin^2(x) - cos^2(x) = -(cos^2(x) - sin^2(x)) =
- cos(2x). This optimization had a minimalistic impact on elapsed time, but it is an optimization nonetheless.

The next optimization is a small one that I thought would have a big impact but it did not. In the code, we can see round(fmod(i_v[i],256)) being used. Next to it, a comment specifies that the value produced from this operation should be an integer. This does make sense as i_v contains i*i values and i is an integer meaning the result will also be an integer. So why use fmod() when we don't have floats to operate on? Also, why use round() when we are operating on integers and are getting an integer remainder? And so I replaced all of these operations with a simple modulus (%), which should definitely be an optimization. After testing, there was a minimalistic improvement on the elapsed time, similar to the previous optimization.

The next optimization improves data reuse. Instead of computing unnecessary operations in the inner loop, we compute them only in the outer loop. This maximizes data reuse in our loop. Again, I also saw very small improvements with this optimization.

The last optimization is simple. There is no need to compute the size() of the vectors as we are the ones inputting their size initially. By making i_N as a global variable, I was able to substitute it with the size() functions. Negligible improvements are expected in this case and indeed they were.

These were the optimizations specific to myfunc() that I was able to perform. There may be more optimizations that can be done such as the usage of SIMD operations. However, coding SIMD operations is slightly complex and was not elaborated on in class so I chose not to perform it. I also performed the spatial blocking optimization. However, it actually produced a longer elapsed time, since I was unnecessarily splitting the workload. The “maximum loop unrolling” is maximizing data reuse and so spatial blocking brought no advantage.

<u>On lab machine:</u>

	N = 10, R = 1000	N = 100, R = 1000	N = 1000, R = 1000
Unoptimized	0.010705	0.512718	58.759934
Optimized	0.013933	0.121269	1.160238

<u>On ACI-I cluster:</u>

	N = 10, R = 1000	N = 100, R = 1000	N = 1000, R = 1000
Unoptimized	0.005855	0.599223	73.613107
Optimized	0.008629	0.121269	1.145862

<u>**3)**</u> For this exercise, I use elapsed time benchmarking similar to the last exercise. The task is to benchmark the matrix-vector multiply code before and after adding loop unrolling and spatial blocking. I chose R = 10000 so that the elapsed time is measurable and not too big. The matrix-vector multiply algorithm is simply two nested loops with each going to N. I initially perform tests without any optimization and get expected results. As N increases by a factor of 10, the total number of operations is actually increasing by a factor of 100.

I first perform loop unrolling. I attempt to test out how much unrolling maximizes performance. After testing with only one line all the way to “maximum unrolling”, I conclude that 5 lines net the best elapsed time. We see a sizeable improvement in elapsed time compared to the unoptimized version.

I now attempt to perform spatial blocking. However, something very unexpected happens. I first try to set the blocking size to 500, half of 1000. I notice terrible performance, almost double of what I had initially. Then I set blocking size to 1000, equal to N, and I still receive almost double elapsed time. This is very strange as when setting bs to 1000, we are essentially not doing spatial blocking. I then tried to set N to 1024 and bs to values ranging from 64 to 512 thinking that maybe the factor of 10 values are messing up with the caches. However, I still obtain the same results. And so, I can only conclude that even though I set blocking size equal to N, the operations being performed in the inner loop, mainly the multiply operations (for (int j = (k-1)*bs+1; j <= k*bs; j++)), are heavily impacting performance as they have to be performed N times. I tried taking these operations out so that they only perform in the k loop. I did not see any improvements.

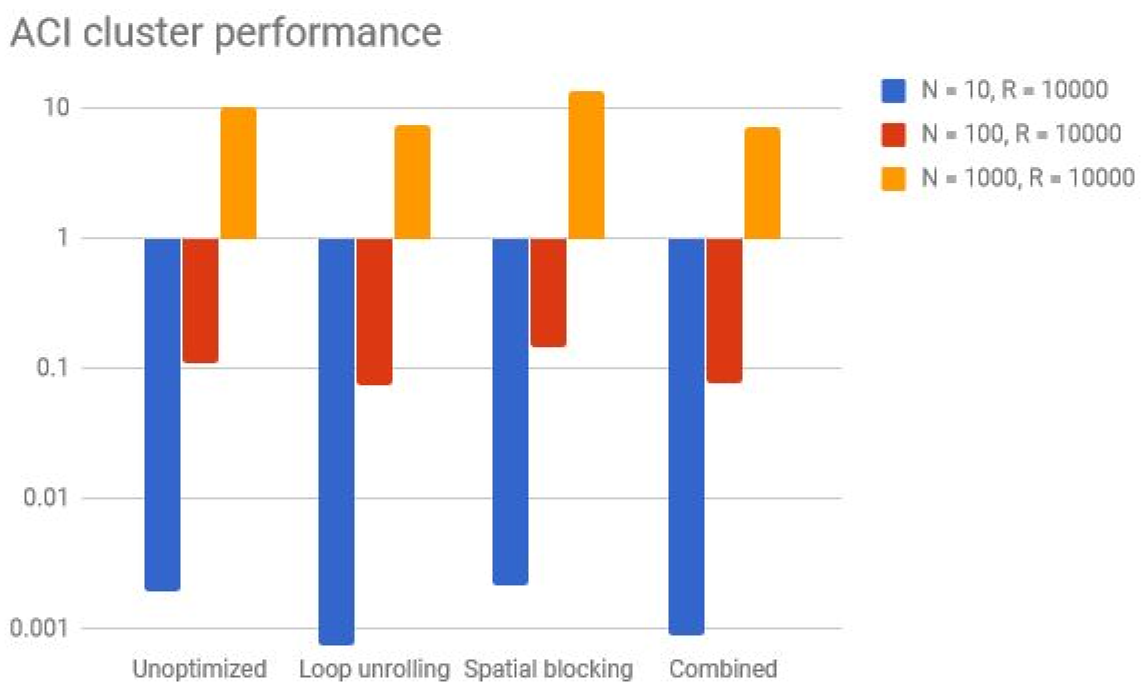
Finally, I attempt to combine both spatial blocking and loop unrolling (just curious) and I notice that the elapsed time is back to “normal”, near equal to the elapsed time when performing loop unrolling. I set blocking size equal to N as that provided me with the best elapsed time. This can only be explained as the expensive operations performed in the inner loop being amortized by the data reuse provided by loop unrolling. The strange thing however is that the elapsed time produced by this combination is always, on average, 0.2 seconds faster than the elapsed time when using only loop unrolling. This is not expected as we saw that spatial blocking actually hurt our elapsed time. The only explanation that I could make sense of is that when setting a blocking size of 1000, equal to N, we are encouraging or triggering better use of cache resources, thus the 0.2 seconds difference.

<u>On lab machine:</u>

	N = 10, R = 10000	N = 100, R = 10000	N = 1000, R = 10000
Unoptimized	0.002704	0.082006	6.616023
Loop unrolling	0.001667	0.058764	5.724850
Spatial blocking	0.003114	0.121724	11.301185
Combined	0.002233	0.064003	5.510949

On ACI-I cluster:

	N = 10, R = 10000	N = 100, R = 10000	N = 1000, R = 10000
Unoptimized	0.001968	0.109261	10.253451
Loop unrolling	0.000737	0.074252	7.454420
Spatial blocking	0.002141	0.146685	13.580124
Combined	0.000902	0.077280	7.161273



4) The first optimization that comes to mind is loop unrolling. After implementation and testing, I chose to have 10 lines for loop unrolling as they provided the most optimal elapsed time. I tried finding other optimizations in terms of unnecessary work done and tried to switch nest structure but could not come up with a better elapsed time. And so, after multiple runs, the fastest elapsed time was 7.447126 on the lab machine and 9.058453 lowest when running on the aci-i cluster with the compile command: “g++ -g -o Project2-4 Project2-4.cpp -O3”.