Kyle Salitrik
CMPSC 450
Homework 2 Report

CPU Performance Analysis

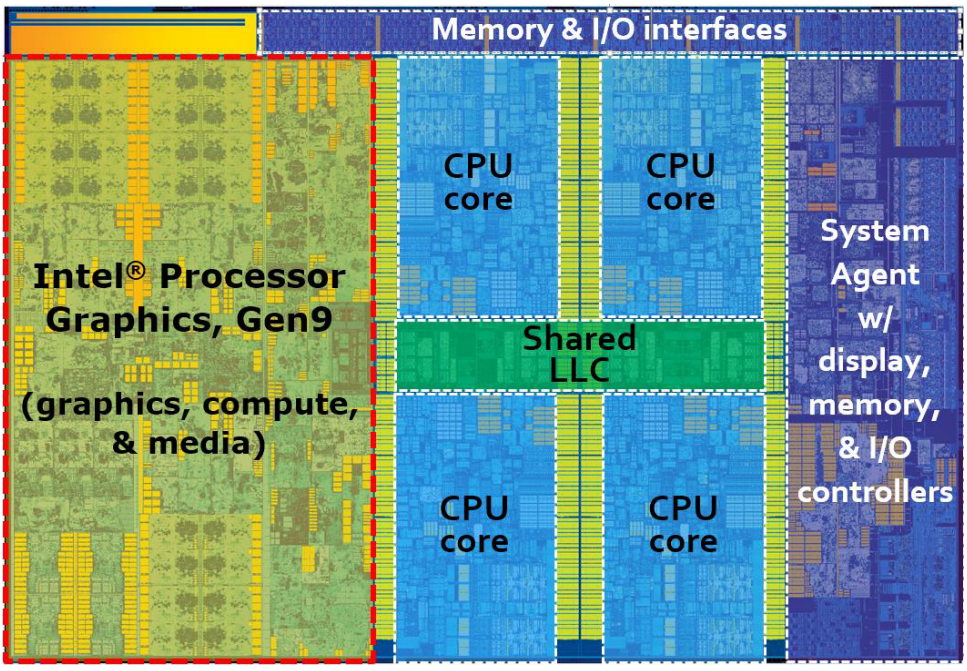
After the corrected code was released, I re-compiled and ran the Vector Triad Benchmark (VTB). The computer used has 16GB of RAM, an Intel i7-6700K processor (4.0 GHz standard clock spee, 4.2 GHz boost speed), and was running Ubuntu 16.04LTS. All compilations were performed with GCC.

The following table includes information gathered from the Wikipedia article on FLOPs and Intel's site. This figure displays the configuration of the 6th Generation

Table 1: i7-6700k Specifications

	i7-6700k	
Memory Bandwidth	34.1	GB/s
FLOPS	16	per cycle
Normal CPU Speed	4	GHz normal
Boost CPU Speed	4.2	GHz boost
Normal CPU FLOPS	64	GFLOPS
Boost CPU FLOPS	67.2	GFLOPS
Normal CPU FLOPS per Core	16	GFLOPS
Boost CPU FLOPS per Core	16.8	GFLOPS
Memory Bandwidth Per Core	34.1	GB/s

(Skylake) Intel Processors. The L3 cache is 8MB, shared across all four cores. In the above table,the memory bandwidth per core was unavailable, so various calculations were made with assumptions where the memory bandwidth is universal across all cores and if it would be split across cores.



The first table below includes the results for the VTB without any optimization flags passed to the compiler.

benchmark_default				
#	R	N	Time	MFLOPS
1	100000	10	0.002742	729.38075
2	58823	17	0.002706	739.077754
3	32258	31	0.002791	716.544907
4	17857	56	0.002833	705.987283
5	10000	100	0.002633	759.63126
6	5649	177	0.00264	757.476984
7	3164	316	0.002627	761.151793
8	1779	562	0.002568	778.656903
9	1000	1000	0.002598	769.879589
10	562	1778	0.002944	678.830507
11	316	3162	0.002667	749.314322
12	177	5623	0.002654	749.994455
13	100	10000	0.002614	765.104706
14	100	17782	0.004586	775.493774
15	100	31622	0.008212	770.13286
16	100	56234	0.014807	759.560393
17	100	100000	0.027073	738.739443
18	100	177827	0.047789	744.219493
19	100	316227	0.084926	744.712832
20	100	562341	0.14969	751.340012
21	100	1000000	0.26883	743.965078
22	100	1778279	0.47404	750.265329
23	100	3162277	0.873647	723.925588
24	100	5623413	1.537545	731.479485

The next table shows the results when optimization flags are passed (-O1, -O2, -03) and are marked as such.

benchmark_O1					benchmark_O2					benchmark_O3				
#	R	N	Time	MFLOPS	#	R	N	Time	MFLOPS	#	R	N	Time	MFLOPS
1	100000	10	0.002263	883.755584	1	100000	10	0.00225	888.90622	1	100000	10	0.001126	1776.118569
2	58823	17	0.002252	888.145315	2	58823	17	0.002183	916.178736	2	58823	17	0.001201	1665.052105
3	32258	31	0.002254	887.306032	3	32258	31	0.002231	896.408551	3	32258	31	0.001171	1708.122831
4	17857	56	0.002264	883.376252	4	17857	56	0.002291	872.987917	4	17857	56	0.000372	5377.269802
5	10000	100	0.002476	807.761964	5	10000	100	0.002217	902.09786	5	10000	100	0.00032	6250.825633
6	5649	177	0.002431	822.630703	6	5649	177	0.00263	760.361041	6	5649	177	0.000289	6920.414725
7	3164	316	0.002253	887.527154	7	3164	316	0.00219	913.133544	7	3164	316	0.000275	7274.181791
8	1779	562	0.002255	886.659636	8	1779	562	0.002172	920.627168	8	1779	562	0.000266	7515.155467
9	1000	1000	0.002253	887.683386	9	1000	1000	0.002189	913.69219	9	1000	1000	0.000264	7570.945848
10	562	1778	0.002248	889.075	10	562	1778	0.002174	919.30238	10	562	1778	0.000395	5058.659688
11	316	3162	0.002284	874.930063	11	316	3162	0.002228	896.93205	11	316	3162	0.000375	5331.952929
12	177	5623	0.002404	828.021251	12	177	5623	0.002245	886.581531	12	177	5623	0.000489	4070.667125
13	100	10000	0.002423	825.487896	13	100	10000	0.002623	762.462098	13	100	10000	0.000422	4739.326554
14	100	17782	0.004002	888.63474	14	100	17782	0.004005	887.947065	14	100	17782	0.00095	3743.192659
15	100	31622	0.007088	892.275429	15	100	31622	0.007119	888.390643	15	100	31622	0.001876	3371.435717
16	100	56234	0.012512	898.883329	16	100	56234	0.012674	887.401675	16	100	56234	0.003259	3451.057007
17	100	100000	0.022871	874.469185	17	100	100000	0.02281	876.809098	17	100	100000	0.00556	3597.173242
18	100	177827	0.039851	892.460526	18	100	177827	0.040474	878.723489	18	100	177827	0.01036	3432.952834
19	100	316227	0.074626	847.498544	19	100	316227	0.074184	852.545659	19	100	316227	0.027197	2325.444533
20	100	562341	0.129867	866.025499	20	100	562341	0.130838	859.599436	20	100	562341	0.064865	1733.878136
21	100	1000000	0.231697	863.196022	21	100	1000000	0.232163	861.46389	21	100	1000000	0.132168	1513.224967
22	100	1778279	0.398679	892.085572	22	100	1778279	0.402249	884.168049	22	100	1778279	0.245028	1451.49033
23	100	3162277	0.752726	840.21986	23	100	3162277	0.744637	849.347252	23	100	3162277	0.505324	1251.583624
24	100	5623413	1.298858	865.901171	24	100	5623413	1.307556	860.141114	24	100	5623413	0.847447	1327.142238

The boost to performance with the optimization flags is apparent, however telling the compiler to use SIMD commands (-mavx flag) produces significant increases in throughput as displayed below.

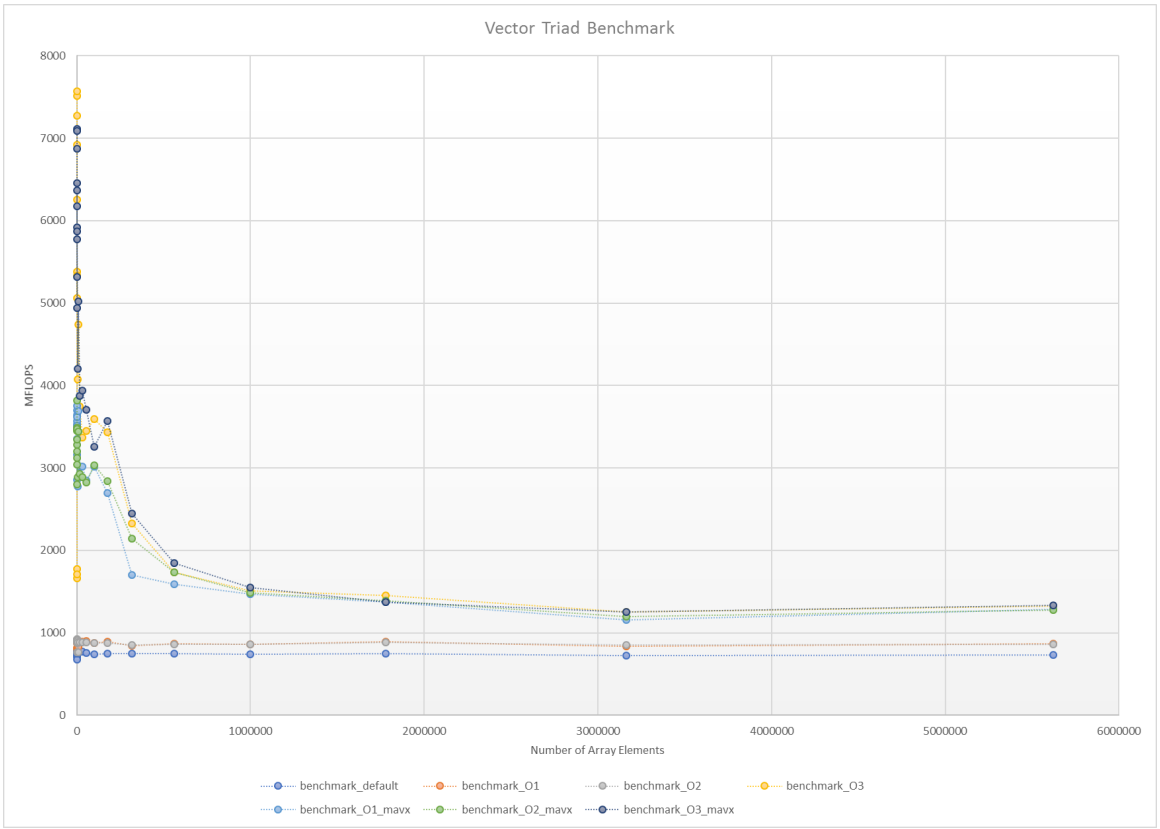
benchmark_O1_mavx					benchmark_O2_mavx					benchmark_O3_mavx				
#	R	N	Time	MFLOPS	#	R	N	Time	MFLOPS	#	R	N	Time	MFLOPS
1	100000	10	0.000549	3644.052129	1	100000	10	0.000524	3816.473157	1	100000	10	0.000405	4937.379635
2	58823	17	0.000567	3527.557823	2	58823	17	0.00058	3447.814428	2	58823	17	0.000338	5919.924137
3	32258	31	0.0007	2857.149599	3	32258	31	0.000714	2800.865183	3	32258	31	0.000314	6369.469417
4	17857	56	0.000632	3164.293056	4	17857	56	0.000657	3043.737624	4	17857	56	0.000324	6172.583437
5	10000	100	0.000596	3355.4432	5	10000	100	0.000641	3120.761905	5	10000	100	0.000341	5866.159441
6	5649	177	0.000572	3496.266214	6	5649	177	0.000609	3284.080911	6	5649	177	0.00031	6451.955882
7	3164	316	0.00056	3570.511539	7	3164	316	0.000598	3344.151358	7	3164	316	0.000291	6874.698037
8	1779	562	0.000553	3615.048923	8	1779	562	0.000624	3204.781621	8	1779	562	0.000281	7113.582274
9	1000	1000	0.000569	3515.761945	9	1000	1000	0.000578	3460.646865	9	1000	1000	0.000282	7090.961961
10	562	1778	0.000541	3694.226136	10	562	1778	0.000573	3488.222681	10	562	1778	0.000376	5315.281613
11	316	3162	0.000532	3756.983418	11	316	3162	0.000575	3475.053899	11	316	3162	0.000346	5776.588563
12	177	5623	0.000716	2779.273726	12	177	5623	0.00069	2884.913018	12	177	5623	0.000474	4199.667139
13	100	10000	0.000542	3690.544655	13	100	10000	0.000581	3442.186295	13	100	10000	0.000398	5023.118563
14	100	17782	0.001178	3018.948137	14	100	17782	0.001213	2932.302486	14	100	17782	0.000919	3870.426244
15	100	31622	0.002097	3015.740816	15	100	31622	0.00219	2887.704792	15	100	31622	0.001605	3940.352973
16	100	56234	0.003951	2846.689894	16	100	56234	0.00398	2825.884995	16	100	56234	0.003033	3708.238207
17	100	100000	0.006635	3014.340436	17	100	100000	0.006594	3032.977077	17	100	100000	0.006142	3256.320795
18	100	177827	0.013187	2696.969852	18	100	177827	0.012529	2838.615811	18	100	177827	0.009961	3570.504308
19	100	316227	0.037161	1701.92432	19	100	316227	0.029477	2145.593353	19	100	316227	0.025831	2448.431686
20	100	562341	0.070671	1591.431708	20	100	562341	0.064961	1731.319949	20	100	562341	0.060887	1847.167027
21	100	1000000	0.136387	1466.416804	21	100	1000000	0.134332	1488.846529	21	100	1000000	0.129048	1549.812477
22	100	1778279	0.258421	1376.265384	22	100	1778279	0.256548	1386.313283	22	100	1778279	0.258291	1376.95774
23	100	3162277	0.545964	1158.419597	23	100	3162277	0.527888	1198.086742	23	100	3162277	0.504673	1253.198397
24	100	5623413	0.8748	1285.645453	24	100	5623413	0.880554	1277.244037	24	100	5623413	0.842627	1334.73356

This next table shows the estimated performances of the code using the Machine and Code Balance equations provided. The first set of performance results assumed that the full memory bandwidth was available to all cores and the second assumed that the memory balance was split between pairs of cores. The graphs following this table shed more light on the performance of the benchmark.

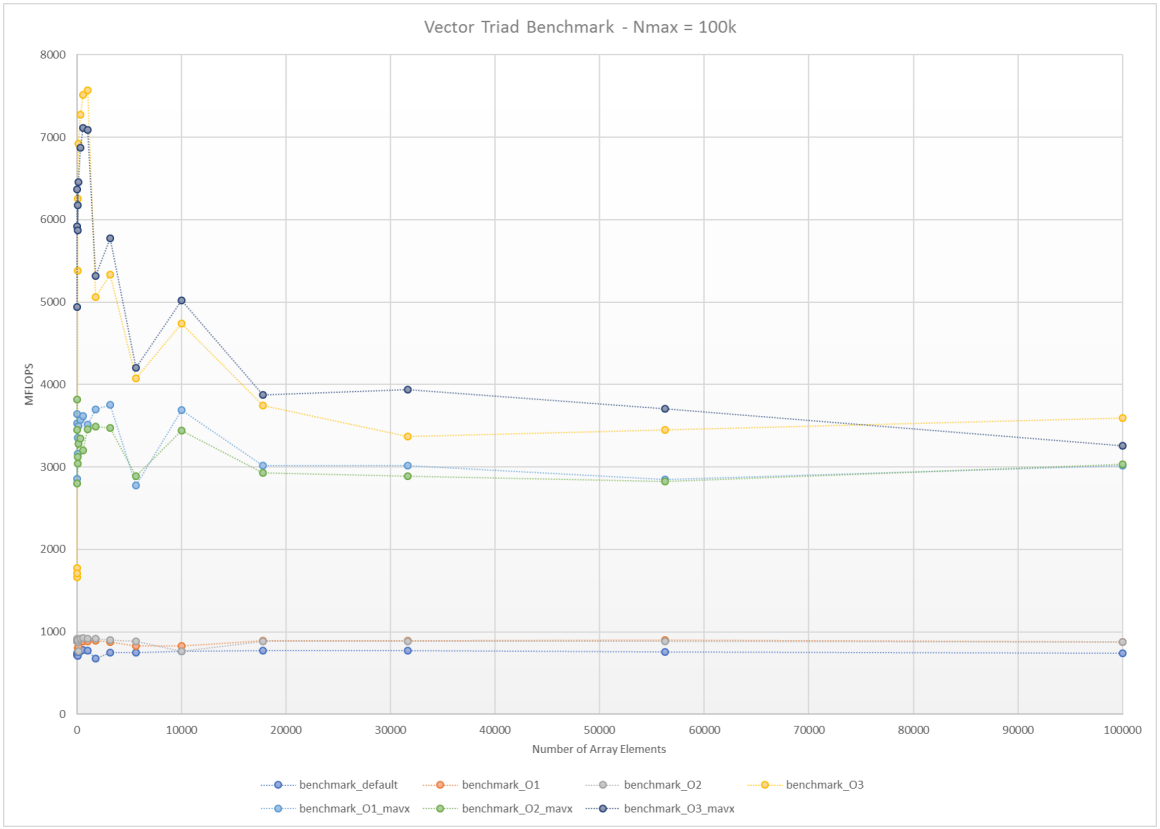
Table 2: CPU Performance Calculations

Memory Bandwidth	4.2625	Gwords/sec
Peak Performance	16	Gflops/sec
Machine Balance	0.26640625	Words/Flop
Data Traffic	4	Words
FLOPS	2	FLOPS
Code Balance	2	Words/Flop
L	0.133203125	
p-l*Pmax	2.13125	GFLOPS/Sec
P-bmax/bc	2.13125	GFLOPS/Sec
p-l*Pmax (bandwidth/2)	1.06563	GFLOPS/Sec
P-bmax/bc (bandwidth/2)	1.06563	GFLOPS/Sec

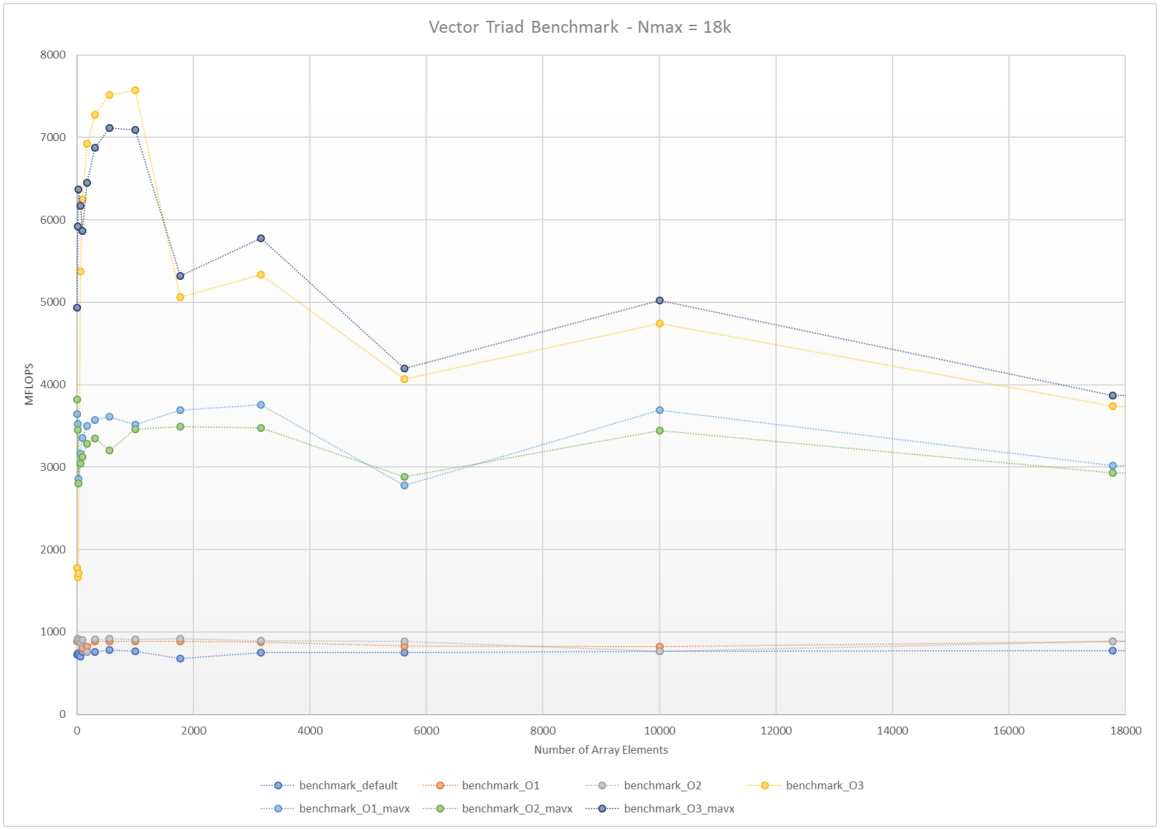
This first graph displays the full range of values of N, with the number of MFLOPS stabilizing between 1200-1300, or 1.2-1.3 GFLOPS, which is somewhere in between the estimated performance. Due to this and the advertisement of the Skylake having a "smart allocated cache" it is possible that the cache bandwidth can vary for each core based on the needed throughput. If the assumption that each core received 1/4 of the total bandwidth, the estimate would show the output being close to 500MFLOPS, so that must not be the case.


Vector Triad Benchmark
MFLOPS
8000
7000
6000
5000
4000
3000
2000
1000
0
0
1000000
2000000
3000000
4000000
5000000
6000000
Number of Array Elements
benchmark_default
benchmark_O1
benchmark_O2
benchmark_O3
benchmark_O1_mavx
benchmark_O2_mavx
benchmark_O3_mavx

This second graph is the same values, trimmed down to N=100k in order to showcase what I assume is where the cpu has run out of cache memory and began to access memory.



The final graph shows N up to 18,000 in order to observe some of the behavior of the L1, L2 and L3 caches which I assume end around N=1300, N=3000, and N=10,000, respectively, based on the drops in performance that follow the next step. However, to truly rule these out, more finely adjusted values of N would need to be used.



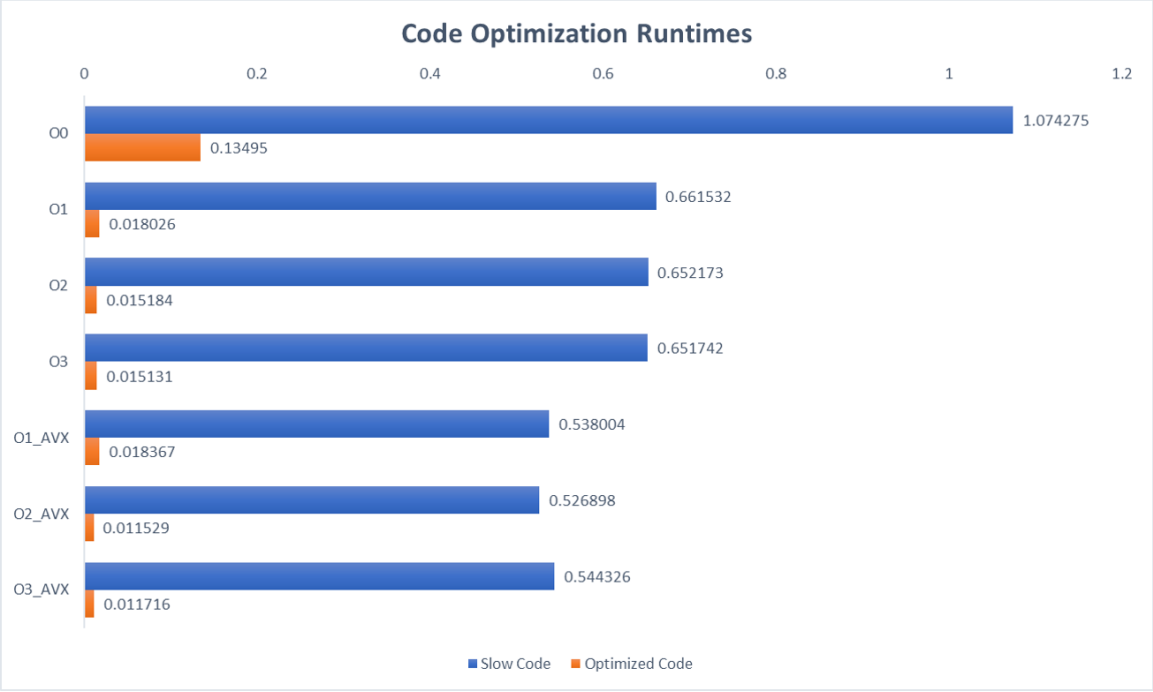
Code Optimization

For the code optimization, I simply moved all of the sine, cosine and floating point operations to a lookup table and populated that before the benchmark loop. I also changed the type of d_val to integer since there was no point in having an unnecessary floating point operation in the loop. The following table shows the drastic boost in performance gained from simply using this lookup table.

Table 3: Code Optimization Execution Times

slow_default	1.074275
slow_O1	0.661532
slow_O2	0.652173
slow_O3	0.651742
slow_O1_mavx	0.538004
slow_O2_mavx	0.526898
slow_O3_mavx	0.544326
optimized_default	0.13495
optimized_O1	0.018026
optimized_O2	0.015184
optimized_O3	0.015131
optimized_O1_mavx	0.018367
optimized_O2_mavx	0.011529
optimized_O3_mavx	0.011716

Finally, the graph below is included in order to visualize the performance increase.



Code Appendix

```
/*
Compiled using:

g++ main.cpp -o main_default
g++ main.cpp -o main_O1 -O1
g++ main.cpp -o main_O2 -O2
g++ main.cpp -o main_O3 -O3
g++ main.cpp -o main_O1_mavx -O1 -mavx
g++ main.cpp -o main_O2_mavx -O2 -mavx
g++ main.cpp -o main_O3_mavx -O3 -mavx

=============================================
CPU: Intel i7-6700k
OS: Ubuntu 16.04

main_default    Elapsed time: 0.134950
main_O1         Elapsed time: 0.018026
main_O2         Elapsed time: 0.015184
main_O3         Elapsed time: 0.015131
main_O1_mavx    Elapsed time: 0.018367
main_O2_mavx    Elapsed time: 0.011529
main_O3_mavx    Elapsed time: 0.011716

*/

#include <stdio.h>
#include <sys/time.h>
#include <vector>
#include <cmath>
#include <stdlib.h>

void get_walltime(double *wcTime)
{
    struct timeval tp;
    gettimeofday(&tp, NULL);
    *wcTime = (double) (tp.tv_sec + tp.tv_usec / 1000000.0);
}

// complex algorithm for evaluation
void myfunc(std::vector<std::vector<double> > &v_s,
            std::vector<std::vector<double> > &v_mat, std::vector<int> &i_v,
            std::vector<double> &value_map)
{
    int d_val;
    for (int j = 0; j < v_s.size(); j++)
    {
        for (int i = 0; i < v_s[0].size(); i++)
        {
            d_val = i_v[i] % 256;
            v_mat[i][j] = v_s[i][j] * (value_map[d_val]);
        }
    }
}

int main(int argc, char *argv[])
{
    // this should be called as> ./slow_code <i_R> <i_N>
    int i_R = 1000;
```

```
59      int i_N = 100;
60
61      double d_S, d_E;
62      // parse input parameters
63      if (argc >= 2)
64      {
65          i_R = atoi(argv[1]);
66      }
67      if (argc >= 3)
68      {
69          i_N = atoi(argv[2]);
70      }
71
72      // some declarations
73      std::vector<std::vector<double> > vd_s(i_N, std::vector<double>(i_N));
74      std::vector<std::vector<double> > vd_mat(i_N, std::vector<double>(i_N));
75      std::vector<int> vi_v(i_N);
76      std::vector<double> vd_difference(256);
77      int sin_val;
78      double sine, cosine;
79      // populate memory with some random data
80      for (int i=0;i<256;i++)
81      {
82          sine = sin(i);
83          cosine = cos(i);
84          vd_difference[i]= sine*sine - cosine*cosine;
85      }
86
87      for (int i = 0; i < i_N; i++)
88      {
89          vi_v[i] = i * i;
90          for (int j = 0; j < i_N; j++)
91          {
92              vd_s[i][j] = j + i;
93          }
94      }
95
96      // start benchmark
97      get_walltime(&d_S);
98
99      // iterative test loop
100     for (int i = 0; i < i_R; i++)
101     {
102         myfunc(vd_s, vd_mat, vi_v, vd_difference);
103     }
104
105     // end benchmark
106     get_walltime(&d_E);
107
108     // report results
109     printf("Elapsed time: %f\n", d_E - d_S);
110
111     return 0;
112 }
```