Project 5: Intel compiler optimization benchmark[1]

Behnam Saeedi
(Saeedib@oregonstate.edu)
CS575: Parallel Programming

Spring 2018

Abstract

CONTENTS

1 SYSTEM SPECS

For the purpose of this assignment we used the Oregon State Universities server. Flip was used in order to generate these results. An lscpu command returned the following:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    2
Core(s) per socket:    6
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 44
Model name:            Intel(R) Xeon(R) CPU           X5650  @ 2.67GHz
Stepping:              2
CPU MHz:               2660.080
BogoMIPS:              5320.16
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              12288K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18,20,22
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
                       cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm
                       pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon
                       pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf pni
                       pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16
                       xtpr pdcm pcid dca sse4_1 sse4_2 popcnt aes lahf_lm
                       tpr_shadow vnmi flexpriority ept vpid dtherm ida arat
```

2 APPROACH

We have setup the c program to be able to dynamically chose how many threads to use from the beginning. This code takes advantage of 1,2,4 and 8 threaded performance in order to compare and contrast the results of the experiment. Random values are assigned to the array A in order to have the program perform properly.

```
for( int i = 0; i < NUMS; i++ )
{
     A[i] = rand() % 1000;
}
```

The following script was made in order to activate the Intel compiler on the flip server for this experiment:

```
#!/bin/csh
setenv INTEL_LICENSE_FILE 28518@linlic.engr.oregonstate.edu
setenv SINK_LD_LIBRARY_PATH /nfs/guille/a2/rh80apps/intel/studio.2013-
```

```
sp1/composer_xe_2015.0.090/compiler/lib/mic/
setenv ICCPATH /nfs/guille/a2/rh80apps/intel/studio.2013-sp1/composer_xe_2015/bin/
set path=( $path $ICCPATH )
source /nfs/guille/a2/rh80apps/intel/studio.2013-sp1/bin/iccvars.csh intel64
```

Furthermore, We are using the following bash command in order to compile the code using the inlet compiler:

```
icpc -DNUMS=100000 -o out main.cpp OMPUtil.cpp
-lm -openmp -align -qopt-report=3 -qopt-report-phase=vec -no-vec
```

After setting up the Intel compiler and test compiling the code another scrip was create to automate the experiment with different numbers of threads and elements:

```
#Non-Vectorized
for thread in `seq 0 3`; do
        for num in `seq 0 14`; do
                icpc -DNUMS=$[2**$num*4000] -o out-$[2**$thread]-$[2**$num*4000] OMPUtil.cpp
        main.cpp -lm -openmp -align-qopt-report=3 -qopt-report-phase=vec -no-vec
                ./out-$[2**$thread]-$[2**$num*4000] $[2**$thread]
        done
done
#Vectorized
for thread in `seq 0 3`; do
        for num in `seq 0 14`; do
                icpc -DNUMS=$[2**$num*4000] -o out-$[2**$thread]-$[2**$num*4000] OMPUtil.cpp
        main.cpp -lm -openmp -align-qopt-report=3 -qopt-report-phase=vec
                ./out-$[2**$thread]-$[2**$num*4000] $[2**$thread]

        done
done
```

These scripts cover 1,2,4 and 8 threaded performance with 4k, to over 65M elements as a power of 2 to 14 times 4k. This will give us 14 nice steps to analyze and compare performance with.

3 Optimization Reports

Each run of the Intel compiler in both non-vectorized and vectorized modes produced a report for each ".cpp" file that was generated. Each one of these reports explains what was changed and why. The reports were stored in files with format ".optrpt". The following two subsections will cover these generated report files.

3.1 Non-Vectorized

Due to use of non-vec, the compiler did not attempt to optimize the code and we can see this by the comment referring to the use of "-no-vec" flag. There was a potential for few loops in the program to be optimized in order to improve performance, but the compiler ignored them.

```
Begin optimization report for: main(int, char **)

    Report from: Vector optimizations [vec]
```

```
LOOP BEGIN at main.cpp(50,2)
   remark #15540: loop was not vectorized: auto-vectorization is disabled with -no-vec flag
LOOP END

LOOP BEGIN at main.cpp(61,2)
   remark #15540: loop was not vectorized: auto-vectorization is disabled with -no-vec flag
LOOP END
=================================================================================
```

3.2 Vectorized

Here we can see that there were few events that got heavily optimized. These are stated in some of the remarks. Another important note to point out in this case is the remark number 15527. Intel compiler does no attempt in optimizing values that are randomly assigned.

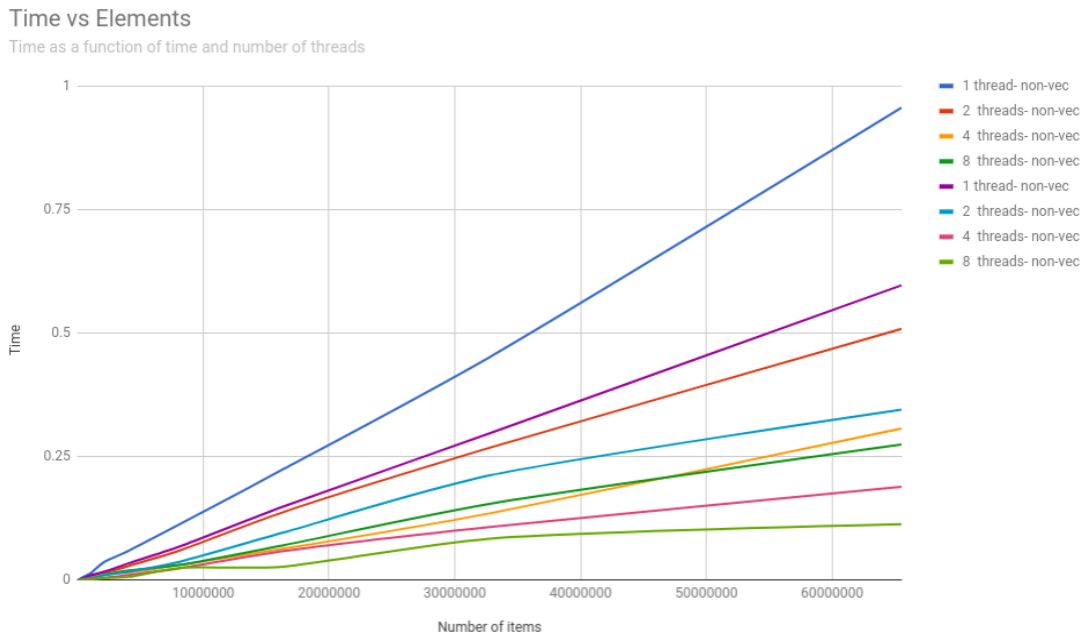
```
Begin optimization report for: main(int, char **)

    Report from: Vector optimizations [vec]


LOOP BEGIN at main.cpp(50,2)
   remark #15527: loop was not vectorized: function call to rand cannot be
   vectorized   [ main.cpp(52,10) ]
LOOP END

LOOP BEGIN at main.cpp(61,2)
<Peeled>
LOOP END

LOOP BEGIN at main.cpp(61,2)
   remark #15300: LOOP WAS VECTORIZED
   remark #15442: entire loop may be executed in remainder
   remark #15448: unmasked aligned unit stride loads: 1
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15467: unmasked aligned streaming stores: 2
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 47
   remark #15477: vector loop cost: 17.000
   remark #15478: estimated potential speedup: 2.760
   remark #15479: lightweight vector operations: 3
   remark #15480: medium-overhead vector operations: 1
   remark #15488: --- end vector loop cost summary ---
LOOP END

LOOP BEGIN at main.cpp(61,2)
   remark #25460: No loop optimizations reported
LOOP END

LOOP BEGIN at main.cpp(61,2)
<Remainder>
```

```
LOOP  END
================================================================================
```

4 PERFORMANCE

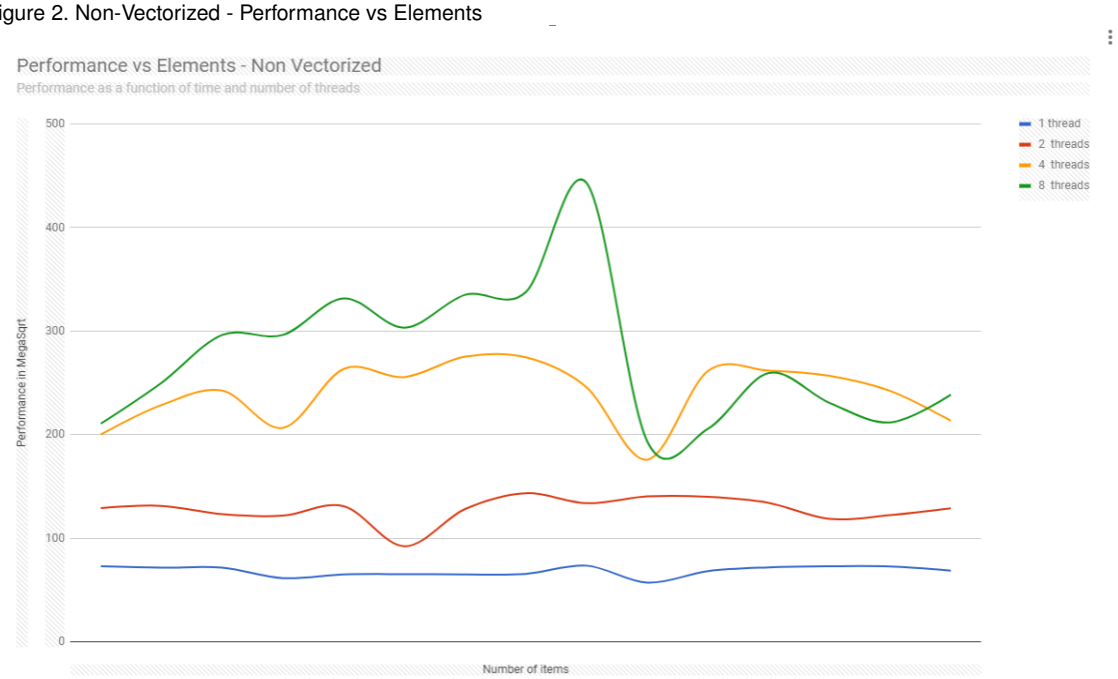
Running the code produced several time intervals that could be analyzed to the following graphs and tables. In this documentation a total of 6 graphs and 4 tables are presented. there are two sets of time trials as a function of threads, time trials as a function if number of elements and performance as a function of number of elements. performance is measured in MegaSqrt/second.

Figure 1. Time vs Elements



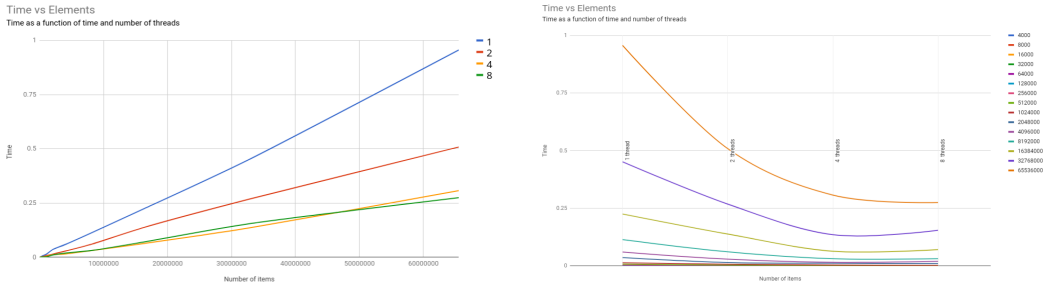
This graph shows all 8 experiments run as a function of time.

Figure 2. Non-Vectorized - Performance vs Elements



The main graph is representing the Performance in MegaSqrt/second. We cam see and compare different number of threads and how they change as number of operations increase. This graph has an odd behavior which I could not figure out what was causing it. I suspect that it is partially due to the time scale and the fact that the values are compared within a span of less than 1 seconds. slightest change in performance can cause significant effects in the shape of our curve. Furthermore, I suspect that the time I selected to run these experiments happen to be a busy time for that particular flip serer. I also think, the few drops in performance that we can see could be due to false sharing. These are visible where the performance drops consistently across all number of threads. Also, we can see that some of the threads (thread 8 for example) sometimes performed worse that the 4 threaded process.

Figure 3. Non-Vectorized - Time vs Elements



These two graphs are with regards to time trials. We can see that this graph looks much smoother and significantly more predictable that the first graph. The reason to this is the fact that the time scale is adjusted for the values. The graph on the left isolates the threads and shows the change as number of elements increase.

The right graph on the other hand is the axis switched. It is the number of operations being constant and the the graph is as a function of number of threads instead.

Table 1
Time VS Elements Non-Vectorized

Item\Thread	1 thread	2 threads	4 threads	8 threads
400	0.000055	0.000031	0.00002	0.000019
8000	0.000112	0.000061	0.000035	0.000032
16000	0.000224	0.00013	0.000066	0.000054
32000	0.000522	0.000263	0.000155	0.000108
64000	0.000987	0.000489	0.000243	0.000193
128000	0.001964	0.00139	0.000501	0.000422
256000	0.003945	0.001998	0.00093	0.000764
512000	0.007827	0.003571	0.001864	0.001516
1024000	0.013954	0.00766	0.004167	0.002311
2048000	0.035983	0.014606	0.011656	0.01059
4096000	0.060189	0.029293	0.015662	0.019921
8192000	0.114227	0.061105	0.031284	0.031581
16384000	0.225171	0.13823	0.063803	0.071028
32768000	0.451583	0.268102	0.135304	0.154747
65536000	0.956398	0.508713	0.307266	0.274802

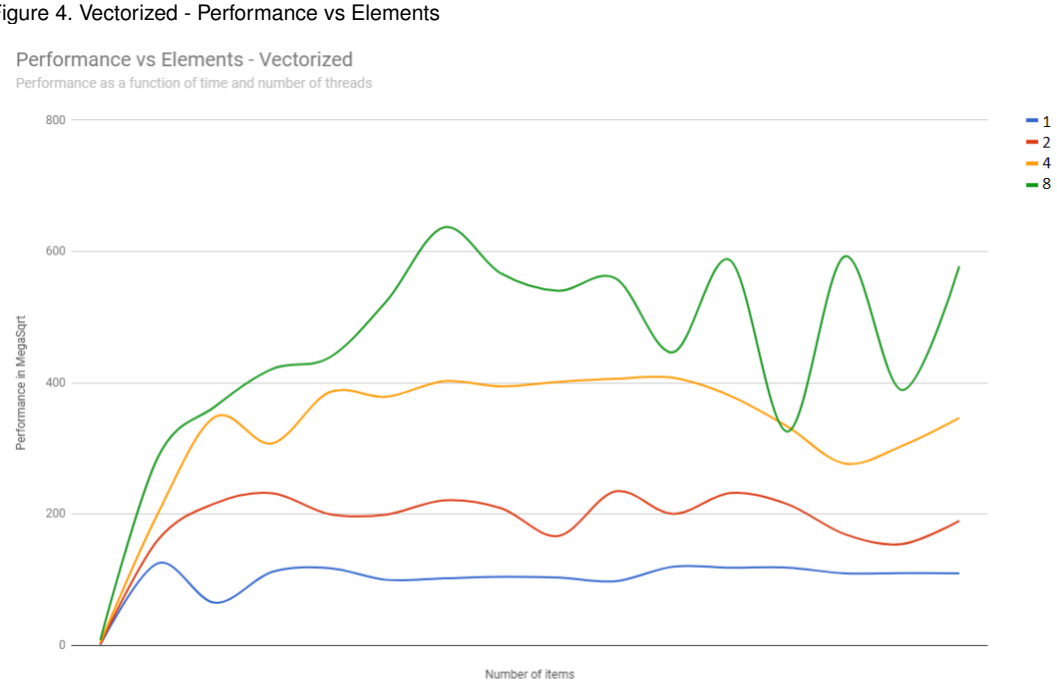
This table contains the information we gathered directly from OMP's timing. We can see some interesting trends in the table and the graph. for example, it appears that there is some amount of false sharing in the code causing certain speed increases on some of the number of elements. This seems to be consistent across different experiments regardless of number of threads.

Table 2
Performance (MegaSqrt/sec) VS Elements Non-Vectorized

Item\Thread	1 thread	2 threads	4 threads	8 threads
4000	72.72727273	129.0322581	200	210.5263158
8000	71.42857143	131.147541	228.5714286	250
16000	71.42857143	123.0769231	242.4242424	296.2962963
32000	61.30268199	121.6730038	206.4516129	296.2962963
64000	64.84295846	130.8793456	263.3744856	331.6062176
128000	65.17311609	92.08633094	255.489022	303.3175355
256000	64.89226869	128.1281281	275.2688172	335.078534
512000	65.41459052	143.3772053	274.6781116	337.7308707
1024000	73.38397592	133.6814621	245.7403408	443.0982259
2048000	56.91576578	140.2163494	175.7035003	193.3899906
4096000	68.05230192	139.828628	261.5247095	205.6121681
8192000	71.71684453	134.0643155	261.8590973	259.3964726
16384000	72.76247829	118.5270925	256.7904331	230.6695951
32768000	72.56251896	122.2221393	242.1805712	211.7520857
65536000	68.52377358	128.8270597	213.2875098	238.4844361

This table on the other hand is generated from the previous one based on the performance.

Figure 4. Vectorized - Performance vs Elements



First thing that we can observe right of the bat is that there is a significant improvement in timing and performance (by a factor of 2 or event more). Another observation is that the curve looks slightly different. (across threads 1,2, and 4 specially). We can see that there is a lot less change in times and performance and I think this is due to the optimization that all of these threads receive.

Figure 5. Vectorized - Time vs Elements

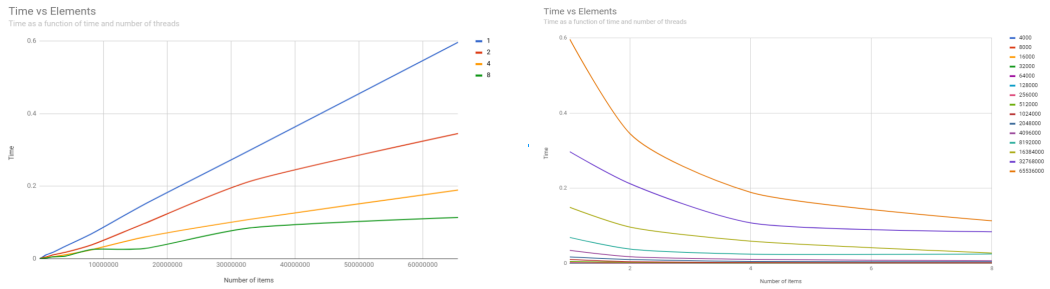


Table 3
Time VS Elements Vectorized

Item\Thread	1	2	4	8
400	0.000032	0.000025	0.00002	0.000014
8000	0.000123	0.000037	0.000023	0.000022
16000	0.000143	0.000069	0.000052	0.000038
32000	0.000272	0.00016	0.000083	0.000073
64000	0.000641	0.000321	0.000169	0.000122
128000	0.001253	0.000579	0.000318	0.000201
256000	0.002449	0.001225	0.000649	0.000452
512000	0.00495	0.00307	0.001276	0.000948
1024000	0.010469	0.004362	0.002522	0.001832
2048000	0.017064	0.010219	0.005025	0.004587
4096000	0.034615	0.017656	0.010772	0.006984
8192000	0.069149	0.03807	0.024573	0.025145
16384000	0.149419	0.096664	0.059141	0.027647
32768000	0.297486	0.212242	0.107837	0.084253
65536000	0.597088	0.345329	0.189217	0.113469

Table 4
Performance (MegaSqrt/sec) VS Elements Vectorized

Item\Thread	1 thread	2 threads	4 threads	8 threads
4000	125	160	200	285.7142857
8000	65.04065041	216.2162162	347.826087	363.6363636
16000	111.8881119	231.884058	307.6923077	421.0526316
32000	117.6470588	200	385.5421687	438.3561644
64000	99.84399376	199.376947	378.6982249	524.5901639
128000	102.1548284	221.0708117	402.5157233	636.8159204
256000	104.5324622	208.9795918	394.4530046	566.3716814
512000	103.4343434	166.7752443	401.2539185	540.0843882
1024000	97.81258955	234.7546997	406.0269627	558.9519651
2048000	120.0187529	200.4109991	407.5621891	446.4791803
4096000	118.3302037	231.9891255	380.2450798	586.4833906
8192000	118.4688137	215.1825584	333.3740284	325.7904156
16384000	109.651383	169.4943309	277.0328537	592.6140268
32768000	110.1497213	154.3898003	303.86602	388.9238365
65536000	109.7593655	189.7784432	346.3536574	577.5674413

The numbers in table also confirm that there is an overall improvement in performance of the code after removing the -no-vec flag.

5 Analysis of results

5.1 Unit analysis

The unit used for the purpose of this experiment is MegaSqrt. This unite is computed by dividing the number of elements by time and the divide all of that by 1 million. This will allow us to compare the performance better and understand what the timing is implying in this experiment.

The non-vectorized and vectorized graphs look fairly similar. However, there are few important differences between them that is worth noting. The first important note is the time difference between the two approaches. We can see that on average the optimized code runs much much faster and performs more per given time. Also we can see for each given number of elements as we increase the number of threads, our performance increases and our time for performing that operation decreases. This is visible in figures 3 and 5 on the time as a function of number of elements portion (right figure).

6 CONCLUSION

In conclusion, the graphs illustrate a few interesting points about the performance of Intel compiler optimized and unoptimized. One of the general trends that we observed was the fact that there is a significant increase in the performance after optimization. Another interesting result was that Intel's optimizer actually takes into account on how much of the performance could be improved by vectorization. It estimates a potential speed up which could be found in the optimization reports after compilation.