Characterizing the Effectiveness of Compilers in Vectorizing Polyhedrally Transformed Code

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Yogesh Chidambarnathan

Graduate Program in Computer Science and Engineering

The Ohio State University

2013

Master's Examination Committee:

Dr. P. Sadayappan, Advisor

Dr. Ümit V. Çatalyürek
© Copyright by

Yogesh Chidambarnathan

2013
Abstract

Many of the compute intensive applications spend most of their time inside nested loops. Hence optimization of these nested loops can provide significant improvements in the speed of the program. A number of optimizations can be performed to a program in order to speed it up on a particular hardware. Optimization techniques such as Tiling, Vectorization, Loop Unrolling etc. can produce significantly better performance. In this study we focus on Tiling and Vectorization. Our study is to evaluate for various benchmarks and various problem sizes, whether one optimization affects the other or not. In cases where one optimization negatively affects the other, we evaluate the extent of the negative effect, which gives us an understanding of the type of optimization that should be performed in order to get an overall gain in the speed. This study evaluates two tiling schemes namely, PLuTo and PTile, with two compilers namely, the GNU C Compiler and the Intel C Compiler.
I dedicate my work to my parents and my brother
Acknowledgments

I would like to express my deep gratitude to my erudite mentors, Professor P. Sadayappan and Professor Ümit V. Çatalyürek for their constant help and support throughout my Masters program. Their easy approachability and valuable guidance stood me in good stead. Their presence has been greatly inspiring encouraging me to put my best foot forward.

I would also like to thank Kevin Stock, Martin Kong and Justin Holewinski for their valuable technical help during the course of the program.

I am grateful to my friends Abhishek, Pranth, Nithin, Deepan, Monadhika, Ritu, Jithendra, Abhijeet, Shweta and Arati for their motivation and support.

I am extremely thankful to my parents, Mr. G. Chidambarnathan, Mrs. C. Saraswathi and my brother C. Yuvaraj, who have been encouraging and motivating me to give my best, always.
Vita


June 2010...........................................Master of Science (Technology)

Information Systems,

BITS Pilani Goa Campus, India.

Sept 2010 – June 2011..........................Project Assistant,

Indian Institute of Science, Bangalore.

Oct 2011 – June 2012...........................Graduate Research Associate,

Ohio Supercomputer Center, Columbus.

June 2012 – Aug 2012..........................SDE Intern, Cisco Systems,

San Jose, California.

Fields of Study

Major Field: Computer Science and Engineering

Studies in:

High Performance Computing Prof. P. Sadayappan
Parallel Computing Prof. P. Sadayappan
# Table of Contents

Abstract ......................................................................................................................... ii

Dedication ....................................................................................................................... iii

Acknowledgements ......................................................................................................... iv

Vita ................................................................................................................................. v

List of Tables .................................................................................................................. viii

List of Figures ............................................................................................................... ix

1. Introduction .................................................................................................................... 1
   1.1 Problem Description ................................................................................................. 1
   1.2 Objective .................................................................................................................. 2

2. Data Aggregation and Analysis ................................................................................. 4
   2.1 Experimental Setup .................................................................................................. 5
   2.2 PAPI Counters ........................................................................................................ 5
      2.2.1 List of PAPI Counters Used ............................................................................. 6
   2.3 Data Generation ...................................................................................................... 7

3. Result Interpretation and Analysis .......................................................................... 9
   3.1 Calculations ............................................................................................................. 9
      3.1.1 Formula 1 .......................................................................................................... 9
      3.1.2 Formula 2 ......................................................................................................... 10
      3.1.3 Formula 3 ......................................................................................................... 10
   3.2 Analysis .................................................................................................................. 10
4. Key Observations and Conclusions

4.1 Classification

4.2 Case 1: Tiling reduces percentage vectorization for both PLuTo and PTile

4.3 Case 2: Better vectorization happens with PTile than with no tiling

4.4 Case 3: PLuTo performs better than PTile in terms of vectorization

4.5 Case 4: PTile performs better than PLuTo in terms of vectorization

4.6 Case 5: Tiling does not affect the percentage vectorization

4.7 Conclusion

Bibliography
List of Tables

Table 1: GFLOPS for various sizes (correlation) ................................................................. 53
Table 2: GFLOPS for various sizes (adi) ........................................................................... 53
Table 3: GFLOPS for various sizes (atax) ........................................................................ 54
Table 4: GFLOPS for various sizes (gesummv) ................................................................. 55
Table 5: GFLOPS for various sizes (bicg) ............................................................................ 55
Table 6: GFLOPS for various sizes (3mm) ....................................................................... 57
Table 7: GFLOPS for various sizes (symm) ..................................................................... 58
List of Figures

Figure 1: Percentage Vectorization for 2mm benchmark ........................................... 11
Figure 2: Performance of 2mm benchmark ................................................................. 12
Figure 3: Percentage Vectorization for 3mm benchmark ........................................... 13
Figure 4: Performance of 3mm benchmark ................................................................. 14
Figure 5: Percentage Vectorization for atax benchmark ........................................... 15
Figure 6: Performance of atax benchmark ................................................................. 16
Figure 7: Percentage Vectorization for cholskey benchmark .................................... 17
Figure 8: Performance of cholskey benchmark ......................................................... 18
Figure 9: Percentage Vectorization for doitgen benchmark ..................................... 19
Figure 10: Performance of doitgen benchmark ......................................................... 20
Figure 11: Percentage Vectorization for gemm benchmark ..................................... 21
Figure 12: Performance of gemm benchmark ......................................................... 22
Figure 13: Percentage Vectorization for gemver benchmark .................................. 23
Figure 14: Performance of gemver benchmark ......................................................... 24
Figure 15: Percentage Vectorization for gesummv benchmark ................................ 25
Figure 16: Performance of gesummv benchmark .................................................... 25
Figure 17: Percentage Vectorization for symm benchmark ..................................... 27
Figure 18: Performance of symm benchmark ......................................................... 28
Figure 19: Percentage Vectorization for syr2k benchmark ..................................... 29
Figure 20: Performance of syr2k benchmark ......................................................... 29
Figure 21: Percentage Vectorization for durbin benchmark .................................... 31
Figure 22: Performance of durbin benchmark .......................................................... 32
Figure 23: Percentage Vectorization for gramschmidt benchmark ...................... 33
Figure 24: Performance of gramschmidt benchmark ............................................. 34
Figure 25: Percentage Vectorization for lu benchmark ......................................... 35
Figure 26: Performance of lu benchmark ............................................................... 35
Figure 27: Percentage Vectorization for ludcmp benchmark ................................. 36
Figure 28: Performance of ludcmp benchmark ..................................................... 37
Figure 29: Percentage Vectorization for correlation benchmark ............................. 38
Figure 30: Performance of correlation benchmark .................................................. 38
Figure 31: Percentage Vectorization for covariance benchmark ............................ 39
Figure 32: Performance of covariance benchmark ............................................... 40
Figure 33: Percentage Vectorization for adi benchmark (GNU C Compiler) ......... 41
Figure 34: Percentage Vectorization for adi benchmark (Intel C Compiler) .......... 42
Figure 35: Performance of adi benchmark ............................................................ 43
Figure 36: Percentage Vectorization for fdtd-2d benchmark (GNU C Compiler) .... 44
Figure 37: Percentage Vectorization for fdts-2d benchmark (Intel C Compiler) ...... 45
Figure 38: Performance of jacobi-1d-imper benchmark ....................................... 46
Figure 39: Percentage Vectorization for jacobi-1d-imper benchmark .................... 47
Figure 40: Performance of jacobi-1d-imper benchmark ........................................ 47
Figure 41: Percentage Vectorization for jacobi-2d-imper benchmark (GNU C Compiler) ........................................................ 48
Figure 42: Percentage Vectorization for jacobi-2d-imper benchmark (Intel C Compiler) 49
Figure 43: Performance of jacobi-2d-imper benchmark

Figure 44: Percentage Vectorization for bicg benchmark

Figure 45: Performance of bicg benchmark
Chapter 1: Introduction

1.1 Problem Description

Many of the compute intensive applications spend most of their time inside nested loops. Hence optimization of these nested loops can provide significant improvements in the speed of the program. Different kinds of transformations can be performed on these nested loops. Some of them are, loop vectorization, loop unrolling, tiling, loop permutation etc. Each of them change the structure of the loop in order to exploit the architectural feature of the particular hardware which reduces the execution time without changing the result.

Vectorization is Parallel Computing technique which can be performed on modern processors to simultaneously perform a single operation on multiple data [4]. Loop vectorization is performed on nested loops, which assigns each pairs of data items to separate processing units. The same instruction is simultaneously executed on different pairs of data items, hence the computation time is reduced.

Tiling is performed with the aim of increasing cache hits. The access pattern inside the loop is changed to exploit data locality and increase data reuse [3]. The idea is to pull a small set of data from the main memory, fit it inside the cache and reuse it as much as possible before over-writing it with a different set of data. This can improve the performance by a great factor as cache is much faster than the main memory. Different
schemes can be used to perform tiling and each can produce a different effect on a particular benchmark. The size of the tile can also be varied depending upon how much data can be fitted inside the cache.

In our study we have considered two types of tiling

1) Fixed Tiling with PLuTo

PLuTo, is a fully automatic polyhedral source-to-source transformation program which is driven by an integer linear optimization framework based on the Polyhedral model, a model for compiler optimization which provides an abstraction to perform high-level transformations such as loop-nest optimization and parallelization on affine loop nests [1].

2) Parametric Tiling using PTile with full tile separation

PTile is a software tool to generate parallel parametric tiled code (where the generated tiled code is parallel and tile sizes are symbolic constants that can be set during run-time) for affine loop nests: loop nests whose bounds and array access functions are affine functions of loop-iterators and program parameters [2].

1.2 Objective

The aim of any sort of optimization is to obviously speedup the code. Both tiling and vectorization are optimizations which can greatly reduce the execution time of a code. Tiling introduces more nesting inside loops thereby making it more complex. This complexity can cause the failure of vectorization heuristics used by the compiler to
vectorize the loops which can have a negative impact on the performance. The objective of this study is to compare the effect of tiling on vectorization for different benchmarks and analyze the overall effect on the speed (in terms of GFLOPS) of the particular benchmark. This study is performed using the GNU C Compiler and the Intel C Compiler after transforming code with Fixed Tiling using PLuTo and Parametric Tiling with full tile separation using PTile.
Chapter 2: Data Aggregation and Analysis

The aim of performing this analysis is to characterize the effectiveness of vectorization by two compilers on twenty nine different benchmarks for varying problem sizes tiled using two different schemes. A large amount of data should be collected for every run performed under similar conditions in order to ensure that the difference in output generated by varying problem sizes and varying compilers is not affected by any external factor. To ensure this, a large number of runs are performed.

For every run, values of various PAPI counters are reported, which indicate the performance of the benchmark for that particular run. The average of the counter values of every run is taken and we keep an eye on the deviation by reporting the minimum, maximum and standard deviation value. These average values can be used to draw conclusion about the performance.

Once the data is generated it can be viewed individually for every benchmark. For every benchmark, we can use the average value of counters by first fixing the compiler and varying the problem sizes and then fixing the problem sizes and varying the compiler, for all three cases, i.e.

1) No Tiling

2) Fixed Tiling using PLUTO
3) Parametric tiling with full tile separation

2.1 Experimental Setup

As mentioned in the previous chapter, the objective was to collect a large amount of data for every benchmark by varying the problem size and the compiler. The experiment was performed on Oakley, the Ohio Supercomputer Center HP Intel Xeon x5650 Cluster. Each node of Oakley has 12 cores with clock rate of 2666.768 MHz with 48 GB of memory per node.

Separate directories were created for each tiling scheme and each compiler. Each directory contained a copy of all the 30 benchmarks. Separate batch scripts were created for each benchmark and the output was collected in separate files. A file containing problem sizes was present in each directory. The scripts pulled up the problem sizes information from this file, and ran the benchmark five times for every problem size present in that file. We used a tile size of 32 for problems with less than 3 dimensional data.

2.2 PAPI Counters

PAPI is the acronym for Performance Application Programming Interface. The modern microprocessors have counters that exist as small set of registers that count events. The
values of these counters can be used to perform a variety of performance analyses of a particular code. For example, a counter which counts the number of L1 cache misses would give us an idea of how efficiently the program makes use of the cache. It can help us optimize the code in such a way that produces less cache misses for that particular kind of hardware thereby speeding up the code.

2.2.1 List of PAPI Counters Used

Following is the list of PAPI counters which we used for our study.

- "PAPI_L1_DCM"  Level 1 data cache misses
- "PAPI_L2_DCM"  Level 2 data cache misses
- "PAPI_L1_TCM"  Level 1 cache misses
- "PAPI_L2_TCM"  Level 2 cache misses
- "PAPI_L3_TCM"  Level 3 cache misses
- "PAPI_TOT_IIS"  Instructions issued
- "PAPI_TOT_INS"  Instructions completed
- "PAPI_FP_INS"  Floating point instructions
- "PAPI_LD_INS"  Load instructions
- "PAPI_SR_INS"  Store instructions
- "PAPI_RES_STL"  Cycles stalled on any resource
- "PAPI_TOT_CYC"  Total cycles
- "PAPI_LST_INS"  Load/store instructions completed
- "PAPI_FP_OPS"  Floating point operations
- "PAPI_SP_OPS"  Floating point operations; optimized to count scaled single precision vector operations
- "PAPI_DP_OPS"  Floating point operations; optimized to count scaled double precision vector operations
- "PAPI_VEC_SP"  Single precision vector/SIMD instructions
- "PAPI_VEC_DP"  Double precision vector/SIMD instructions

### 2.3 Data Generation

We characterize the effect of vectorization on the original code and code transformed using two tiling schemes for two different compilers. So we have $3 \times 2$ different directories each having the benchmark c files. Each directory has a file describing problem sizes which are of three types 1) power of 4 sizes 2) $13/16$ of the power of 4 sizes used in 1st part, and, 3) $19/16$ of the power of 4 sizes used in 1st part. The power of 4 sizes will line up in the cache better. Whereas slightly lower and slightly higher than power of 4 problem sizes will give us a more realistic picture as real world problem sizes are not powers of 2 most of the times. A tile size of 32 was used in all the cases.

We used a separate script for each benchmark so that they could be run in parallel on the cluster. We used the –O3 (level three optimization) option during compilation for both the compilers to enable vectorization.
Each script first transforms the code using the following command

1) PLuTo

pocc --pluto --pluto-tile --pluto-parallel --pragmatizer --vectorizer input.c –o output.c

2) PTile

pocc --pluto --ptile --ptile-fts --past-hoist-lb --pragmatizer --vectorizer input.c –o output.c

For the case where no tiling is done, the original code is directly fed to the compiler.

Once the codes are polyhedrally transformed, they are compiled using the following commands

1) gcc -O3 -DPOLYBENCH_PAPI -I/usr/include -L/usr/lib64 -lpapi -lm -Wl,-rpath /usr/lib64 polybench.c -DPROBSIZE=$j "$k.32.c" -I/nfs/12/yogesh/thesis/pocc-1.1/polybench-c-3.2/utilities -o "$k.32.$j.x"

2) icc -O3 -DPOLYBENCH_PAPI -I/usr/include -L/usr/lib64 -lpapi -lm -Wl,-rpath /usr/lib64 polybench.c -DPROBSIZE=$j "$k.32.c" -I/nfs/12/yogesh/thesis/pocc-1.1/polybench-c-3.2/utilities -o "$k.32.$j.x"

In the above commands, $k and $j are variables inside the script.

The scripts were run in parallel and the outputs were collected in a file and the average, min, max and standard deviation were calculated.
Chapter 3: Result Interpretation and Analysis

After conducting the experiment as mentioned in the previous section, we had the value of the 18 PAPI counters for every benchmark, run without tiling, with PLuTo and with PTile, for different problem sizes, for different compilers with a fixed tile size of 32.

We looked at all the 29 benchmarks to see the effect of tiling on vectorization on each of the result that we have obtained. We took the average of five runs for every counter value. Values of all the runs were copied to a spreadsheet and the average, min, max and standard deviation values were calculated.

To characterize the effectiveness of compilers in vectorizing the polyhedrally transformed code, we looked at the fraction of vectorized floating point operations in each case. We used the following formulas to calculate the percentage of vectorized floating point operations from the PAPI counter values which gives us a very good idea of the vectorization that is being done by the compiler. We also calculate GFLOPS.

3.1 Calculations

3.1.1 Formula 1

Percentage of vectorized floating point operations

\[
\text{Percentage} = \frac{(\text{Double precision vector/SIMD instructions} \times 2)}{(\text{Double precision vector/SIMD instructions} \times 2 + \text{Scalar Operations})}
\]
The denominator in the above equation is the total number of floating point operations which includes the scalar operations. To calculate the scalar operations one can apply the following formula.

3.1.2 Formula 2

Scalar Operations = Floating point instructions - Double precision vector/SIMD instructions

3.1.3 Formula 3

GFLOPS = Total FPO * Clock Rate in GHz / Total Cycles

Let us analyze the above vectorization percentage value and GFLOPS value for every benchmark.

3.2 Analysis

1) 2mm benchmark
   a) GCC case

We observe that in case of GCC, no vectorization occurs without tiling. But after tiling is performed using PLuTo or PTile, vectorization occurs for a few problem sizes.

   b) ICC case

It is interesting to note that ICC as compared to GCC, performs close to 90% vectorization in case of no tiling. With PLuTo, ICC performs more vectorization as
compared to the case where no tiling occurs. With PTile the fraction of vectorized operations is pretty low compared to no tiling or PLuTo.

The following is a line graph plotted for the ICC Compiler

![2mm / Intel C Compiler](image)

Figure 1: Percentage Vectorization for 2mm benchmark
2) 3mm benchmark

   a) GCC case

      We see absolutely no vectorization in all three cases, that is without tiling, with PLuTo and with PTile.

   b) ICC case

      Similar to the 2mm case, ICC performs very good vectorization compared to GCC.

      The percentage of vectorization without tiling is 100 for 62, 256, 208 and 1216 problem sizes. The percentage is lowest for problem size 19.
With tiling, vectorization with PLuTo is far better than with PTile, as we observed in the 2mm case. The following is the line graph along with the percentage under the problem size axis.

![Graph showing vectorization percentages for Normal, Pluto, and Ptile with 3mm benchmark.](image)

**Figure 3:** Percentage Vectorization for 3mm benchmark

We see that in both 2mm and 3mm cases, the graphs look very similar in all the three cases.
3) atax benchmark

a) GCC case

Similar to the previous cases, absolutely no vectorization occurs in all the three cases.

b) ICC case

Unlike the previous cases, PTile performs much better vectorization than PLuTo which does absolutely no vectorization. The following line graph gives a clear picture of how well PTile performs compared to no tiling case and PLuTo tile case.
Figure 5: Percentage Vectorization for atax benchmark
4) cholskey benchmark

a) GCC case

Similar to the previous cases, GCC does not vectorize in all the three cases.

b) ICC case

Like other benchmarks, ICC does perform a good amount of vectorization. Below is the line graph for this case. All the three lines almost overlap, which means that the fraction of vectorized operations is exactly the same for all the three cases for a fixed problem size. Also, the fraction of vectorized operations increases with the problem size.
Figure 7: Percentage Vectorization for cholskey benchmark

The three lines overlap in the above graph.
5) doitgen benchmark

a) GCC case

Similar to the previous cases, GCC does not vectorize in all the three cases.

b) ICC case

Good amount of vectorization occurs in all the three cases. PLuTo performs marginally better than PTile. The shape of the line graph is very similar in all the three cases. The percentage of vectorization in general increases with the problem size.
Figure 9: Percentage Vectorization for doitgen benchmark
6) gemm benchmark

a) GCC case

We see very small amount of vectorization with GCC for PLuTo and PTile for problem size 13. For other sizes the vectorization is either too small or is zero.

b) ICC case

A good amount of vectorization happens in all the three cases. PLuTo performs marginally better than PTile. Overall there is better vectorization without tiling. The amount of vectorization increases with the problem size.

Here is the line graph which gives a clear picture for this case.
Figure 11: Percentage Vectorization for gemm benchmark
7) gemver benchmark

a) GCC case

The percentage of vectorized floating point operations is either too small or zero in most of the cases.

b) ICC case

We observe that the percentage of vectorized floating operations is less than 50% in all the three cases. We obtain maximum vectorization without tiling. With tiling, PTile performs much better than PLuTo which is evident from the graph below.
Figure 13: Percentage Vectorization for gemver benchmark
8) gesummv benchmark

   a) GCC case

      For GCC in case of PLuTo, 4-8% vectorization occurs for very small values of the problem size, where as in case of PTile, the vectorization is zero.

   b) ICC case

      Contrary to the previous cases, vectorization does not happen in the case where no tiling is done. Whereas there is a very good percentage of vectorization happening in case of PTile. In case of PLuTo, vectorization is close to zero for all the cases.
Figure 15: Percentage Vectorization for gesummv benchmark

Figure 16: Performance of gesummv benchmark
9) mvt benchmark

   a) GCC and ICC cases

       No vectorization happens in any of the three cases.

10) symm benchmark

   a) GCC case

       No vectorization happens in any of the three cases.

   b) ICC case

       Like cholskey benchmark, here also we notice that all the three lines almost overlap with each other. The percentage of vectorization increases with the problem size.
Figure 17: Percentage Vectorization for symm benchmark

<table>
<thead>
<tr>
<th></th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>13</th>
<th>52</th>
<th>208</th>
<th>832</th>
<th>19</th>
<th>76</th>
<th>304</th>
<th>1216</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>40.255</td>
<td>77.677</td>
<td>90.469</td>
<td>93.953</td>
<td>92.798</td>
<td>48.777</td>
<td>89.855</td>
<td>93.953</td>
<td>91.495</td>
<td>93.019</td>
<td>93.019</td>
<td>93.019</td>
</tr>
<tr>
<td>Pluto</td>
<td>36.39</td>
<td>76.042</td>
<td>90.339</td>
<td>93.229</td>
<td>92.698</td>
<td>41.886</td>
<td>89.405</td>
<td>92.698</td>
<td>91.176</td>
<td>92.993</td>
<td>92.993</td>
<td>92.993</td>
</tr>
<tr>
<td>Ptile</td>
<td>37.031</td>
<td>76.117</td>
<td>90.576</td>
<td>93.586</td>
<td>92.382</td>
<td>42.035</td>
<td>89.321</td>
<td>93.522</td>
<td>94.26</td>
<td>91.394</td>
<td>94.26</td>
<td>94.26</td>
</tr>
</tbody>
</table>
11) syr2k benchmark

a) GCC case

Percentage of vectorization in all the three cases is close to zero.

b) ICC case

Almost no vectorization happens in case of PTile and the case where no tiling is done. PLuTo performs very well in this case. Highest vectorization percentage occurs when the problem size is 256.
Figure 19: Percentage Vectorization for syr2k benchmark

Figure 20: Performance of syr2k benchmark
12) syrk benchmark
   a) GCC case
      Percentage of vectorization in both the cases is close to zero.
   b) ICC case
      In general, in this case, PLuTo performs marginally better than PTile. The three lines almost overlap with each other except the case when the problem size is 13, where without tiling, the vectorization is much better.

13) trisolv benchmark
   a) GCC and ICC
      No vectorization happens in any of the three cases.

14) trmm benchmark
   a) GCC and ICC cases
      No vectorization happens in any of the three cases.

15) durbin benchmark
   a) GCC case
      No vectorization happens in any of the three cases.
   b) ICC case
A good percentage of vectorization occurs in all three cases. Also, the three lines almost perfectly overlap, which means that tiling has no effect on the percentage of vectorization.

Figure 21: Percentage Vectorization for durbin benchmark
16) dynprog benchmark
   
   a) GCC and ICC cases

   No vectorization happens in any of the three cases.

17) gramschmidt benchmark
   
   a) GCC case

   No vectorization happens in any of the three cases.

   b) ICC case

   The three lines almost overlap with each other except in cases of small non-power of four problem size, for example 13 and 19.

Figure 22: Performance of durbin benchmark
Figure 23: Percentage Vectorization for gramschmidt benchmark
18) lu benchmark

   a) GCC

      Percentage of vectorization is zero in this case.

   b) ICC case

      Vectorization in case of PLuTo is almost as good as the case where no tiling is
done, except for problem size 52 and 76.

      Vectorization in case of PTile is much worse compared to PLuTo. In case of
PTile, the percentage of floating point operations vectorized is less for very small
and very big problem sizes. The percentage is high at 64, 52 and 76.
Figure 25: Percentage Vectorization for lu benchmark

Figure 26: Performance of lu benchmark
19) Ludcmp benchmark

a) GCC case

Similar to the previous cases, GCC does not vectorize in all the three cases.

b) ICC case

Like other benchmarks, ICC does perform a good amount of vectorization.

Below is the line graph for this case. All the three lines almost overlap, which means that the fraction of vectorized operations is exactly the same for all the three cases for a fixed problem size. Also, the fraction of vectorized operations increases with the problem size.

![Figure 27: Percentage Vectorization for ludcmp benchmark](image-url)
20) Correlation benchmark

a) GCC case

For GCC in case of PTile, 4-8% vectorization occurs for very small values of the problem size, where as in case of PLuTo, the vectorization is zero.

b) ICC case

Tiling reduces the percentage of vectorized floating operations greatly for both PLuTo and PTile. A small percentage of vectorization does occur for very small values of the problem size.
Figure 29: Percentage Vectorization for correlation benchmark

Figure 30: Performance of correlation benchmark
21) Covariance benchmark

a) GCC case

The percentage of vectorization is very less or close to zero with all the three cases.

b) ICC case

High percentage of vectorization occurs in case of no-tiling and in the case when it is tiles with PLuTo. Normal and PLuTo lines overlap more for higher problem sizes. In case of PTile, almost no vectorization occurs.

![Figure 31: Percentage Vectorization for covariance benchmark](image-url)
Figure 32: Performance of covariance benchmark

22) Floyd-warshall benchmark
   a) GCC and ICC cases
      No vectorization happens in any of the cases.

23) reg_detect benchmark
    a) GCC and ICC cases
       No vectorization happens in any of the cases.

24) adi benchmark
a) GCC case

Unlike the previous 21 benchmarks, here vectorization does happen for large problem sizes in the case where no tiling is done. But like other cases, vectorization by GCC after tiling with either PLuTo or PTile is very less.

![Percentage Vectorization for adi benchmark (GNU C Compiler)](image)

Figure 33: Percentage Vectorization for adi benchmark (GNU C Compiler)

b) ICC case

High percentage of vectorization occurs in the case where no tiling is done. Vectorization reduces to less than 50% in general when the code is tiled with PLuTo. With PTile the vectorization is close to zero.
Figure 34: Percentage Vectorization for adi benchmark (Intel C Compiler)
25) ftdt-2d benchmark

a) GCC case

Unlike the first 21 benchmarks, here vectorization does happen for large problem sizes in the case where no tiling is done. But like other cases, vectorization by GCC after tiling with either PLuTo or PTile is very less.
b) ICC case

High percentage of vectorization occurs in the case where no tiling is done, whereas vectorization is almost reduced to zero in cases where tiling is done with PLuTo and PTile.

Figure 36: Percentage Vectorization for fdtd-2d benchmark (GNU C Compiler)
Figure 37: Percentage Vectorization for fdts-2d benchmark (Intel C Compiler)
Figure 38: Performance of jacobi-1d-imper benchmark

26) jacobi-1d-imper benchmark

   a) GCC case

      No vectorization occurs in any of the three cases.

   b) ICC case

      High percentage of vectorization occurs in the case where no tiling is done, whereas vectorization is almost reduced to zero in cases where tiling is done with PLuTo and PTile.
Figure 39: Percentage Vectorization for jacobi-1d-imper benchmark

Figure 40: Performance of jacobi-1d-imper benchmark
27) jacobi-2d-imper benchmark

a) GCC case

49.721% of vectorization occurs in the case where no tiling is done for problem size 19. For other problem sizes, vectorization percentage is zero. When tiling is done using PLuTo and PTile, a small percentage of vectorization occurs for size 13 and 19, for other problem sizes it is close to zero.

Figure 41: Percentage Vectorization for jacobi-2d-imper benchmark (GNU C Compiler)
b) ICC case

A high percentage of vectorization occurs in the case where no tiling is done. The percentage increases with the size of the problem. When the code is tiled with PLuTo the percentage of vectorization is marginally less than the no-tiling case. With PTile, the vectorization percentage is close to zero.

Figure 42: Percentage Vectorization for jacobi-2d-imper benchmark (Intel C Compiler)
Figure 43: Performance of jacobi-2d-imper benchmark

28) seidel-2d benchmark
   a) GCC and ICC cases
      No vectorization happens in any of the three cases.

29) bicg benchmark
   a) GCC case
      No vectorization happens in any of the three cases.
   b) ICC case
      No vectorization happens in the no-tiling case and in the case where the code is tiled with PLuTo. A good percentage of vectorization occurs with PTile, which increases with the size of the problem.
Figure 44: Percentage Vectorization for bicg benchmark

Figure 45: Performance of bicg benchmark
Chapter 4: Key Observations and Conclusions

4.1 Classification

We classified our observation in terms of the percentage of vectorization into the following cases

1) Tiling reduces percentage vectorization for both PLuTo and PTile.
2) Better vectorization happens with PTile than with no tiling.
3) PLuTo performs better than PTile in terms of vectorization
4) PTile performs better than PLuTo in terms of vectorization
5) Tiling does not affect the percentage vectorization

After classification, we also discuss the performance of the code in GFLOPS.

4.2 Case 1: Tiling reduces percentage vectorization for both PLuTo and PTile.

This happens in the following cases

1) gemver
2) correlation
3) adi
4) fdt-2d
5) jacobi-1d-imper
This type of behavior is expected. Tiling increases the complexity of the loops hence the vectorization heuristics used by the compilers, fail to vectorize the loops well. In three out of the above five cases, namely, correlation, fdtd-2d and Jacobi-1d-imper, the performance of ICC-PTile was found to be much better than the others.

Table 1: GFLOPS for various sizes (correlation)

<table>
<thead>
<tr>
<th>Size</th>
<th>GCC-Normal (GFLOPS)</th>
<th>GCC-PluTo (GFLOPS)</th>
<th>GCC-PTile (GFLOPS)</th>
<th>ICC-Normal (GFLOPS)</th>
<th>ICC-PluTo (GFLOPS)</th>
<th>ICC-PTile (GFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0.704699</td>
<td>0.638929</td>
<td>0.564884</td>
<td>0.720987</td>
<td>0.681908</td>
<td>1.125712</td>
</tr>
<tr>
<td>64</td>
<td>1.437869</td>
<td>0.664629</td>
<td>1.173308</td>
<td>1.197982</td>
<td>1.398481</td>
<td>1.784046</td>
</tr>
<tr>
<td>256</td>
<td>0.438042</td>
<td>0.547751</td>
<td>0.615731</td>
<td>0.438389</td>
<td>0.62408</td>
<td>1.23937</td>
</tr>
<tr>
<td>1024</td>
<td>0.374034</td>
<td>0.50305</td>
<td>0.567765</td>
<td>0.445611</td>
<td>0.583049</td>
<td>1.162247</td>
</tr>
<tr>
<td>13</td>
<td>0.560146</td>
<td>0.536218</td>
<td>0.439406</td>
<td>0.503058</td>
<td>0.467551</td>
<td>0.846054</td>
</tr>
<tr>
<td>52</td>
<td>1.348792</td>
<td>0.670245</td>
<td>1.104103</td>
<td>1.099537</td>
<td>1.218542</td>
<td>1.765145</td>
</tr>
<tr>
<td>208</td>
<td>0.870844</td>
<td>0.688555</td>
<td>1.342041</td>
<td>0.798382</td>
<td>1.424465</td>
<td>1.845571</td>
</tr>
<tr>
<td>832</td>
<td>0.434157</td>
<td>0.643675</td>
<td>1.292572</td>
<td>0.437632</td>
<td>1.357587</td>
<td>1.782223</td>
</tr>
<tr>
<td>19</td>
<td>0.798441</td>
<td>0.686146</td>
<td>0.584949</td>
<td>0.645531</td>
<td>0.712026</td>
<td>1.257433</td>
</tr>
<tr>
<td>76</td>
<td>1.336847</td>
<td>0.677691</td>
<td>1.238231</td>
<td>1.165567</td>
<td>1.311395</td>
<td>1.77091</td>
</tr>
<tr>
<td>304</td>
<td>0.606693</td>
<td>0.689471</td>
<td>1.391629</td>
<td>0.659978</td>
<td>1.445592</td>
<td>1.847582</td>
</tr>
<tr>
<td>1216</td>
<td>0.372623</td>
<td>0.64222</td>
<td>1.302891</td>
<td>0.403082</td>
<td>1.331743</td>
<td>1.756203</td>
</tr>
</tbody>
</table>

Table 2: GFLOPS for various sizes (adi)

<table>
<thead>
<tr>
<th>Size</th>
<th>GCC-Normal (GFLOPS)</th>
<th>GCC-PluTo (GFLOPS)</th>
<th>GCC-PTile (GFLOPS)</th>
<th>ICC-Normal (GFLOPS)</th>
<th>ICC-PluTo (GFLOPS)</th>
<th>ICC-PTile (GFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0.36183</td>
<td>0.365119</td>
<td>0.357889</td>
<td>0.334846</td>
<td>0.349043</td>
<td>0.354981</td>
</tr>
<tr>
<td>64</td>
<td>0.355186</td>
<td>0.306565</td>
<td>0.31276</td>
<td>0.355684</td>
<td>0.334203</td>
<td>0.320887</td>
</tr>
<tr>
<td>256</td>
<td>0.45971</td>
<td>0.364076</td>
<td>0.321977</td>
<td>0.500863</td>
<td>0.384547</td>
<td>0.381384</td>
</tr>
<tr>
<td>1024</td>
<td>0.51155</td>
<td>0.252862</td>
<td>0.226149</td>
<td>0.372411</td>
<td>0.311389</td>
<td>0.256274</td>
</tr>
<tr>
<td>13</td>
<td>0.357312</td>
<td>0.365487</td>
<td>0.357447</td>
<td>0.334028</td>
<td>0.339947</td>
<td>0.360595</td>
</tr>
<tr>
<td>52</td>
<td>0.355151</td>
<td>0.317022</td>
<td>0.308874</td>
<td>0.351451</td>
<td>0.339485</td>
<td>0.329164</td>
</tr>
</tbody>
</table>
4.3 Case 2: Better vectorization happens with PTile than with no tiling

This happens in the following cases

1) atax
2) gesummv
3) bicg

This behavior is unexpected because generally tiling has a negative impact on the percentage of vectorization. In all the three cases, the vectorization performed on code tiled with PLuTo is either zero or very close to zero. One key observation in all the three cases is that, the performance of GCC-Normal (GCC without tiling) is extremely good.

Table 3: GFLOPS for various sizes (atax)

<table>
<thead>
<tr>
<th>Size</th>
<th>GCC-Normal (GFLOPS)</th>
<th>GCC-PLuTo (GFLOPS)</th>
<th>GCC-PTile (GFLOPS)</th>
<th>ICC-Normal (GFLOPS)</th>
<th>ICC-PLuTo (GFLOPS)</th>
<th>ICC-PTile (GFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0.335073</td>
<td>0.381996</td>
<td>0.418908</td>
<td>0.407164</td>
<td>0.320086</td>
<td>0.336108</td>
</tr>
<tr>
<td>64</td>
<td>1.125085</td>
<td>0.787351</td>
<td>0.920598</td>
<td>1.088996</td>
<td>0.85949</td>
<td>0.705322</td>
</tr>
<tr>
<td>256</td>
<td>1.126871</td>
<td>0.765066</td>
<td>0.951544</td>
<td>1.432185</td>
<td>0.906023</td>
<td>0.825712</td>
</tr>
<tr>
<td>1024</td>
<td>1.079841</td>
<td>0.473954</td>
<td>0.650866</td>
<td>1.348436</td>
<td>0.632319</td>
<td>0.611452</td>
</tr>
<tr>
<td>13</td>
<td>0.192887</td>
<td>0.244271</td>
<td>0.338212</td>
<td>0.254309</td>
<td>0.226615</td>
<td>0.25888</td>
</tr>
<tr>
<td>52</td>
<td>0.895328</td>
<td>0.701147</td>
<td>0.895652</td>
<td>1.037561</td>
<td>0.81625</td>
<td>0.722168</td>
</tr>
<tr>
<td>208</td>
<td>1.148627</td>
<td>0.766018</td>
<td>1.020158</td>
<td>1.392231</td>
<td>0.978901</td>
<td>0.90359</td>
</tr>
</tbody>
</table>
Table 4: GFLOPS for various sizes (gesummv)

<table>
<thead>
<tr>
<th>Size</th>
<th>GCC-Normal</th>
<th>GCC-PLuTo</th>
<th>GCC-PTile</th>
<th>ICC-Normal</th>
<th>ICC-PLuTo</th>
<th>ICC-PTile</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0.406575</td>
<td>0.325029</td>
<td>0.366177</td>
<td>0.303748</td>
<td>0.258908</td>
<td>0.1913</td>
</tr>
<tr>
<td>64</td>
<td>0.745552</td>
<td>0.492144</td>
<td>0.690205</td>
<td>0.716596</td>
<td>0.678028</td>
<td>0.470068</td>
</tr>
<tr>
<td>256</td>
<td>0.864806</td>
<td>0.535366</td>
<td>0.842503</td>
<td>0.932479</td>
<td>0.838558</td>
<td>0.538021</td>
</tr>
<tr>
<td>1024</td>
<td>0.981653</td>
<td>0.362937</td>
<td>0.541064</td>
<td>0.797253</td>
<td>0.494429</td>
<td>0.345356</td>
</tr>
<tr>
<td>13</td>
<td>0.327389</td>
<td>0.233373</td>
<td>0.266968</td>
<td>0.244183</td>
<td>0.175441</td>
<td>0.206532</td>
</tr>
<tr>
<td>52</td>
<td>0.737008</td>
<td>0.436576</td>
<td>0.687337</td>
<td>0.643746</td>
<td>0.588484</td>
<td>0.404544</td>
</tr>
<tr>
<td>208</td>
<td>0.925543</td>
<td>0.520582</td>
<td>0.829006</td>
<td>0.852581</td>
<td>0.76622</td>
<td>0.488859</td>
</tr>
<tr>
<td>832</td>
<td>0.954058</td>
<td>0.401246</td>
<td>0.531233</td>
<td>0.957315</td>
<td>0.501221</td>
<td>0.381599</td>
</tr>
<tr>
<td>19</td>
<td>0.411764</td>
<td>0.354355</td>
<td>0.403428</td>
<td>0.372225</td>
<td>0.314159</td>
<td>0.335307</td>
</tr>
<tr>
<td>76</td>
<td>0.801376</td>
<td>0.488782</td>
<td>0.681026</td>
<td>0.744188</td>
<td>0.66304</td>
<td>0.502703</td>
</tr>
<tr>
<td>304</td>
<td>0.923318</td>
<td>0.431802</td>
<td>0.529685</td>
<td>0.821891</td>
<td>0.51892</td>
<td>0.450695</td>
</tr>
<tr>
<td>1216</td>
<td>0.875237</td>
<td>0.409251</td>
<td>0.545008</td>
<td>0.839803</td>
<td>0.517503</td>
<td>0.395947</td>
</tr>
</tbody>
</table>

Table 5: GFLOPS for various sizes (bicg)

<table>
<thead>
<tr>
<th>Size</th>
<th>GCC-Normal</th>
<th>GCC-PLuTo</th>
<th>GCC-PTile</th>
<th>ICC-Normal</th>
<th>ICC-PLuTo</th>
<th>ICC-PTile</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0.328636</td>
<td>0.411823</td>
<td>0.391014</td>
<td>0.441716</td>
<td>0.310503</td>
<td>0.239726</td>
</tr>
<tr>
<td>64</td>
<td>0.877099</td>
<td>0.791182</td>
<td>0.904913</td>
<td>1.02403</td>
<td>0.883026</td>
<td>0.694507</td>
</tr>
<tr>
<td>256</td>
<td>0.994878</td>
<td>0.71771</td>
<td>0.871719</td>
<td>1.034726</td>
<td>0.85177</td>
<td>0.819018</td>
</tr>
<tr>
<td>1024</td>
<td>1.009303</td>
<td>0.548558</td>
<td>0.617369</td>
<td>1.027224</td>
<td>0.558864</td>
<td>0.646716</td>
</tr>
<tr>
<td>13</td>
<td>0.22013</td>
<td>0.318965</td>
<td>0.282035</td>
<td>0.245715</td>
<td>0.211525</td>
<td>0.231356</td>
</tr>
<tr>
<td>52</td>
<td>0.800504</td>
<td>0.724182</td>
<td>0.889507</td>
<td>0.83117</td>
<td>0.846871</td>
<td>0.6235</td>
</tr>
<tr>
<td>208</td>
<td>0.99217</td>
<td>0.803488</td>
<td>1.073849</td>
<td>0.98565</td>
<td>1.014564</td>
<td>0.806219</td>
</tr>
<tr>
<td>832</td>
<td>1.025933</td>
<td>0.665591</td>
<td>0.749833</td>
<td>1.023895</td>
<td>0.696906</td>
<td>0.670646</td>
</tr>
</tbody>
</table>
4.4 Case 3: PLuTo performs better than PTile in terms of vectorization

We have many cases that fall into this category as this behavior is not unexpected. Since PTile performs much complex tiling than PLuTo, the percentage of vectorization reduces in case of PTile. Following are the cases

1) 2mm
2) 3mm
3) Syr2k
4) Lu
5) Covariance
6) Adi
7) Jacobi-2d-imper

In case of syr2k benchmark, the percentage of vectorization with PLuTo is much better than the Normal case (without tiling). Whereas with adi benchmark the percentage of vectorization with Normal is much better than with PluTo. For all the other cases, the percentage of vectorization is almost the same with PLuTo and Normal.

In almost all the cases, we notice that the performance of ICC-PTile is much better than others, even though the percentage of vectorization with PTile is less than PLuTo.
Table 6: GFLOPS for various sizes (3mm)

<table>
<thead>
<tr>
<th>Size</th>
<th>GCC-Normal (GFLOPS)</th>
<th>GCC-PLuTo (GFLOPS)</th>
<th>GCC-PTile (GFLOPS)</th>
<th>ICC-Normal (GFLOPS)</th>
<th>ICC-PLuTo (GFLOPS)</th>
<th>ICC-PTile (GFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1.848463</td>
<td>1.61015</td>
<td>0.966691</td>
<td>0.745999</td>
<td>0.67284805</td>
<td>1.620352</td>
</tr>
<tr>
<td>64</td>
<td>1.631485</td>
<td>0.843201</td>
<td>1.321233</td>
<td>1.475285</td>
<td>1.207464859</td>
<td>1.991657</td>
</tr>
<tr>
<td>256</td>
<td>0.678357</td>
<td>0.717354</td>
<td>1.061339</td>
<td>0.872076</td>
<td>1.013302945</td>
<td>1.733263</td>
</tr>
<tr>
<td>1024</td>
<td>0.293956</td>
<td>0.626702</td>
<td>0.835154</td>
<td>0.555497</td>
<td>0.89310211</td>
<td>1.508179</td>
</tr>
<tr>
<td>13</td>
<td>1.136688</td>
<td>1.217502</td>
<td>0.796795</td>
<td>0.659033</td>
<td>0.61589258</td>
<td>1.24779</td>
</tr>
<tr>
<td>52</td>
<td>1.639034</td>
<td>0.862677</td>
<td>1.296739</td>
<td>1.525896</td>
<td>1.14021601</td>
<td>1.91239</td>
</tr>
<tr>
<td>208</td>
<td>1.151025</td>
<td>0.802048</td>
<td>1.374486</td>
<td>1.282329</td>
<td>1.163530075</td>
<td>1.960616</td>
</tr>
<tr>
<td>832</td>
<td>0.480463</td>
<td>0.756462</td>
<td>1.325669</td>
<td>1.323534</td>
<td>1.15481225</td>
<td>1.936345</td>
</tr>
<tr>
<td>19</td>
<td>1.372561</td>
<td>1.238925</td>
<td>0.99875</td>
<td>1.001372</td>
<td>0.76515738</td>
<td>1.668282</td>
</tr>
<tr>
<td>76</td>
<td>1.521771</td>
<td>0.865075</td>
<td>1.340909</td>
<td>1.357427</td>
<td>1.128026465</td>
<td>1.951445</td>
</tr>
<tr>
<td>304</td>
<td>0.89787</td>
<td>0.791931</td>
<td>1.384226</td>
<td>1.083563</td>
<td>1.176268664</td>
<td>1.986049</td>
</tr>
<tr>
<td>1216</td>
<td>0.521411</td>
<td>0.759203</td>
<td>1.309262</td>
<td>1.295025</td>
<td>1.144462114</td>
<td>1.933059</td>
</tr>
</tbody>
</table>

4.5 Case 4: PTile performs better than PLuTo in terms of vectorization

We have less number of cases in this category as this behavior is unexpected. As already mentioned, PTile makes more complex loops than PLuTo, hence the vectorization percentage is expected to be less in case of PTile, but that does not happen in these four cases. The following are the cases that fall into this category

1) atax
2) gemver
3) gesummv
4) bicg

The performance of GCC-Normal (no tiling) is the best in all the four cases.
4.6 Case 5: Tiling does not affect the percentage vectorization

These are the cases where the three lines (Normal, PLuTo, PTile) of the percentage vectorization graph overlap almost perfectly.

1) Cholskey
2) Doitgen
3) Gemm
4) Symm
5) Durbin
6) Gramschmidt
7) Ludcmp

So, in all these cases, tiling does not affect the vectorization. If we look at the performance graphs of these benchmarks, we observe that all the lines almost overlap in case of cholskey and symm, but that does not happen with the other benchmarks.

Table 7: GFLOPS for various sizes (symm)

<table>
<thead>
<tr>
<th>Size</th>
<th>GCC- Normal</th>
<th>GCC- PLuTo</th>
<th>GCC- PTile</th>
<th>ICC- Normal</th>
<th>ICC- PLuTo</th>
<th>ICC- PTile</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1.24261</td>
<td>1.458593</td>
<td>1.272515</td>
<td>1.017144</td>
<td>1.303205</td>
<td>1.093425</td>
</tr>
<tr>
<td>64</td>
<td>1.8949</td>
<td>1.920666</td>
<td>1.903473</td>
<td>1.275924</td>
<td>1.311435</td>
<td>1.248527</td>
</tr>
<tr>
<td>256</td>
<td>0.482211</td>
<td>0.660703</td>
<td>0.673785</td>
<td>0.492318</td>
<td>0.526298</td>
<td>0.45292</td>
</tr>
<tr>
<td>1024</td>
<td>0.272772</td>
<td>0.361276</td>
<td>0.287879</td>
<td>0.227319</td>
<td>0.229117</td>
<td>0.158494</td>
</tr>
<tr>
<td>13</td>
<td>0.87022</td>
<td>1.244734</td>
<td>1.056579</td>
<td>0.874448</td>
<td>0.938431</td>
<td>0.904602</td>
</tr>
<tr>
<td>52</td>
<td>2.038678</td>
<td>2.155595</td>
<td>2.155819</td>
<td>1.354155</td>
<td>1.442865</td>
<td>1.402566</td>
</tr>
<tr>
<td>208</td>
<td>1.000407</td>
<td>1.145189</td>
<td>1.142075</td>
<td>0.895772</td>
<td>0.902066</td>
<td>0.874049</td>
</tr>
<tr>
<td>832</td>
<td>0.370465</td>
<td>0.729689</td>
<td>0.300275</td>
<td>0.636835</td>
<td>0.618698</td>
<td>0.442167</td>
</tr>
<tr>
<td>19</td>
<td>1.307345</td>
<td>1.522581</td>
<td>1.521166</td>
<td>1.10473</td>
<td>1.326703</td>
<td>1.069844</td>
</tr>
</tbody>
</table>
4.7 Conclusion

Looking at the key observations we can conclude that the percentage of vectorization with PTile is generally less when compared with PLuTo. PTile creates more complex loops than PLuTo, hence the compiler heuristics fail in performing proper vectorization. It was also concluded that this may not happen in all the cases as a few cases were noticed with better vectorization with PTile compared to the Normal (no tiling) case, which is unexpected. We also observed many cases where tiling did not affect the percentage vectorization which proves that tiling optimization can positively affect the overall performance, by not affecting the percentage vectorization.
Bibliography


