Factors Affecting the Performance of Tiled Numerical Codes

Author: Besan A. Radwan

Advisor: Professor David G. Wonnacott

May 7th, 2014

A thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in the Department of Computer Science at Haverford College

Contents

1 Part 0: Abstract **4**

2 Part I: Background-What is the problem we are investigating? **5**

2.1 Data Dependence Overview 6

2.2 Data Dependence Analysis for Arrays 9

2.3 The Basics of Tiling 10

2.4 Investigating different shaped tiles: Diamonds (fully oblique) and Parallelogram shaped tiles and Tile Size 15

2.5 Investigating the Structure of the original code and memory access: Handling loads, stores and pointers 18

2.6 Software tools used in code optimization: ISCC 18

2.7 Intra-tile order 20

2.8 Multiple cores vs Single core Machines 20

3 Part II: Experiments and Results **21**

3.1 Experimental Methods 21

3.2 Experimental Results in Tabular Format 22

3.3 Experimental results in graphical format 26

4 Part III: Discussion of Results **43**

4.1 Single core for Heat-1d 43

4.2 Four core for Heat-1d 44

4.3 Single core for Heat-1d-twocalc 45

4.4 Four core for Heat-1d-twocalc 46
4.5 Heat-1d vs. Heat-1d-twocalc single core 47
4.6 Heat-1d vs. Heat-1d-twocalc four core 47
4.7 Ratio of single core to four core runs for Heat-1d 48
4.8 Ratio of single core to four core runs for Heat-1d-twocalc . . . 49

5 Conclusion **50**

5.1 Conclusions on four-core runs 50
5.2 Conclusions on single-core runs 52
5.3 Summary 54

6 Future Directions **55**

1 Part 0: Abstract

The topic I am investigating is High Performance Computing. I am investigating the factors affecting the outcome of speed of tiled dense array programs. The significance and applications of parallel computing are countless. As the field expands, different approaches are taken to optimize performance in HPC. It is paramount to not only investigate methods for continued innovation in the field, but to look at the methods already out there and how a combination of these methods can provide promising results.

The ISCC software transformation tool was used with different tile arrangements to investigate which factors affected the speed at which programs could be run sequentially or in a parallel. In addition to this, the tile arrangements and distribution with different tile sizes were tested. The results obtained were significant and showed that using the certain combinations of tile shape and iteration performed a significant amount better than other tile arrangements.

2 Part I: Background-What is the problem we are investigating?

This chapter will outline the background information that includes the research present in the field as it pertains to our specific investigation. In chapter three we will discuss our hypotheses in relation to this information.

It is important to consider the research already present in the field before attempting to make significant changes to it. However, it may be the case that different aspects of the research already present in the field executed simultaneously can produce significant results. For this reason, there are a number of factors that have already been researched and investigated that this study will use. These factors are listed below. Additionally, there will be a number of factors that have not yet been investigated that will be included in this experiment. The combinations of the factors (both not-previously investigated and previously investigated) will be used in order to determine which combination of factors most affects speed/performance.

1. Investigating different shaped tiles: Diamonds (fully oblique) and Parallelogram shaped tiles
2. Tile Size
3. Intra-tile order: Scheduling within individual tiles
4. Investigating the Structure of the original code and memory access: Handling loads, stores and pointers
5. Multiple Cores vs. Single Core Machines

2.1 Data Dependence Overview

Data dependence is necessary to understand many of the factors that were investigated as part of the previously stated factors. A useful 'everyday' example of data dependence in the field of Computer Science is compilers. The way in which compilers optimize code is primarily through reorganizing and re-ordering the original program. There are many factors that need to be considered when re-ordering and reorganizing the original program. One of the factors necessary to consider is data dependence. There are three types of data dependence:

1. Flow Dependence
2. Anti-Dependence
3. Output Dependence

Flow dependence is described as assigning a "variable in one statement and using it later in another statement"[1]. Output dependence occurs when a "variable is assigned in one statement and then this variable is reassigned"[1]. Anti-dependence is when a "variable is used in one statement and reassigned in a subsequently executed statement"[1]. There are many ways of representing the different types of data dependence information. The main way of representing this information is through data dependence graphs. In a data dependence graph there are arrows that have a direction property. The direction of the arrow indicates which statements or variables depend on each other. The diagram below is a helpful example of data dependence statement table (this example has been adapted from Michael Wolfe's textbook[1]):

Declaration Number	Statement
1	A = 0
2	B = 5
3	D = A
4	A = B + 1
5	C = A +2

In the above diagram we see examples of the three different types of data dependence previously described. The distinction between the different types is important because different data dependencies have different restrictions on how you can re-order them. For example, the data dependence between

declaration one and five is flow dependence, as is the case between statements one and three. It is important to know because identifying two statements being dependent means that you cannot simply re-order them or run them at the same time without risk of changing the output of the program.

If it has been determined that there is no dependence; you can execute any statement in any order (i.e. concurrently). However, with some dependencies you can re-order as long as there are no changing dependencies. For example, if you can move statement two below statement three.

Flow dependencies are important because some of them communicate values. Thus even in the presence of some dependencies there are "clever" ways to re-order as long as you do not change the flow dependencies that carry a given value.

Clever reordering can be done in two ways. Using the above table as an example will show these two methods. The first method is by renaming variable A from statements one and three: (A1). This removes dependencies that do not carry values such as the flow dependence between statement one and five, thus allowing new ordering. The second method is moving statements one and three below statement five. Note that in clever reordering (such as in this example) some dependencies can change, but no dependencies in which values flow change.

It is important to note there is a relationship between the example pictured above and the research hypothesis that is being investigated. The main programs that this research is hoping to optimize are primarily loop based.

This means that the programs have loop-based dependence. Since we want to re-order or parallelize our loops, we need to be aware of dependencies among iterations of the loops. To clarify, this means we are concerned with the iterations both within the loop, and between the iterations of different loops.

2.2 Data Dependence Analysis for Arrays

In the previous section it was conveyed that compliers make use of data dependent systems and that those data dependent systems are important for thinking about reordering of statements. While a compiler relies on the idea of data dependence to eventually evaluate the re-ordering of statements in the program for efficiency, compilers also analyze iterations of loops to evaluating the dependency. Rather than create a table and evaluate the data dependency of every statement in the table, the complier solves the data dependence system by solving "a system of linear equations and inequalities"[1] where the dependencies are represented with linear equations. The complier forms an equation to determine the dependence between any of the array references. Solving this system of equations aids in understand the dependencies that are present within the array within the loops. Solving these equations is beyond the scope of this research.

2.3 The Basics of Tiling

Before the concept of tiling can be introduced, it is important to discuss what an iteration space is. If there is a loop with index `i` that goes from `1 to N`, then the space contains all the values of i for $1 \leq i \leq N$. If the iteration space is `N by N` as in Figure 1, then the space contains all the pairs i, j for $1 \leq i \leq N$ and $1 \leq j \leq N$.

Tiling for the purposes of this specific research project is the way in which we divide the iteration space into many individual pieces and reorder those pieces without the fear of over-writing data or affecting the legality of the program. This brings up the two fundamental aspects of tiling: legality and impact on performance. Legality is important because you must maintain that despite running multiple "chunks/tiles" of the data, the same result is still obtained. The same result must be obtained because it would not be useful to have a very fast program that gave you the wrong result. The impact on performance by using tiling, can be seen and understood by means of the diagram depicted below which represents an example of doing one time step of a heat calculation on a two-dimensional data-set.

```
for(i = 1; i < N-1; i++) {
  for(j = 1; j < N-1; j++) {
    Anew(i,j) = (A(i-1,j)+A(i,j-1)+A(i+1,j)+A(i,j+1)+2*A(i,j))*0.167;
```

Figure 1: Code for two-dimensional data set

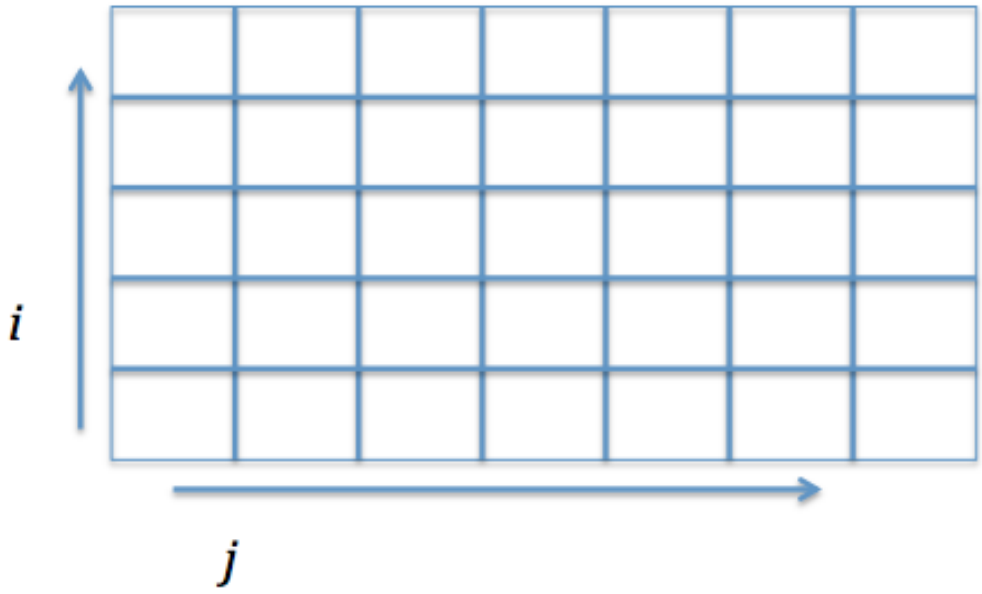

Figure 2: Iteration Space

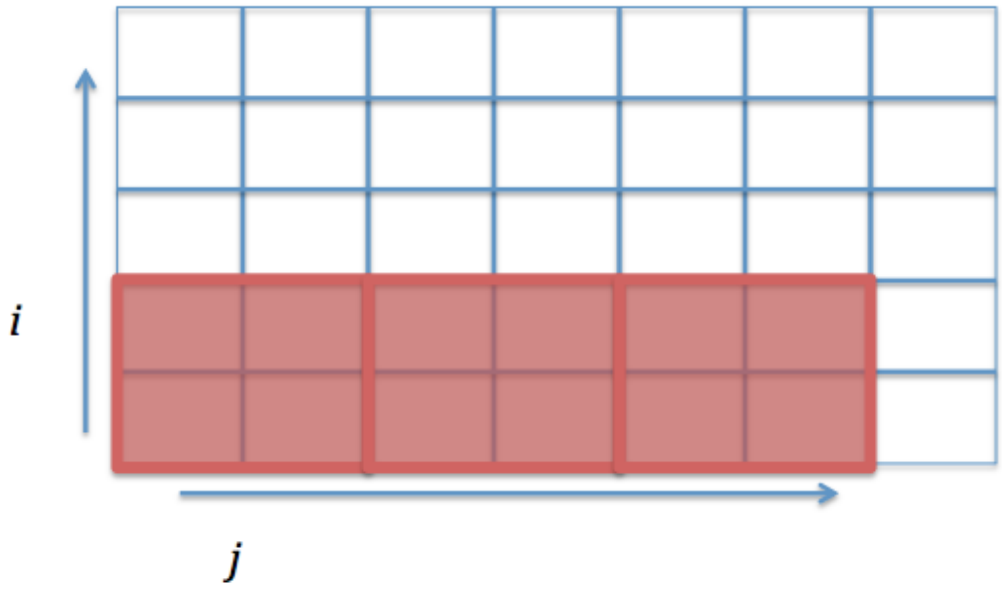

Figure 3: Iteration Space with Tiles

A basic iteration space is depicted in figure 2. In figure 3 the red squares depict tiles that were constructed. It is possible to run multiple red squares at once. This is because the values do not depend on each other since they are all in the same time-step. Seeing as multiple squares can be run is why/how tiling can improve performance. In addition, the relationship between re-ordering and parallelizing is paramount. This relationship is the ability to run multiple squares simultaneously or in either order because they are not data dependent.

Tile size is another factor that is important to understand when thinking about tiling. There are pros and cons to both having small and large tile sizes. With smaller tiles, you can use more cores, because there are more tiles, which means you can run more tiles simultaneously. However, with larger tiles, you can't run as many at once because there are less of them. The positive aspect of big tiles is that large tiles have more data points that do not have neighbors with data on other cores. This is important because the fraction of neighboring data points that have to ask another core for information is less as the tile size gets larger.

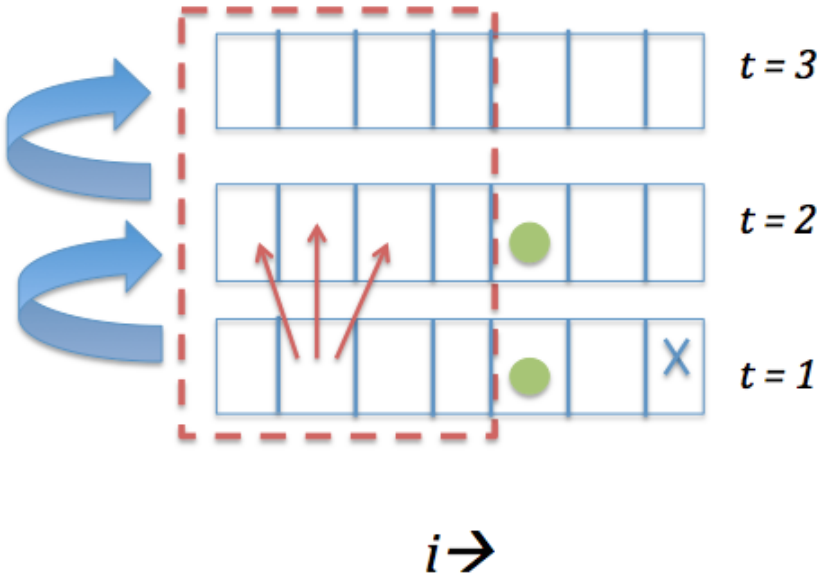

Figure 4: Illegal Iteration Space with Tiles

```
for(t = 1; t <= T; t++) {
    for(i = 1; i < N-1; i++) {
      A2(t,i) = (A2(t-1,i-1)+2*A2(t-1,i)+A2(t-1,i+1))*0.25;
    }
```

Figure 5: Code for Figure four-known as Heat-1d

An example of illegal tiling is depicted In figure 4. This is to convey the point that the tiling must be legal otherwise the program will produce a different output or result. In the above diagram, "t" indicates time steps. We are seeing a dotted red line indicated where the user is trying to create a tile. This is illegal because in the original program if you attempted to get the value corresponding to the circle at t = 2, you would instead get the green circle with the square at t = 1.

2.4 Investigating different shaped tiles: Diamonds (fully oblique) and Parallelogram shaped tiles and Tile Size

A basic approach to tiling using squares was outlined in the previous section. However, tiling cannot be done with rectangles in some cases as we saw previously. Additionally, the square shaped tiles do not always produce the best results in terms of performance. The different shaped tiles that will be introduced in this section are the main tile shapes that will be used in the experiments being run for the research investigation.

The main two types of tiling are fully oblique tiling (otherwise known as diamond tiling) and pipelined (otherwise known as parallelogram tiling). It is useful to explain these two types of tiling as they relate it to the paper *Tiling Stencil Computations to Maximize Parallelism*[2]. There are two parts to understanding tiling: tile shape and tile size. Tile shape is the "di-

rection [in which you choose] to slice the iteration space"[2]. Tile size is how large the slices of the iteration space you want. Many of the experiments look to find the optimal tile shape and size paring. It is hypothesized that it could be possible that the fully oblique tiles when paired with different aspects of program manipulation can outperform pipelined tiles under the same conditions. Hence, the investigation into the aspects that are relevant for performance has potential for interesting results.

Another aspect that is important to tilling is the division of the iteration space into different shaped tiles. When the iteration space is divided into tiles is an important question to address whether or not the concurrent or pipelined startup is superior. These two types of startup are based on the way the tile is sliced. If the tiles are diamond shaped, the first set of tiles can be run in parallel. Whereas with parallelogram shaped tiles you cannot run the first set of tiles in parallel since data flows among them. The images on the next page that are taken from Bandishti's paper[2] will illustrate this further:

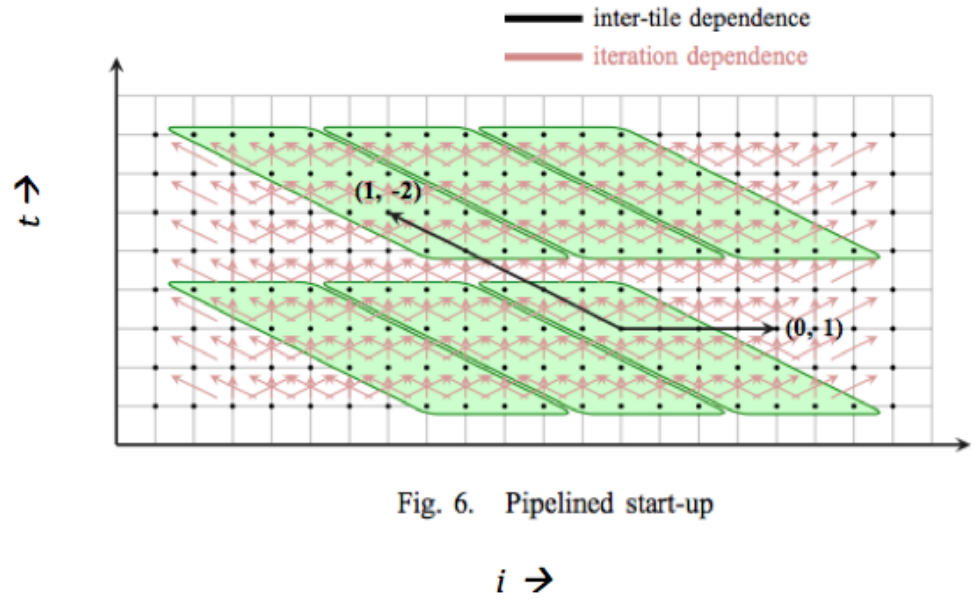

Fig. 6. Pipelined start-up

Figure 6: Pipelined Startup[2]

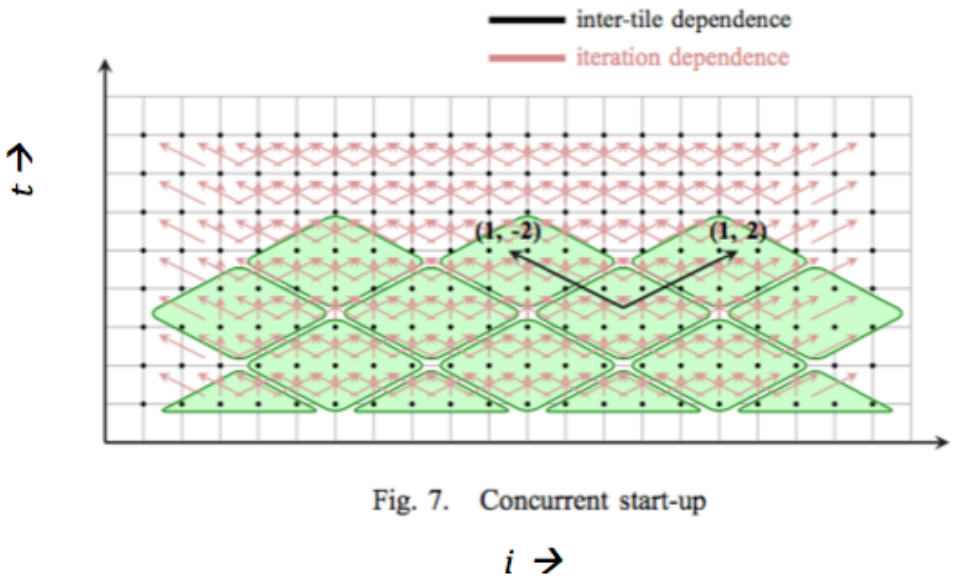

Fig. 7. Concurrent start-up

Figure 7: Concurrent Startup[2]

2.5 Investigating the Structure of the original code and memory access: Handling loads, stores and pointers

The main program that was used to execute the experiments was the "heated-wire" program. This program models the flow of heat on a one-dimensional wire as previously specified in the heat flow equations. There are three variations of this program: Heat-1d, Heat-1d-twocalc, and Heat-1d-withcopy (all located in the Appendix). These three variations differ in how they access the memory. The experiments that were run were in three sets, where each set was a different variation of the original heated wire program

2.6 Software tools used in code optimization: ISCC

The software tool ISCC is an essential piece for investigating the various aspects of performance. This is an "interactive interface to the barvinok counting library as well as providing an interface to the CLooG code generation library, to the pet Polyhedral model extractor and to some operations of the isl integer set library. [It is inspired] by Omega Calculator from the Omega Project"[6].

While ISCC was ultimately used for the optimizations pertaining to the research hypothesis, it is important to mention other software tools that were considered, but ultimately decided against. Each software tool listed (including ISCC) that was considered has its individual drawbacks. The

software tools considered were:

1. Pluto
2. AlphaZ
3. ISCC

Before explaining these three software tools and their pros and cons, it is important to note the basis that the present day tools for parallelizing software tools. The tools that were the origin of later more sophisticated tools focused on determining data dependence and are called The Omega Test and polylib. The paper that discusses The Omega Test is *The Omega Test: a fast and practical integer programming algorithm for dependence analysis*[4]. The Omega test determines dependence between array references. These tests use the basis discussed in Wolfe's text, i.e. determining "if there is an integer solution to an arbitrary set of linear equalities and inequalities"[4]. These tests developed over years and was adopted/adapted by different software tools to investigate other ways of applying this dependency check to different parallel situations.

Now that the basis for the three tools listed above has been determined, it is important to give a brief list of the pros and cons for each of the tools. The software tool Pluto has the ability to run concurrent startup tiles, which is useful for our purposes and goals, however it does not have manual control of the programs we write, which provides less flexibility. AlphaZ would be the optimal choice because it can check correctness and perform storage mapping. However, AlphaZ does not support use of concurrent startup tiles. For this

reason, we settle on using ISCC, which allows concurrent and pipelined tiling as well as manual control.

2.7 Intra-tile order

Previous work on tiling has not explored order of iterations within a tile, but we wish to explore intra-tile iterations (called the ti version of the program) to see how significant of a difference it makes compared to the factors that have received more study in the past.

2.8 Multiple cores vs Single core Machines

For multiple cores vs single core machines, the investigation seeks to look at the difference that parallelizing the code and running it on different cores can make. Also, it investigates aspects such as whether reordering the loop and parallelization has an overall positive net effect on multiple cores and how much of a difference this is compared to the single core version. Also, the investigation examines whether the same factors that matter on one-core are still important on more than one core.This indicates how it will perform on supercomputers that have many more cores.

3 Part II: Experiments and Results

3.1 Experimental Methods

A series of factors were hypothesized to affect the performance of running a program. It was determined which combination of factors would have the greatest effect on the speed of running the program through empirical testing.

The experiments were structured so that certain variables could change while keeping others constant. This helped show which factors were the ones actually affecting the speed. The variables we are concerned with are listed as follows:

1. Tile Size represented as TAU/SIGMA
2. Number of Time steps (T)
3. Tile Shape (Pipelined/Diamond)
4. Tile Distribution (Single core or OMP-i.e 4 cores)
5. Intra-tile ordering (ti)
6. Structure of the original code and memory access: Handling loads, stores and pointers

Each of the above variables was investigated by setting up a series of experiments. Each variable was matched with every other variable in an attempt to determine which combination of variables most affects the speed at which the program runs. For example, one experimental run could be the

fully oblique tiles paired with the ti iteration layout, run on a single core machine. This can be compared to the same experimental run without the ti iteration layout to see which factors affect speed in a particular environment.

Within each group a variation of the variables mentioned at the beginning of the section are alternated to attempt to test every possible combination of the variables. This is done to determine which combination of factors has the greatest effect on the speed of the program

3.2 Experimental Results in Tabular Format

The table of results for Heat-1d with 25,000 time steps with variations on the pipelined tile arrangement is below:

TAU/SIGMA	Original (MF)	ti (MF)	OMP (MF)	OMP-ti (MF)
32	2169.03	2217.44	8453.47	8801.5
64	2531.13	2545.91	9872	10127.9
1250	3034.59	3022.34	10009.8	10022.9
1360	3038.15	3042.84	9202.18	9208.91
1365	2977.29	3038.43	9152.94	9160.39
1907	2977.29	2973.69	9870.88	9869.79
2720	2908.69	2911.59	6586.28	6593.02
4096	2902.87	2903.75	8721.68	8721.92
5460	2896.97	2902.79	6552.23	6559.31
8192	2898.09	2895.18	4372.92	4380.68

The table of results for Heat-1d with 25,000 time steps with variations on the fully oblique tile arrangement is below:

TAU/SIGMA	Original (MF)	ti (MF)	OMP (MF)	OMP-ti (MF)
32	546.738	1375.3	2190.33	5537.38
64	568.896	1905.21	2283.93	7557.71
1250	590.793	2987.87	2330.58	10789.6
1360	590.866	2995.28	2355.66	10975.9
1365	592.374	2997.09	2155.67	10939.7
1907	588.971	3018.6	1945.77	9803.34
2720	592.371	2990.13	1959.69	8803.33
4096	591.452	2945.63	1798.56	6613.91
5460	584.415	2926.29	1326.97	8871.84
8192	583.491	2910.11	1770.59	9047.02

The table of results for Heat-1d-twocalc with 25,000 time steps with variations on the pipelined tile arrangement is below:

TAU/SIGMA	Original (MF)	ti (MF)	OMP (MF)	OMP-ti (MF)
32	2417.3	2644.15	9571.99	10447.2
64	2821.2	30278	11231.3	11853.1
1250	3606.5	3557.47	11406.4	11713.7
1360	3610.9	3561.04	10503.6	10745.2
1365	3613.3	3556.29	10863.9	10721.3
1907	3612.7	3498.01	11571.8	11356.7
2720	3369.9	3305.6	7404.36	7456.55
4096	3253	3261.96	9694.48	9736.68
5460	3365.8	3257.7	7306.78	7325.83
8192	3248.9	3252.55	4897.58	4902.43

The table of results for Heat-1d-twocalc with 25,000 time steps with variations on the fully oblique tile arrangement is below:

TAU/SIGMA	Original (MF)	ti (MF)	OMP (MF)	OMP-ti (MF)
32	705.957	1736.55	2793.52	7020.63
64	720.257	2283.8	2877.09	9062.59
1250	743.087	3508.72	2691.41	12660.9
1360	743.539	3515.96	2720.94	12882.9
1365	743.462	2191.98	2708.93	13004.6
1907	743.367	3497.78	2418.93	11698.7
2720	744.291	3457.76	2263.13	10407.2
4096	744.244	3352.37	2257.49	10053.6
5460	744.401	3311.75	1692.16	7463.41
8192	739.81	3278.45	2259.72	9881.19

3.3 Experimental results in graphical format

It is difficult to understand the results as presented so far. For this reason, the data previously represented in tabular format was transformed into graphs to better convey the meaning of the results. This graphs shows trends in speeds as well as which factor contributed the most to a higher achieved speed for running the program.

Before displaying the graphs and discussing the results it is important to state the hypotheses:

1. For large N and relatively small T, diamonds are much better than parallelograms

2. Diamond tiles would be much more sensitive to tile size that parallelograms (at least the Diamond-ti version would be)

A discussion of whether or not the hypotheses stated are true as well as an evaluation of the data will follow the graphs. The evaluation will comment on trends displayed in the graphs that may not have been previously expected.

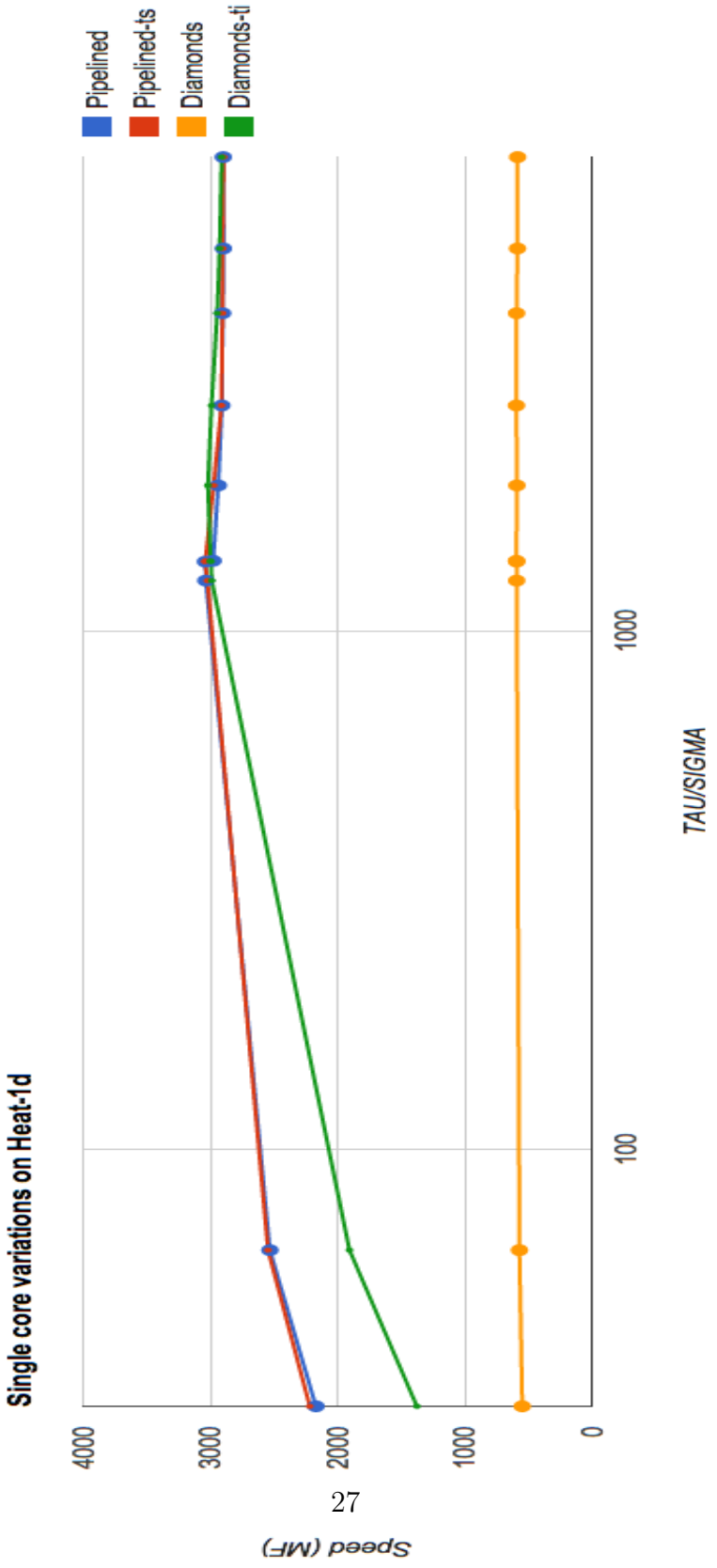

Figure 8: Heat-1d single core linespoints

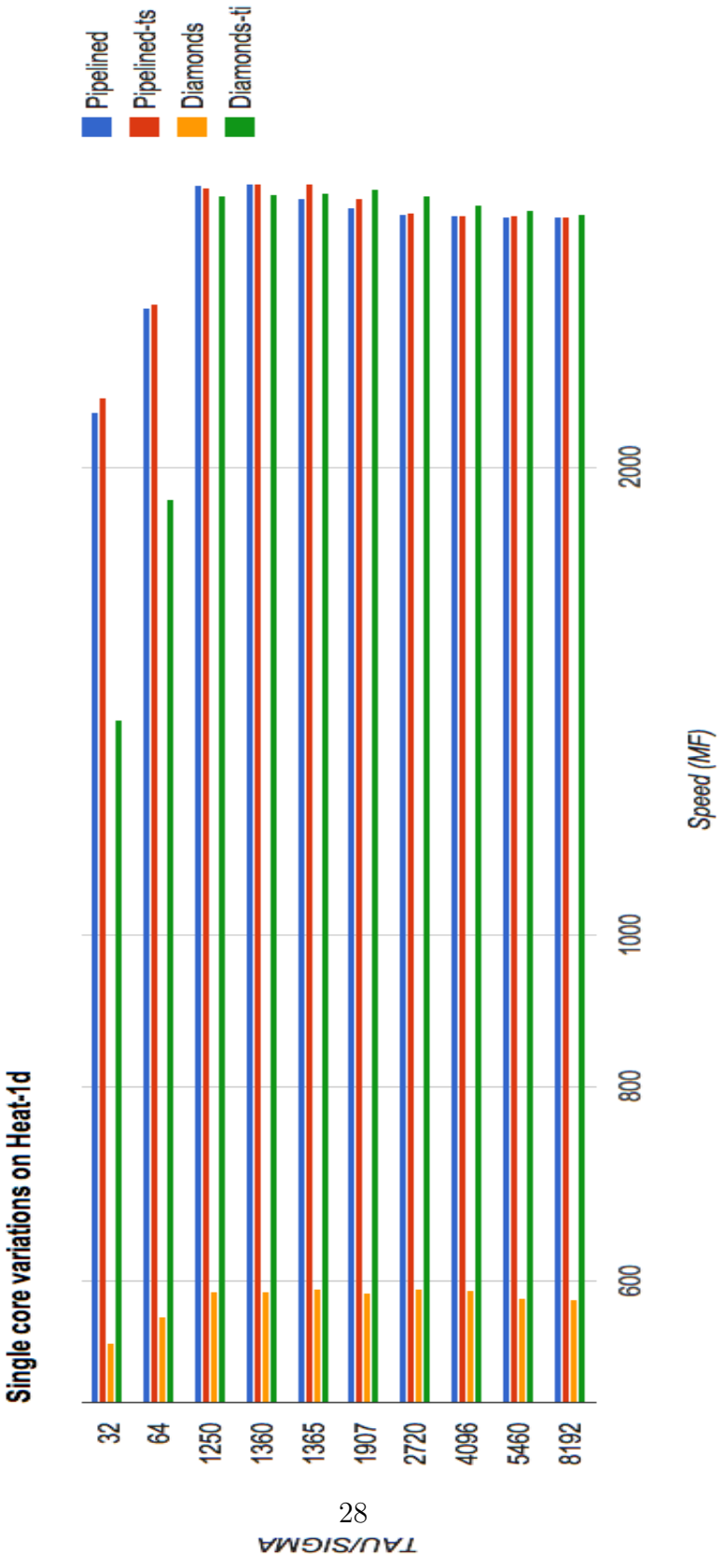

Figure 9: Heat-1d single core bar graph

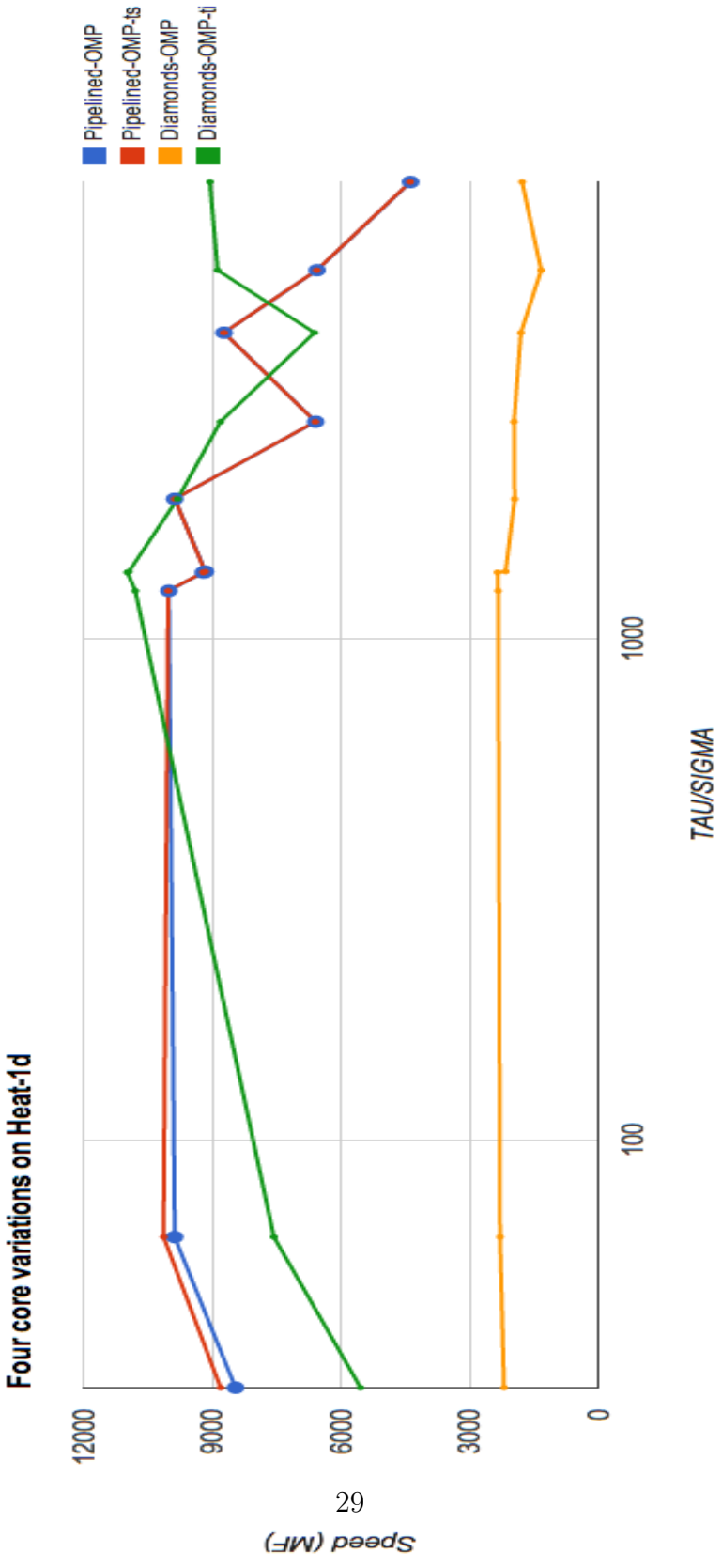

Figure 10: Heat-1d four core linespoints

Figure 11: Heat-1d four core bar graph

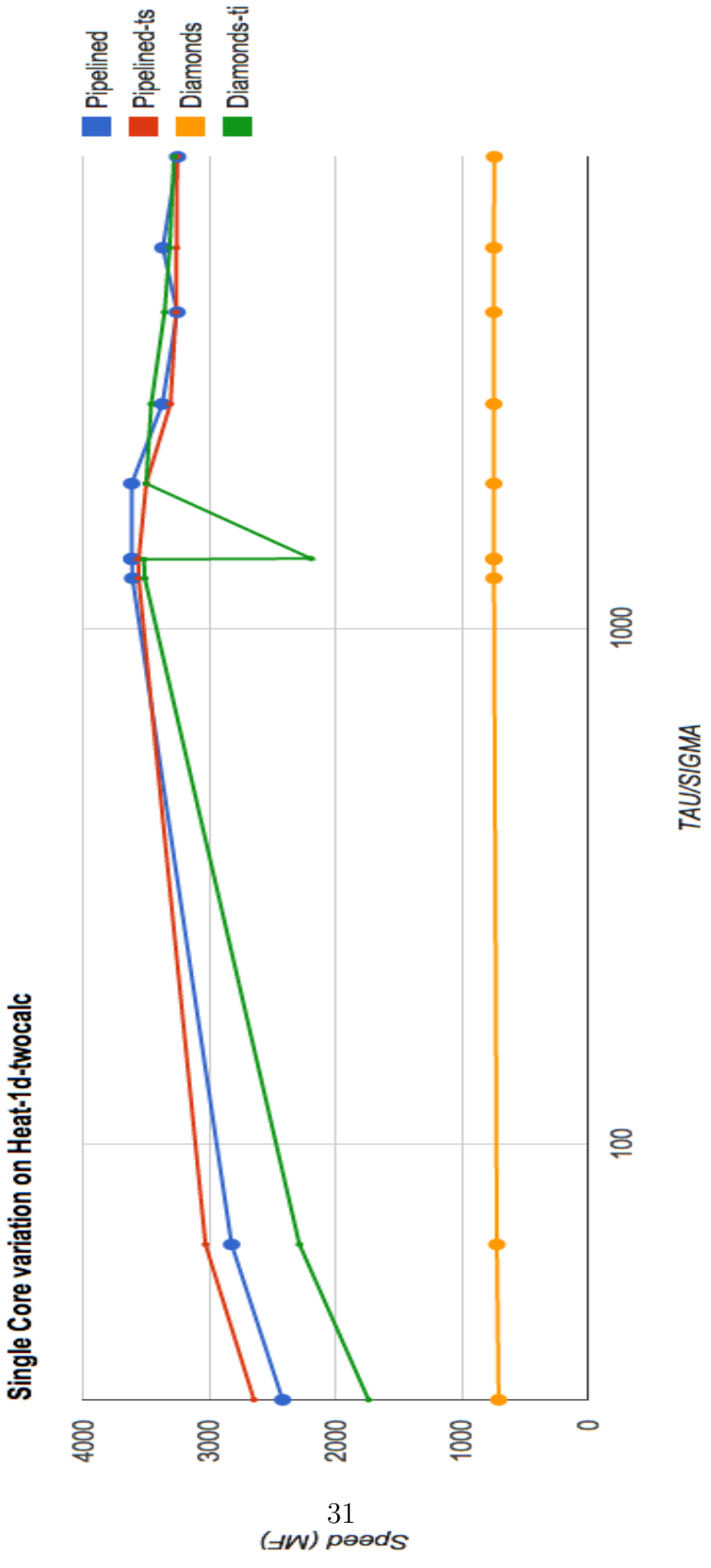

Figure 12: Heat-1d-twocalc single core linespoints

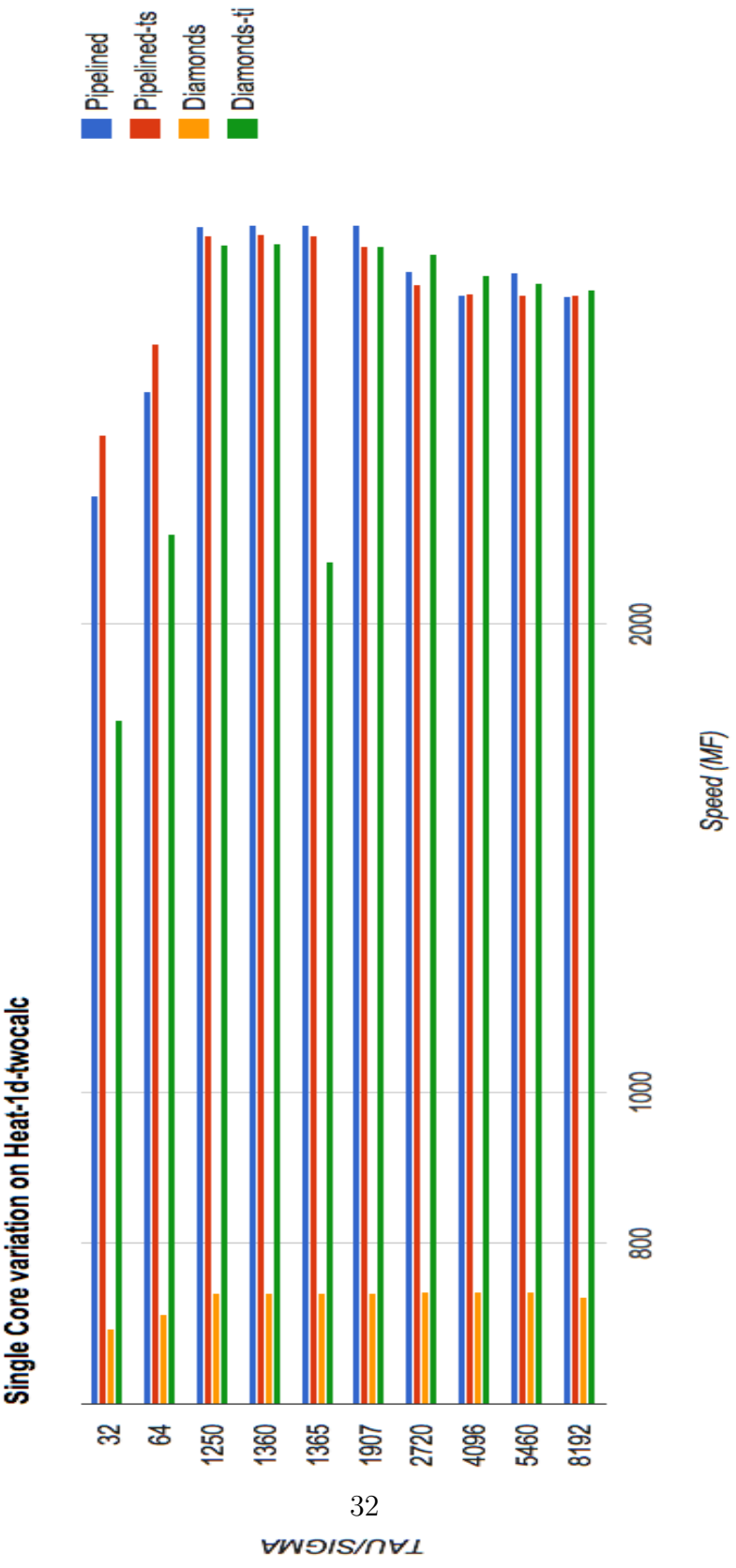

Figure 13: Heat-1d-twocalc single core bar graph

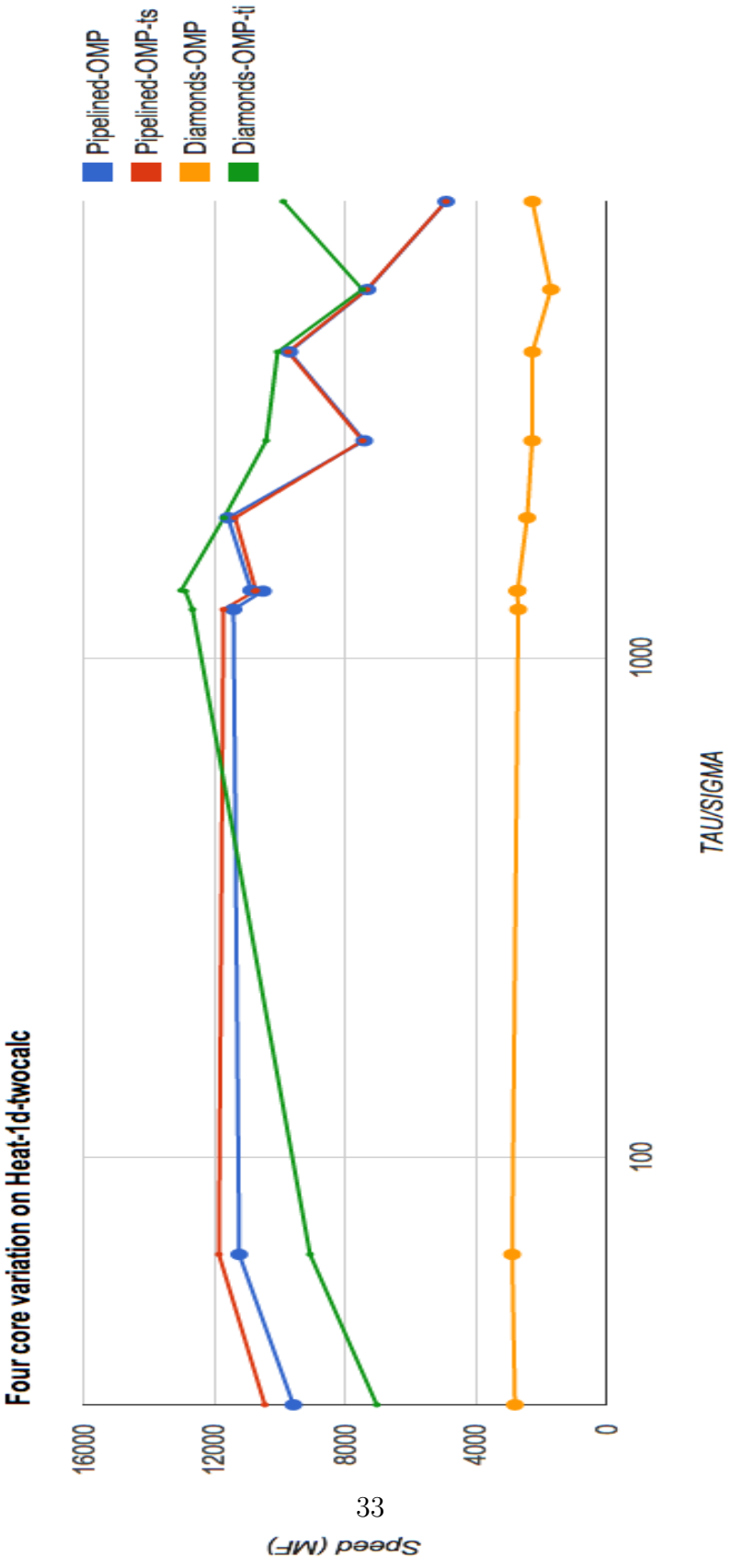

Figure 14: Heat-1d-twocalc four core linespoints

Figure 15: Heat-1d-twocalc four core bar graph

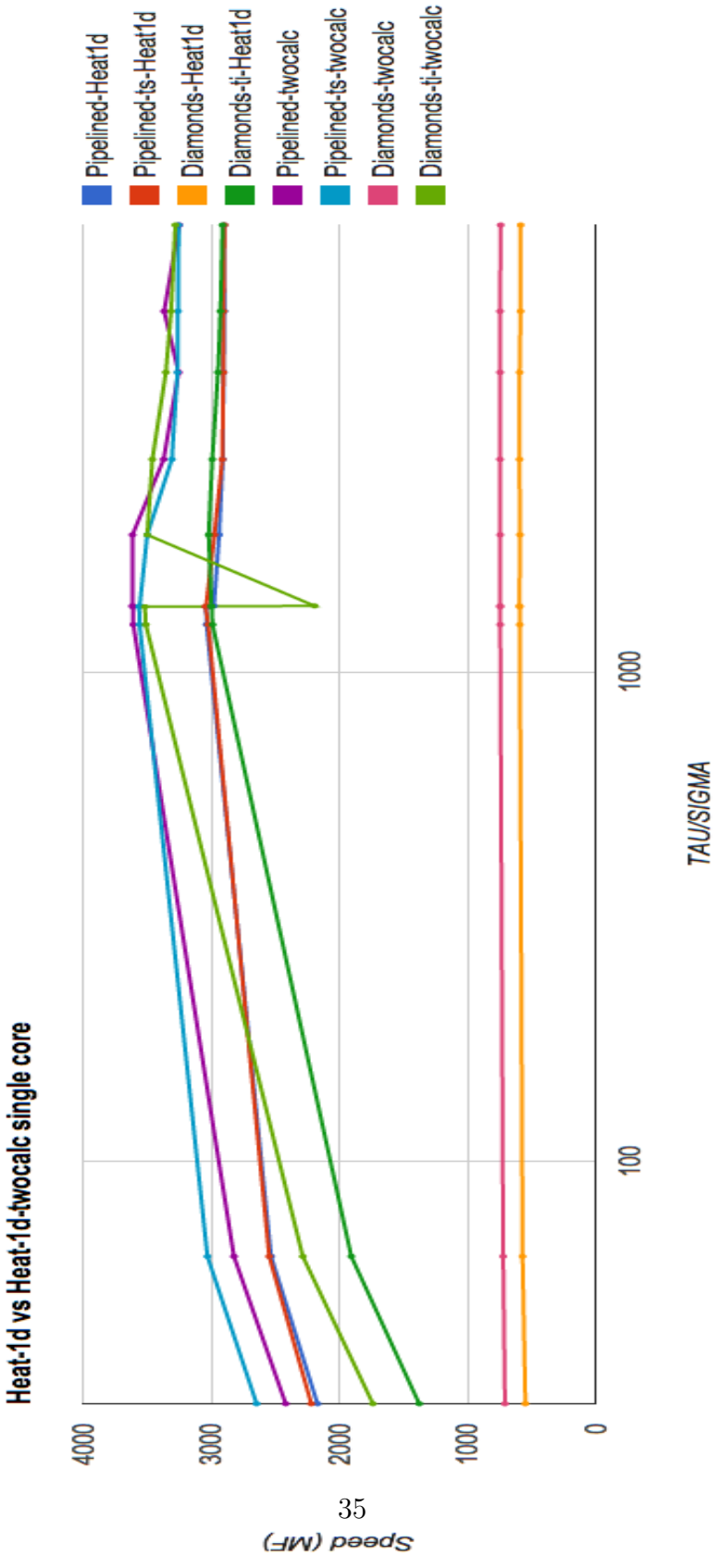

Figure 16: Heat-1d vs Heat-1d-twocalc single core linespoints

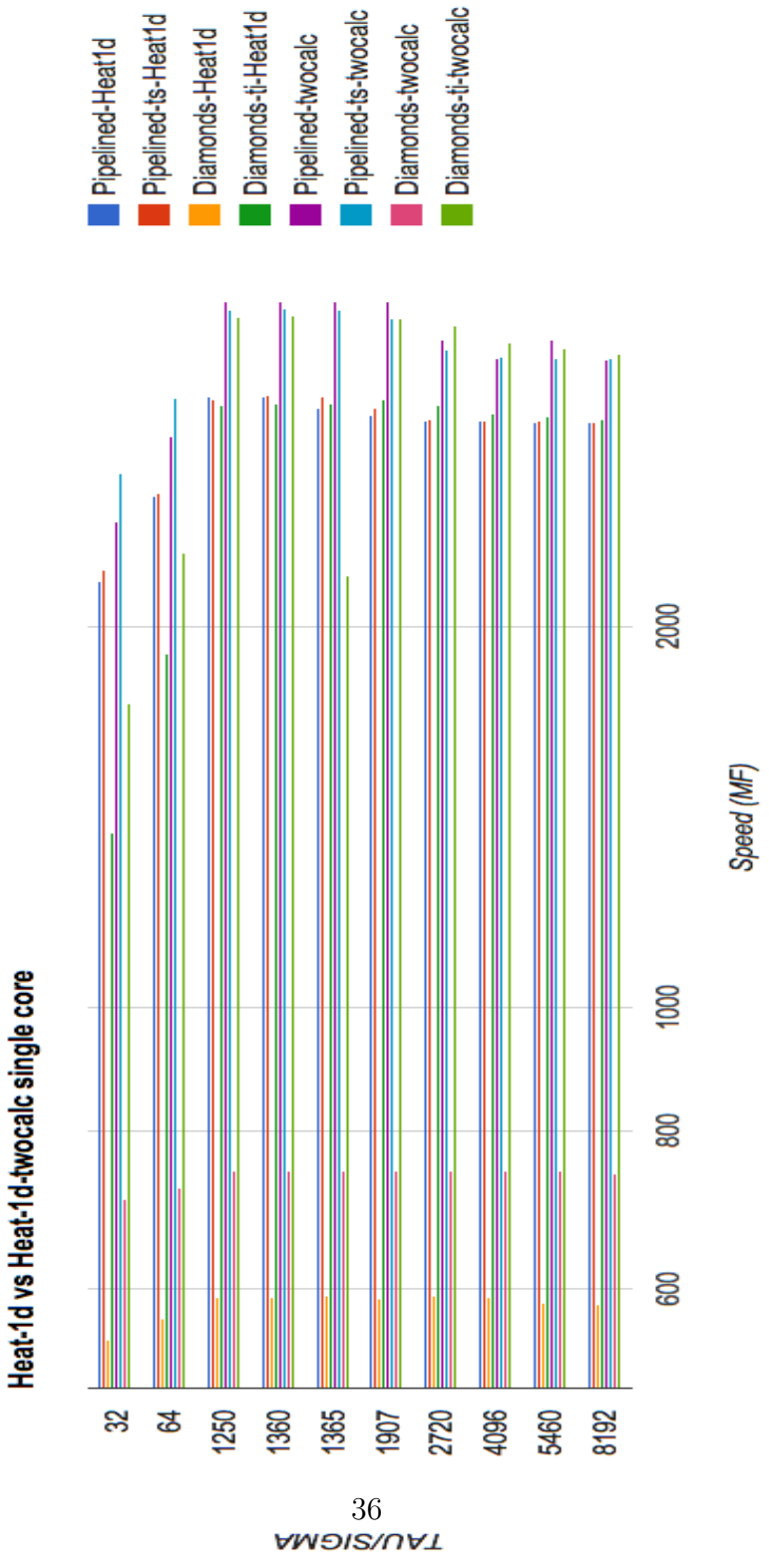

Figure 17: Heat-1d vs Heat-1d-twocalc single core bar graph

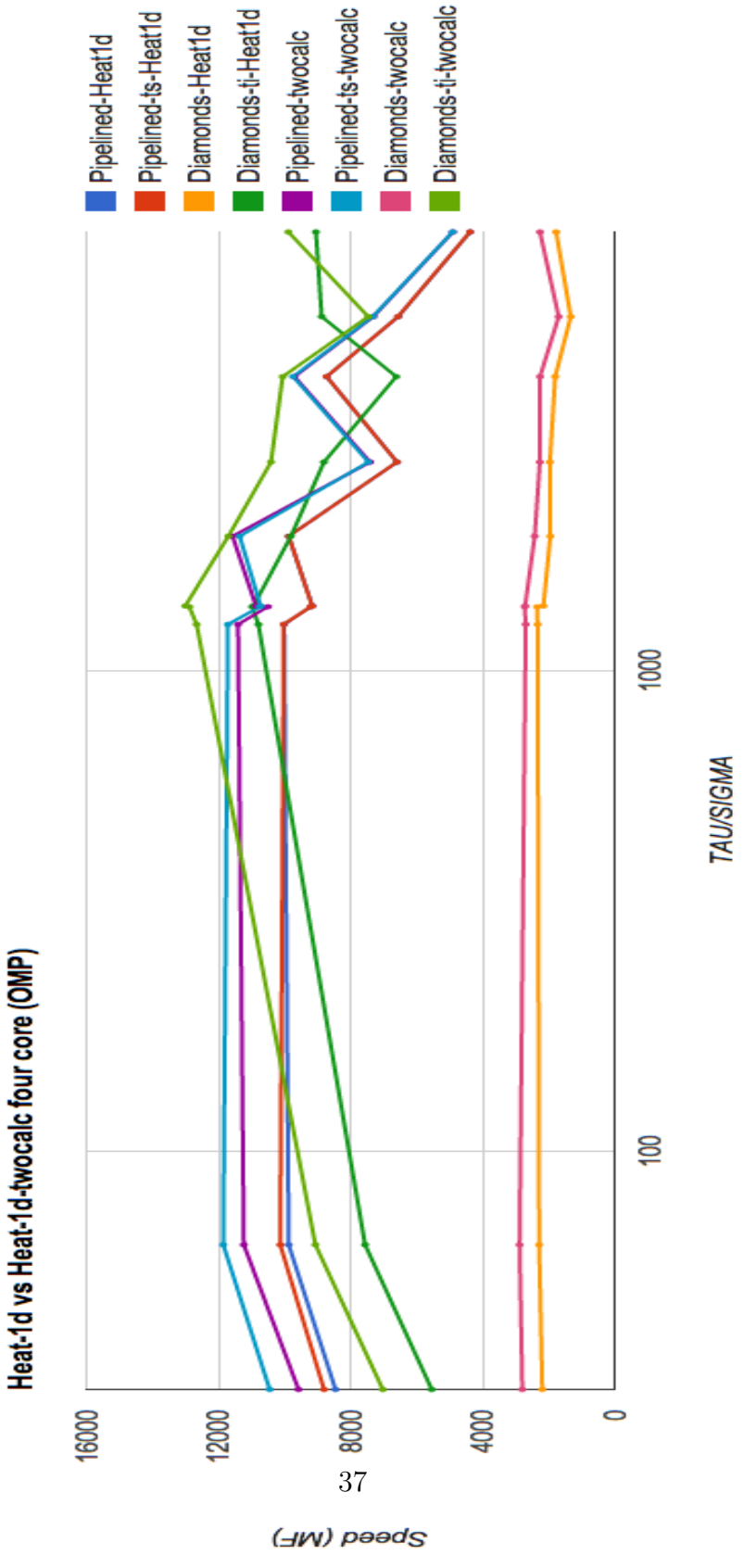

Figure 18: Heat-1d vs Heat-1d-twocalc four core linespoints

Figure 19: Heat-1d vs Heat-1d-twocalc single core bar graph

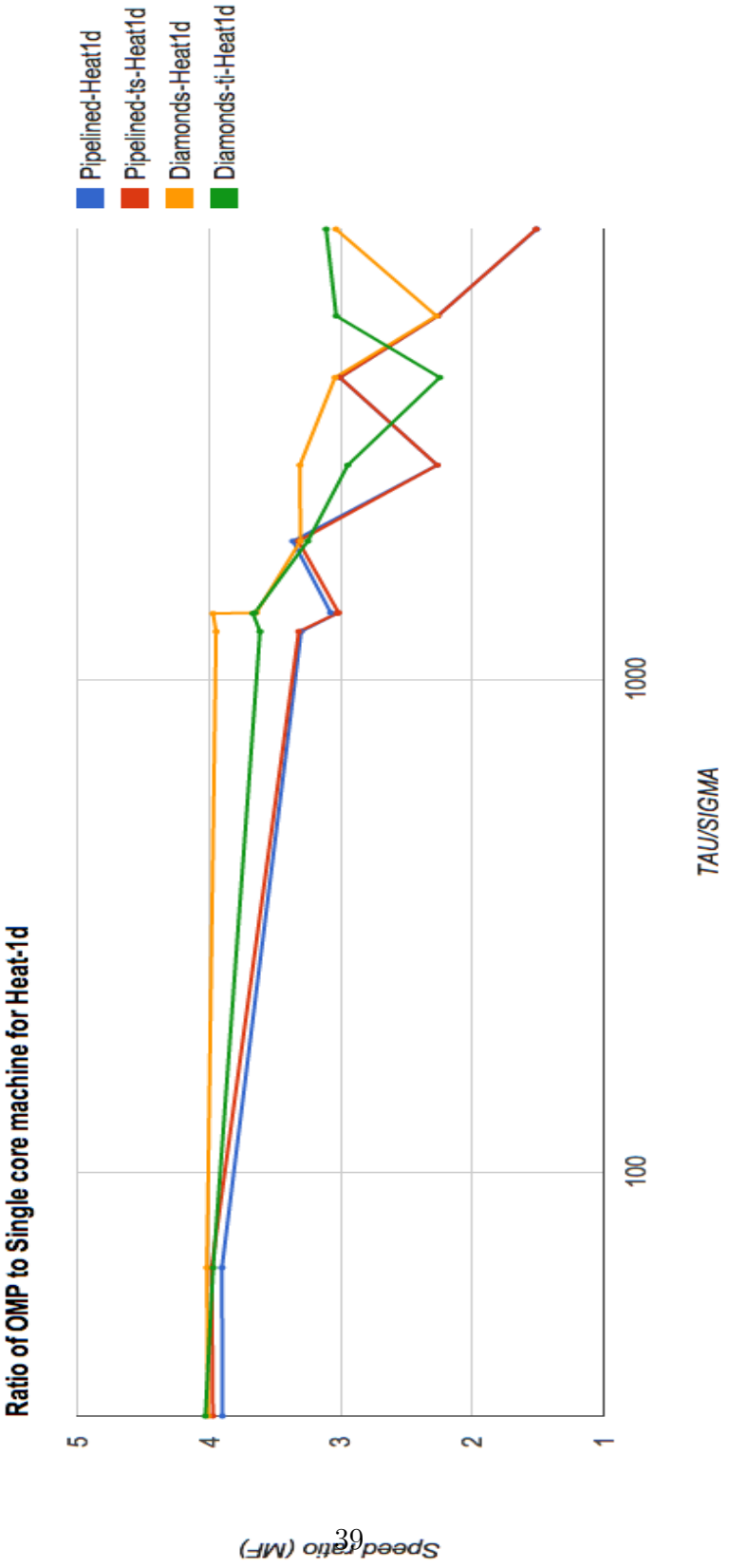

Figure 20: Ratio of OMP to single-core for Heat-1d linespoints

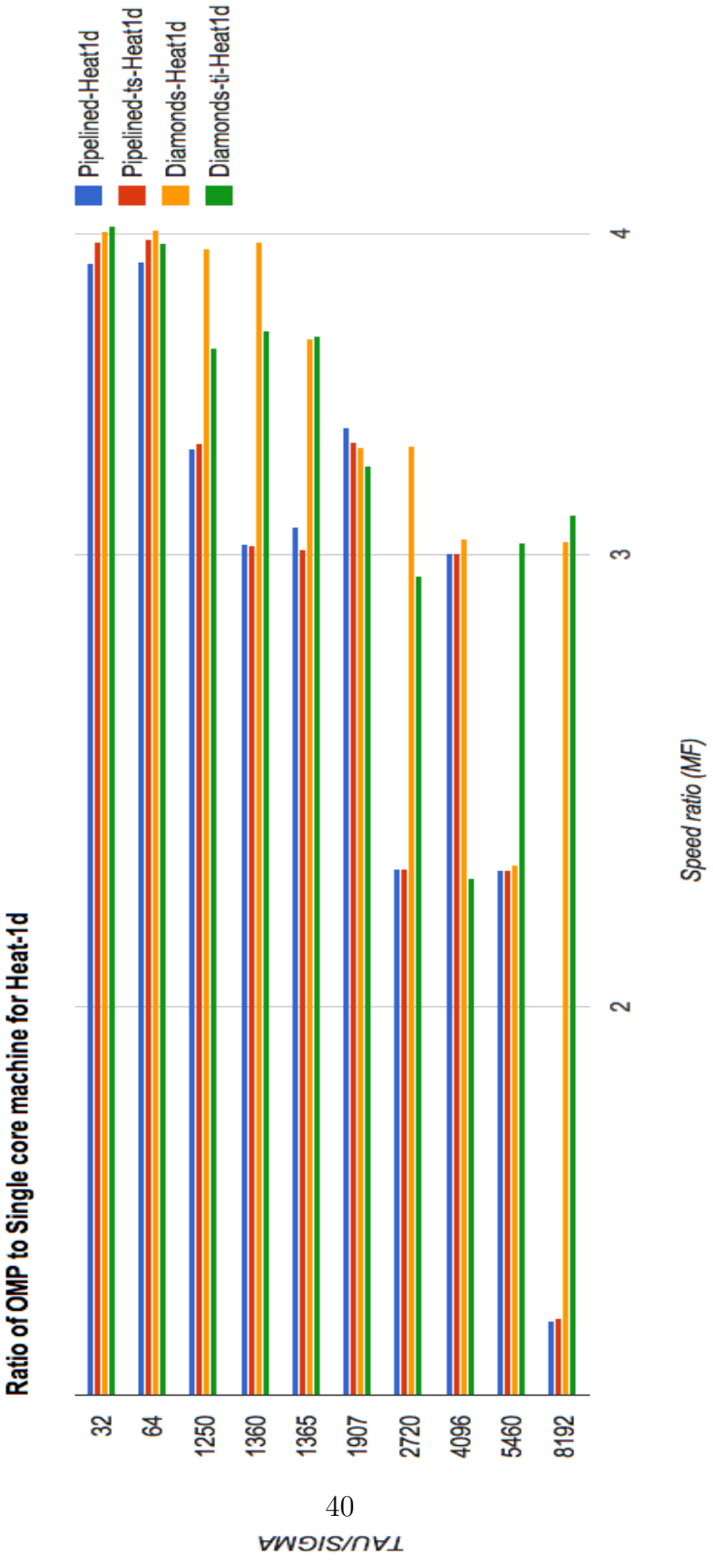

Figure 21: Ratio of OMP to single-core for Heat-1d bar graph

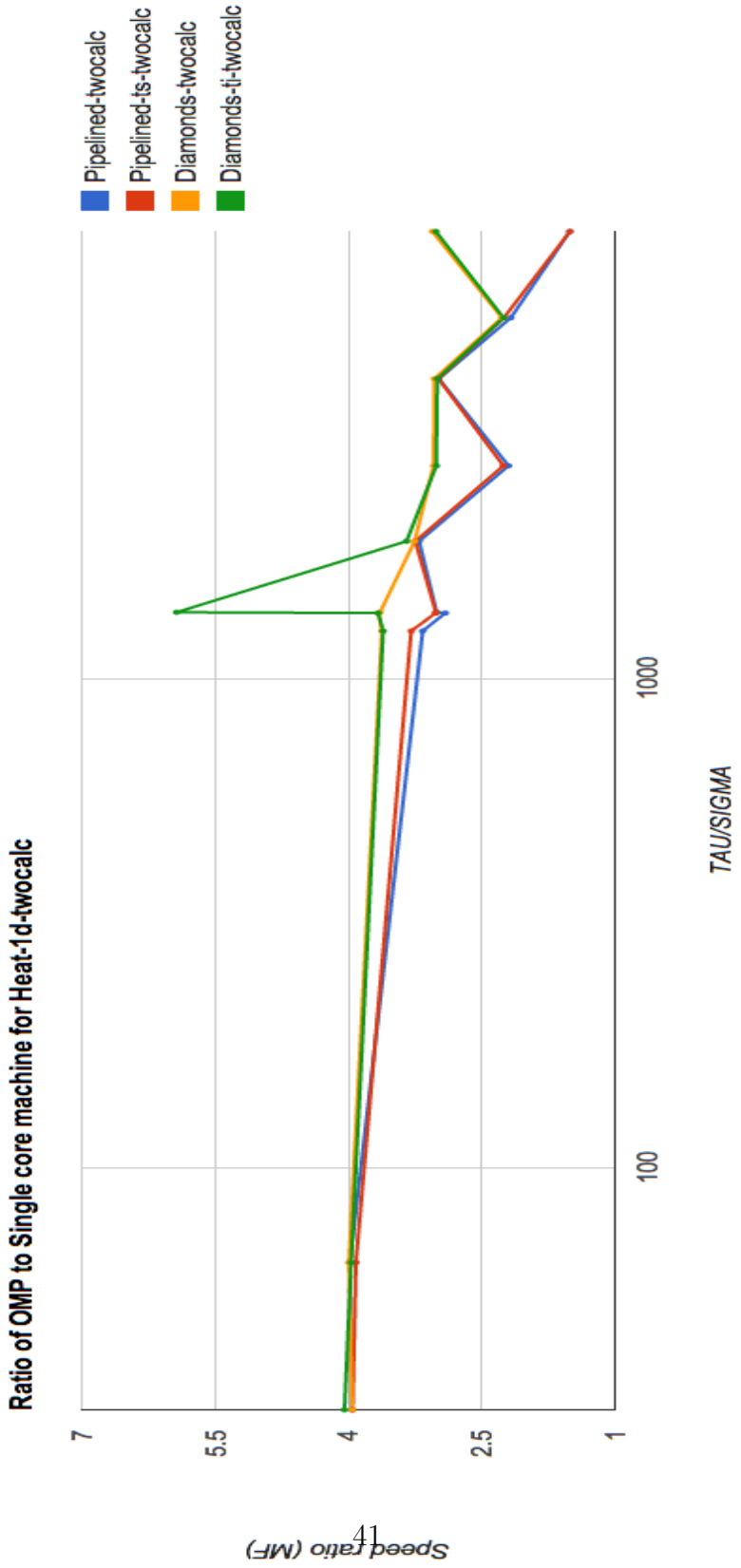

Figure 22: Ratio of OMP to single-core for Heat-1d-twocalc linespoints

Figure 23: Ratio of OMP to single-core for Heat-1d-twocalc bar graph

4 Part III: Discussion of Results

A total of sixteen graphs were constructed for the purposes of this analysis. The graphs are in pairs where for every variation of graph, there is a lines-points graph and a bar graph. For example, to investigate the single core for Heat-1d variation, we have both a lines-points single core for Heat-1d graph, as well as a bar graph for single core Heat-1d.This is helpful because it provides two visual representations of the same information. Each variation of graph will be discussed separately, and there will be a concluding section to summarize the information at the end of the discussion of the individual graphs.

It is important to note that all the experiments were done in 25,000 time steps (i.e. T = 25,000), and N equal to 8 million. Additionally, each experiment was repeated three times and the average speed of the three runs was taken. All three runs came within +/- 5 MF of the median result of that run.

*All graphs are logarithmically scaled.

4.1 Single core for Heat-1d

For this graph, the focus is on the case of the single core run for Heat-1d. This helps focus on how varying ti/ts with the pipelined and diamond tile shapes affects the outcome on a single core machine. Therefore, when expanding to look at the more interesting four-core runs, it becomes easy to see if the

change in speed was a result of the increase in cores or a different variable.

In this graph, all but the diamond with no ti variation increase with the increase of TAU/SIGMA. With this graph a maximum speed of close to 3000 MF is reached by all of the variations of pipelined, diamond tiling with/without ti/ts. There is little to no difference between the pipelined with and without ts runs. Their data-points are almost exactly superimposed. The ti version of diamond tiling shows a massive increase in speed over the non-ti version. Interestingly, diamonds-ti increases a lot more quickly as TAU/SIGMA increases and then plateaus at the same point that pipelined (/ts).

4.2 Four core for Heat-1d

For this graph, the focus is on the case of the four-core run for Heat-1d. This helps focus on how varying ti/ts with the pipelined and diamond tile shapes affects the outcome on a four core machine as opposed to the earlier single core machine. Thus, it becomes easy to compare the four-core aspect to the single-core aspect with all other variables staying constant.

In this graph, all but the diamond with no ti variation change speed substantially with the increase of TAU/SIGMA. With this graph a maximum speed of close to 11,000 MF is reached by Diamonds-ti, with the closest run to this are pipelined with and without ti. There is little to no difference between the pipelined with and without ts runs. Their data-points are almost exactly superimposed. The ti version of diamond tiling shows a massive increase

in speed over the non-ti version. As with the single core run, diamonds-ti increases a lot more quickly as TAU/SIGMA increases and then drops in speed after it has overtaken the pipelined runs at around TAU/SIGMA equal to 1365.

Additionally, at the point that Diamonds-ti reached it's best performance, the pipelined variations dip in speed significantly (i.e. by a factor of 2000 MF). Both Diamonds-ti and the pipelined variations decrease as the TAU/SIGMA increases past 1365, with Diamonds-ti plateauing near to TAU/SIGMA equal to 8000 MF.

4.3 Single core for Heat-1d-twocalc

For this graph, the focus is on the case of the single core run for Heat-1d-twocalc. This helps focus on how varying ti/ts with the pipelined and diamond tile shapes affects the outcome on a four core machine as opposed to the earlier single core machine. Thus, it becomes easy to compare the four-core aspect to the single-core aspect with all other variables staying constant.

In this graph, all but the diamond with no ti variation change speed with the increase of TAU/SIGMA. With this graph the Diamonds-ti runs reach a maximum speed of close to 3500 MF. With the Heat-1d-twocalc run, there is a small difference between the pipelined with and without ts runs as opposed to the minimal and insignificant difference between pipelined with and without ts in Heat-1d. With the Heat-1d-twocalc runs the data-points are no longer

superimposed; pipelined-ts starts at a higher speed than pipelined without ts at TAU/SIGMA equal to 32. However, after this point, both variations of pipelined increase until they reach the point where the data points are again almost exactly superimposed.

Diamonds-ti starts at a lower speed than the pipelined variations and eventually intersects with the pipelined runs and runs at almost the same speed as them. It is important to note that there is a trough in the Diamonds-ti run at TAU/SIGMA equal to 1365.

4.4 Four core for Heat-1d-twocalc

For this graph, the focus is on the case of the four-core run for Heat-1d-twocalc. This helps center the focus on how varying ti/ts with the pipelined and diamond tile shapes affects the outcome on a four core machine for Heat-1d-twocalc as opposed to the same variation in variables on the Heat-1d version.

In this graph, all but the diamond with no ti variation change speed with the increase of TAU/SIGMA. With this graph Diamonds-ti and all pipelined runs reach a maximum speed of close to 13,500 MF. With the Heat-1d-twocalc runs, there is a small difference between the pipelined with and without ts runs as opposed to the minimal and insignificant difference between pipelined with and without ts in Heat-1d. With the Heat-1d-twocalc four core runs the data-points are no longer superimposed; pipelined-ts starts at a higher speed than pipelined without ts at TAU/SIGMA equal to 32.

This is similar to the Heat-1d-twocalc single core results. However, after this point, both variations of pipelined increase until they reach the point where the data points are again almost exactly superimposed.

Diamonds-ti starts at a lower speed than the pipelined variations and eventually surpasses the pipelined runs and runs around TAU/SIGMA equal to 1365.

4.5 Heat-1d vs. Heat-1d-twocalc single core

For this graph, the focus is on the case of the single core runs for Heat-1d and Heat-1d-twocalc. This helps center the focus on changing the variables affected the outcome of the Heat-1d run versus how it affected the Heat-1d-twocalc run.

In this graph, we see that Diamonds in both Heat-1d and Heat-1d-twocalc shows the worst performance with no increase in speed as TAU/SIGMA increase. The best runs both occurred in the Heat-1d-twocalc run. The best runs were the Pipelined version of Heat-1d-twocalc with Diamonds-ti for the Heat-1d-twocalc version reaching the same speeds at a higher TAU/SIGMA value.

4.6 Heat-1d vs. Heat-1d-twocalc four core

For this graph, the focus is on the case of the four core runs for Heat-1d and Heat-1d-twocalc. This helps center the focus on changing the variables

affected the outcome of the Heat-1d run versus how it affected the Heat-1d-twocalc run.

In this graph, we see that Diamonds in both Heat-1d and Heat-1d-twocalc shows the worst performance with no increase in speed as TAU/SIGMA increase. The best runs both occurred in the Heat-1d-twocalc run. The best runs were the Diamonds-ti version of Heat-1d-twocalc with Diamonds-ti for the Heat-1d-twocalc version reaching the same speeds at a higher TAU/SIGMA value.

4.7 Ratio of single core to four core runs for Heat-1d

For this graph, the focus is on the case of the four core runs for Heat-1d versus the single core runs. This helps easily compare the Heat-1d single and four core runs and thus determine which factors improved the most when the number of cores was increased.

We see that Diamonds has the greatest increase in speed with the increase in the number of cores. This information is not terribly important, because the Diamonds runs did not reach a high enough speed to compete with the other runs.

The next greatest increase in speed with the increase in the number of cores is the Diamonds-ti run. This is a paramount result because in both the single core and the four core run of Heat-1d, diamonds-ti ended up running at the fastest speed. Therefore, the significant increase from the single core

to the four-core run shows that diamonds-ti maintains both the greatest increase in speed as well as the greatest speed reached overall.

4.8 Ratio of single core to four core runs for Heat-1d-twocalc

For this graph, the focus is on the case of the four core runs for Heat-1d-twocalc versus the single core runs. This helps easily compare the Heat-1d single and four core runs and thus determine which factors improved the most when the number of cores was increased.

Unlike the previous graph, here the greatest increase in speed from the single core to the four-core run is the Diamonds-ti and the Diamonds runs. Again, the Diamond run ratio is not terribly important, because the Diamonds runs did not reach a high enough speed to compete with the other runs.

The Diamonds-ti run is paramount because in both the single core and the four core run of Heat-1d, diamonds-ti ended up running at the fastest speed. Therefore, the significant increase from the single core to the four-core run shows that diamonds-ti maintains both the greatest increase in speed as well as the greatest speed reached overall. This is the same situation as the Heat-1d version of the runs.

5 Conclusion

The previous section conveyed how changing various variables affect the performance (measured in Megaflops). It is important to discuss how varying the variables affects the performance overall. The main conclusions can be divided into two sections: the fastest results obtained by running tests on four cores and the fastest results obtained by running tests on a single core.

5.1 Conclusions on four-core runs

When looking at the four core runs, it is important to ask which of the four factors investigated had the greatest influence on the results. It is also important to ask how much did each variable matter. To re-cap the four variables that were changing were: The tile size, the structure of the original code (Heat-1d versus Heat-1d-twocalc), the intra-tile ordering, and the tile shapes.

To begin, varying the tile size seems not to have an entirely positive or negative effect on the speed reached by a specific run for both Heat-1d and Heat-1d-twocalc at first. As the tile gets to a large enough size, the speed of the run begins to fluctuate between going down and then back up, but never surpassing the highest speed obtained at any previous tile size. For this reason, it seems as though the tile size has an overall minimal affect on the speed at which the program runs. An average difference of 1875 MF in difference in a single run was calculated, and therefore negligible in comparison to the factors investgiated.

Varying the program from Heat-1d to Heat-1d-twocalc has a distinctly different affect on the outcome of the program. That is to say that the Heat-1d program follows a very similar trend to the Heat-1d-twocalc program, however, the Heat-1d-twocalc version starts at a higher initial speed (TAU/SIGMA equal to 32), and reaches a higher overall speed. The difference in the fastest speed reached by the Heat-1d-twocalc program versus the Heat-1d program is around 3,000 MF.

As for the tile shape, in both the Heat-1d and the Heat-1d-twocalc program, the fastest speed reached is reached by the Diamonds tiling. The type of intra-tile ordering obviously has an affect on whether or not those run reached a high speed, but that will be discussed in the next paragraph. The difference in speed reached by a certain tile shape changes based on the intra-tile ordering as previously stated, but the difference between the lowest Diamond tile speed and the pipelined tile speed (both iterations of the pipelined tiling are almost the same) is approximately 8,000 MF. While, the difference between the highest speeds reached by the Diamond tiling and the pipelined tile speed are approximately 2,000 MFs.

Varying the intra-tile ordering has almost no affect on the pipelined runs between the Heat-1d and the Heat-1d-twocalc programs. However, with regard to the Diamond tiles, there is a very large increase in speed between the regular and the Diamond-ti iteration. In the Heat-1d version, the difference in the highest speeds reached by Diamonds and Diamond-ti is about 9,000 MF. In the Heat-1d-twocalc version, the difference in the highest speeds reached by Diamonds and Diamond-ti is about 10,000 MF

5.2 Conclusions on single-core runs

When looking at the single core runs, it is important to ask which of the four factors investigated had the greatest influence on the results. As well as this, it is important to ask how much did each variable matter. To re-cap the four variables that were changing were: The tile size, the structure of the original code (Heat-1d versus Heat-1d-twocalc), the intra-tile ordering, and the tile shapes.

To begin, increasing the tile size seems to increase the maximum speed reached by a specific run until a certain point at which the speed plateaus and stays constant. This is the case in both the Heat-1d and Heat-1d-twocalc runs. As the tile gets to a large enough size, the speed of the run begins to increase by a lesser degree until it stops increasing with the increase in tile size. For this reason, it seems as though the tile size has an overall minimal affect on the speed at which the program runs and is thus negligible.

Varying the program from Heat-1d to Heat-1d-twocalc has a small, but positive affect on the outcome of the program. That is to say that the Heat-1d program follows a very similar trend to the Heat-1d-twocalc program, however, the Heat-1d-twocalc version starts at a higher initial speed at TAU/SIGMA equal to 32, and thus reaches a higher overall speed. The difference in the fastest speed reached by the Heat-1d-twocalc program versus the Heat-1d program is around 500 MF.

As for the tile shape, in both the Heat-1d and the Heat-1d-twocalc pro-

gram, the tile shape is the biggest determiner in how high the initial speed of the run is, as well as how much the speed of the run increases by. All of the runs, aside from the Diamond non-ti version, end up converging at the same point (i.e. TAU/SIGMA equal to 1360) and after this, they stay at the same speed. This happens in both the Heat-1d and Heat-1d-twocalc version of the program. The only difference being that one of the programs starts at a higher speed as stated previously. The difference in speed reached by a certain tile shape changes based on the intra-tile ordering as previously stated, but the difference between the lowest Diamond tile speed and the pipelined tile speed (both iterations of the pipelined tiling are almost the same) is approximately 2500 MF in both the Heat-1d and the Heat-1d-twocalc versions. While, the difference between the highest speed reached by the Diamond tiling and the highest speed reached by the pipelined tile is almost 0 MFs.

Varying the intra-tile ordering has almost no affect on the pipelined runs in both the Heat-1d and the Heat-1d-twocalc programs. However, with regard to the Diamond tiles, there is a very large increase in speed between the regular and the Diamond-ti iteration. In the Heat-1d version, the difference between Diamonds and Diamond-ti is about 2500 MF at the highest speed for each.. In the Heat-1d-twocalc version, the difference between Diamonds and Diamond-ti is about 2800 MF at the highest speed for each.

5.3 Summary

We saw how changing different variables affected each of the four core and the single core runs and by how much. It seems that the four core version reached higher speeds overall, and thus the number of cores seems to correlate directly with the biggest increase in speed. However, within the four-core runs, using Diamond tiling with ti had the greatest affect on the outcome of the speed. These two factors had the greatest affect and they obtained the fastest overall speed.

Therefore, relating this back to the hypothesis, we see that by comparing the factors we chose to investigate, the variation of intra-tile order with the Diamond shaped tiles appears to have the greatest result/impact.

6 Future Directions

The experiments shown in this paper were doing with T = 2500. However, in the future, it could be interesting to see how the increase in the number of time steps could affect the outcome of the program. Therefore, this variable is a very interesting one to investigate further.

Another direction that we would like to explore is the two-dimensional iteration space programs. This would basically be the same as Heat-1d, Heat-1d-twocalc, and Heat-1d-withcopy, but instead of 1d, they would be Heat-2d, Heat-2d-twocalc, and Heat-2d-withcopy.

Additionally, initially, there were meant to be three programs investigated: Heat-1d, Heat-1d-twocalc, and Heat-1d-withcopy. The Heat-1d-withcopy results could ultimately not be included, because when the Heat-1d-withcopy tests were run they produced significantly different results to the Heat-1d and Heat-1d-twocalc versions. When the Heat-1d and Heat-1d-twocalc programs were re-run the results could not be replicated, and therefore, the results from Heat-1d-withcopy could not be compared to the Heat-1d and Heat-1d-twocalc runs. This created an inconsistency with the Heat-1d and the Heat-1d-twocalc versions of the program. Therefore, in the future, the Heat-1d-withcopy would ultimately be run in a similar setting to that which Heat-1d and Heat-1d-twocalc were run.

References

[1] Michael J. Wolfe *High Performance Compilers for Parallel Computing* 1996: Addison-Wesley.

[2] Vinayaka Bandishti, Irshad Pananilath and Uday Bondhugula *Tiling Stencil Computations to Maximize Parallelism* Indian Institute of Science, Bangalore 560012

[3] David G. Wonnacott and Michelle Mills Strout *On the Scalability of Loop Tiling Techniques* Haverford College, Haverford PA, 19041 and Colorado State University, Fort Collins, CO, 80523

[4] William Pugh *The Omega Test: a fast and practical integer programming algorithm for dependence analysis* University of Maryland, College Park MD, 20742

[5] Carnegie Mellon University *Computing the Localized Iteration Space* Carnegie Mellon University, n.d. Web. 23 Apr. 2014.

[6] Verdoolaege, Sven. iscc Tutorial. PowerPoint presentation. Team Alchemy, Saclay, France. 22 November 2011.