Factors Affecting Stencil Code Performance

Ting Zhou
Supervised by David G. Wonnacott

03–10–2016

Abstract

In the field of scientific computation, loop tiling is an indispensable technique for improving cache performance, and thereby the overall performance of the code. Research so far has predominantly been focusing on optimizing the code of a particular tiling choice, under a specific problem setting. In this thesis, I wish to both statistically explore the most important factors in different tiling scenarios, as well as the role the problem parameters may play when making tiling decisions.
Contents

1 Introduction 3

2 Literature 8
   2.1 A Data Locality Optimizing Algorithm ......................... 8
   2.2 On the Scalability of Loop Tiling Techniques .................... 8
   2.3 Achieving Scalable Locality with Time Skewing ................... 9
   2.4 Optimal Orthogonal Tiling of 2-D Iterations ...................... 9
   2.5 Optimal Semi-Oblique Tiling .................................. 10
   2.6 Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles ............................................. 10
   2.7 Parameterized Loop Tiling ..................................... 10
   2.8 Hierarchical Overlapped Tiling .................................. 10

3 Methodology 11

4 Problem 14

5 Tiling 17
   5.1 One Level Tiling ............................................. 17
   5.2 Two Level Tiling ............................................. 21

6 Other Factors 26

7 Tests 28

8 Results and Analysis 30

9 Deviations 41

10 Conclusions 46

A Some MySQL Queries Used 50
1 Introduction

In this section, I wish to introduce some basic concepts, paired with examples, to provide some background for the content to come.

Compiler  A compiler is defined as a computer program that translates code written in one computer language into another. The languages involved can be high-level human readable languages, such as C++ or Swift, or low level machine language / assembly code, such as x86/64 instructions or ARM assembly. Usually when we talk about a compiler, we mean one that translates source files written in high-level language to binary machine executable.

Translating high-level source code into machine binary involves more than simply matching keywords to control structures and variables to registers. As there are a limited number of registers and comparatively small set of instructions for a given CPU, the compiler needs to undertake a great deal of decision-making to produce an executable from a source file. Yet modern compilers have reached such an advanced state that they already incorporate many optimization techniques and can detect and apply them when appropriate.

Memory Hierarchy  While the processing power of CPUs have been consistently growing, the actual performance of computation is often bottlenecked by the speed at which the processor can access data from the memory. There are two primary factors affecting the access speed for a particular memory device: its distance from the CPU, and the way it is manufactured.\footnote{We consider the bus-width and bus-speed as part of the memory’s construction for this discussion.}

On the scale of a CPU, where billions of operations occur every second, even centimeters of difference in distance can cause a variation in access latency. As for the manufacturing of the memory devices, the faster a memory device is, the more expensive it is. Usually the faster and more costly SRAM (Static Random Access Memory) is used as CPU cache, as it typically does not require refreshing; the slower DRAM (Dynamic Random Access Memory) is used in what we usually refer to as RAM.

The cost-effective balancing of these two factors determines the kind of memory hierarchy we see in most computers today. On the top level we have registers right inside of the processors, which holds the pieces of data the processor is actively operating on. After these we may have multiple levels of CPU cache that may be assigned to cores or be shared among them; they are slower than registers, but also
holds much more data. A typical three-level CPU cache may have 32KB of level 1 data cache and 32KB of level 1 instruction cache per CPU core, 256KB of level 2 cache per core, and 8MB of shared level 3 cache. After the caches we have the RAM, which are commonly multiple gigabytes in size for home-owned PCs, and the hard drive, which are non-volatile memory devices used to store data across multiple sessions of computer usage.

![Memory hierarchy of a typical computer](image)

Figure 1: Memory hierarchy of a typical computer

The data in the memory are stored in ‘words’, a unit of data that can be determined by an address in memory, typically 64 bits. The cache stores multiple words in what is called a ‘line’, a consecutive group of words. When the processor needs to obtain some data from the memory, it does so by first checking the cache to see if a line containing that data is already present. If it successfully finds it (called a cache-hit), it simply retrieves it from cache and write any changes back to it. If not (a cache-miss), the processor requests an entire line containing the data, and places it in cache, before operating on it. When there is not sufficient empty space in the cache, it (usually) replaces the least-recently used (LRU) line with the new line. This way, the cache always maintains a set of most recently accessed data.

**Stencil**  An Iterative Stencil code is a type of loop code where a set of data points are repeatedly updated according to some fixed pattern (called a *stencil*) for multiple time steps, involving usually several dimensions of data space and one time dimension. This type of code is frequently seen in scientific computing, and examples include physical simulation of real life phenomena, such as heat conduction, and solving partial differential equations by finite difference. Due to their fixed data usage pattern, they are usually great candidates for tiling.
Tiling  Tiling is one of the most common means to improve stencil code performance. By rearranging and grouping the order of execution (or \textit{iteration space}) while preserving the information flow (or \textit{dependency}), we can try to fit a segment of data in cache, and perform as many operations on them as possible without violating data dependency before having to displace it. In this way, we reduce the amount of cache misses and thus time taken to fetch data from memory.

There are many factors to decide when performing tiling on a piece of code. The most basic of which include the shape and size of the tile. Parallelograms (‘pipelines’) and diamonds are the most common for 1-D data arrays, while data in higher dimensions have corresponding high-dimensional analogies of parallelograms (like parallelepipeds in for 2-D data) and diamonds (cubes for 2-D data).

\begin{figure}[h]
\centering
\includegraphics[width=0.3\textwidth]{simple_block_tiling.png}
\caption{Simple Block Tiling [Won02]}
\end{figure}

Suppose we have a CPU with only one level of cache, capable of holding 4 floats at once. Also suppose we are given the problem to update 8 floats 5 times each, with each updating to a float only requiring information from itself in a previous time step. Then we have a data flow graph as in Figure 2, and the code would look like so:

```java
for (int t = 1; i < 6; i++) {
    for (int i = 1; i < 9; i++) {
        A[i] = update_from_value(A[i]); // A is an array
    }
}
```


If we approach the problem naively, we may decide to use \( t \) as the outer loop, and \( i \) as the inner loop. Since the stencil in the problem is very simple, it is tempting to think that the choice of the order of loop variables does not matter. However, if we do proceed as described, then since our level 1 cache can only hold 4 floats at once, at the first time step the processor would request data points 0 through 3 from the RAM, update them, then request data points 4 through 7. As they are not in cache, they are replaced from the cache. On time step two, since now points 0 through 3 are no longer in cache, they are accessed again from the RAM, and the same process repeats.

If instead, we tile our iteration space along the \( i \) axis, and execute our code block by block, then for each block, we would only need to request data from RAM once, and can finish while using only two cache-misses, both of which are necessary anyway.

![Figure 3: Pipelined Tiling][Won02]

In this example, we see the data dependency is more complex: each data point depends on data from both itself and its two neighbors’ value at the previous time step. In this case, we perform what is known as ‘pipeline tiling’, and now since we have chosen an appropriately sized parallelogram-shaped tile, we only need to read each value once per tile, which results in reduced miss-rate.

In addition to better cache locality, tiling also allows for easier parallelization of code. For the first example, we can assign each rectangular tile to a separate core. For the second example, simply assigning one parallelogram to a core does not work: if we run the tile containing time steps \([1, 2]\) and data points \([1, 2, 3, 6]\)
concurrently with the tile above it, containing time steps [1, 2] and data points [4, 5, 6, 7, 8], it is possible that at some point in calculation, we need to calculate the value for point \((t = 4, i = 2)\), but the value for point \((t = 1, i = 3)\) is not up to date. As such, a tile can only be run concurrently with other tiles not depending on them, which in this case, are the tiles that share only one vertex with one another on the acute angle.
2 Literature

2.1 A Data Locality Optimizing Algorithm

[WL91] Wolf and Lam discuss an algorithm to optimize data locality by means of interchange, reversal, skewing and tiling loop code. The article mentions that these following factors may affect the efficiency of optimized code:

1. Processor count
2. Cache size
3. Tile shape
4. Problem size
5. Tile size
6. Tile Style, e.g. 2D, 3D tiling

It is worth noting that while Wolf and Lam acknowledges the effect of problem size on the performance of the code, they consider its effect largely due to cache interference, and mitigable by simply changing the tile size.

2.2 On the Scalability of Loop Tiling Techniques

[WS13] Strout and Wonnacott examines whether some of the current techniques for tiling are well-suited for machines that demand greater level of parallelism and memory traffic control. Of the factors potentially affecting performance, the primary focus of the article are tile shapes, but they also listed the following:

1. Processor Count
2. Problem size
3. Code style, e.g complexity, vectorization
4. Hardware specs

They talk about the fact that, while pipeline tiling can achieve some degree of parallelization, it fails to scale well as the problem parameters grow, due to how it can only run concurrently on a slanted wavefront. For linearly scaling parallelism, it demands the problem grow in both dimensions.
They note that it might be possible to reduce tile sizes in order to achieve better parallelization, and that for different hardware and perhaps even problem sizes, there may be different tile shapes that are suitable, in addition to tile sizes.

They also survey some possible tilings that could yield different results, namely overlapping tiling, which increases execution redundancy; trapezoid tiling, which have good theoretical scalability, but can be really hard to implement; diamond tiling, which has parallelism along the time dimension and scales well with it, but may not extend well to higher dimensions; and molecular tiling, which exhibits good scalability but, like trapezoid tiling, could be challenging to implement.

The authors then talk about a new tiling that both potentially scales well, and extends well to higher dimensions. They then conclude that automatic parallelization of code would require a general tiling technique that scales well, extends well for higher dimensions, and is not too difficult to implement.

2.3 Achieving Scalable Locality with Time Skewing

[Won02] Wonnacott discusses how to use time-skewing on certain time-step based loop calculations to achieve ‘scalable locality’. That is, to be able to increase reuse rate of data transferred to or from memory without requiring linear growth of cache memory. He notes that time-skewing may allow certain code that is previously unscalable by other techniques to have scalable locality, by reordering the iteration space and optionally remapping the storage. This reiterates the relation between types of tiling and the kinds of computation being performed, and provides a basis of the discussion I wish to present in my thesis.

The author mentions that the following factors may affect tiled-code performance:

1. Hardware specs
2. Code style
3. Cache size
4. Problem size

2.4 Optimal Orthogonal Tiling of 2-D Iterations

[AR97] Andonov and Rajopadhye explore the effect that tile size for a given tile shape has on the execution time of the code. Their model is under the assumption that ‘the dependencies are uniform, the iteration space is two-dimensional and parallelogram shaped, and the tile boundaries are parallel to the domain boundary’.
Under these constraints, they gave a closed-form optimal tile size solution for a problem of given size, on a certain machine. This confirms that with all else being equal, there exists an optimal size setting; it does not explore, however, if or how changing other variables, such as tile shape and loop style, may affect this optimal size.

2.5 Optimal Semi-Oblique Tiling

[ABRY03] Andonov et al. examines how to optimally tile 2-D iterations with semi-oblique tiling. They claim that under the constraint of a parallelogram-shaped 2-D iteration domain, the problem is completely specified by the problem’s length and height, the number of processors, and the tile shape. The authors then produces a theoretical optimal tile size mathematically.

2.6 Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles

[GCK+13] Grosser discusses a method of tiling for GPUs called split-tiling. It proposes a way to enable inter-tile parallelization by subdividing each tile on the spatial dimensions. He states that this method works without requiring skewing of the data or having redundant calculations, and works on arbitrary number of dimensions. While this discussion focuses on GPU code tiling, it is likely that some portions of it may also apply to CPUs.

2.7 Parameterized Loop Tiling

[RKSR12] Strout and Rajopadhye discuss a tiling technique known as parameterized tiling. It differs from the common practice of setting tile-size at compile-time by allowing the program to dynamically determine tile-sizes at run time. While this thesis is more oriented towards statically-sized tiling, this may still provide insight to our exploration.

2.8 Hierarchical Overlapped Tiling

[ZGG+12] Zhou et al. discuss how to improve computational efficiency on GPUs by using tiles that have overlaps. The main reason for doing so is that adjusting for irregular tile shapes and boundary conditions can be expensive on GPUs, and it may be worth the trade-off between the improved data/instruction cache locality and the added redundant calculation.
3 Methodology

The primary focus of this thesis will be attempting to determine of the extent to which various tiling choices and factors may affect the performance of the tiled code.

At its core, this is an optimization problem in a high dimensional space. Note that our task is no longer finding the solution to our problem, but rather the time taken to find the solution. The variables are the decisions the compiler should make in order to optimize the time. Therefore, the original input to the problem are now actually constants, but are nonetheless factors to consider when making the decisions. The full sets of variables/factors we hope to explore are as follows:

1. The algorithm. This factor is determined prior to our experiments, since the compiler cannot modify the core of the algorithm.\(^2\)

2. The hardware. The various hardware specs, such as processor rate, number of cores, cache sizes, etc. These factors are determined prior to our experiments: the compiler cannot change the hardware.

3. The problem parameters, and specifically, the size and shape of the data set. This factor is also determined prior to our experiments.

Importantly, while these factors above are constants, they do greatly impact our decisions in the following factors. In addition to them, we also have these variable factors to consider:

1. Implementation of the algorithm. Note that this is different from the algorithm itself: the same algorithm can be written in different ways, such that while they have the same complexity, their run time could significantly differ. We will see an example of this in the following sections.

2. Single versus multiple core computation, and the infrastructure to enable communication between the cores.\(^3\)

3. Levels of tiling, and strategy of tiling. As there are multiple levels of CPU cache, it makes sense that we would attempt to subdivide tiles into sub-tiles for better intra-tile cache performance. Strategy of tiling refers to whether we choose to split the time-dimension (regular) or not (prism).

\(^2\)That is, until smart compilers start to recognize the problem the algorithm is trying to solve, then automatically switch to the optimal algorithm for that problem.

\(^3\)the latter of which is more dependent on the hardware and multi-core processing APIs such as OpenMP.
4. Tile shape. Note that we may very well have different tile shapes on different levels of tiles, such as diamond-shaped sub-tiles inside large parallelogram tiles.

5. Tile sizes. For parallelograms/parallelepipeds, this may require several separate parameters, while diamonds/cubes have a consistent edge length on all sides.

6. Intra-tile execution order.

7. Inter-tile execution order.

8. Other loop-optimization technique in the loop body

9. Data skewing, or the transformation of the data space (not the iteration space, which is the skewing the time-dimension).

10. Static (determined at compile time) or parametric (determined at run time) tile sizes.

11. Compiler used for optimizations other than tiling.

In this thesis, we will explore factors experimentally, leaving 9, 11 for future work.

**Data Collection** Of these variables, most are discrete variables with 2–5 options, except tile-size and problem-size, which despite being integers, vary over a large enough range to be considered as continuous. Of the approximately 1000 – 10000 combination of those discrete variables. One of the main reasons why we cannot test each variables by varying them one at a time, is that we cannot say for sure that they are independent. In fact, given what we know, some of the variables are highly dependent, which means there is probably no ‘one-size-fits-all’ for all the variables. Meanwhile some other variables are rather independent from the others, and we have a decent idea which of them are.

A good way to determine these more independent variables would be to run a group of randomized test suites, each one fixing all variables but the ones suspected to be always good. If in all of these cases the performance of setting a variable to a certain value is better than if it is set to another, regardless of the other variables, we would be able to say with some confidence that they are some what independent from the rest of the variables, and we can just optimize them, then use them as constants throughout the following experiments.

Having done so, we hopefully will be left with around 4 – 7 discrete variables. In previous phases of this research, we have created scripts that can largely automate
the complete process of compiling code for a set of inputs. We may be able to find certain combinations of variables particularly undesirable, and eliminate them from further experiments. Once we are left within a reasonably sized number of variable combinations, we would start running tests and collecting results.

We foresee setting up a list of problems that are distinct in shape and size as benchmarks, and for each viable combination determined above, run the tests multiple times to find the optimal tile size for that particular combination. By the end of this phase we should have a large array of vectors, consisting of input variables values and their corresponding optimal tile size and time performance.

**Information**  The base code used in the thesis is provided by Dave Wonmacott, while the script used to run batch tests and the two-level tiling code is done by me. All of the code, unless otherwise specified, is compiled using GCC-4.8.5 on Ubuntu 14.04. The machine on which the tests are run has a Intel i7-860 quad-core processor with Hyper-threading off, having four cores with 2.8 GHz rate and practical bound of 4 GHz. Each core has 32KB level-1 instruction cache, 32KB level-1 data cache, 256 KB level-2 cache, and the four cores share a 8MB level-3 cache.\(^4\)

\(^4\)The information obtained directly through `/proc/cpuinfo` and `/sys/devices/system/cpu` from the machine.
4 Problem

The code we use in this exploration is that of 1-dimensional and 2-dimensional Jacobi code. What are they? It is a model used to simulate the spread of heat over a rod or a plate, for the two versions respectively. In the one dimensional case, we have a rod defined as an array of floats with length $N$, denoted $A$, where $A_i$ denotes the temperature, in some arbitrary unit, of the $i$’th segment of the rod. The value of $A_0$ and $A_{N-1}$ are fixed, and we would like to know the status of the rod after $T$ number of time steps. In each iteration, we update the array by calculating the weighted average for each segment $A_i = \frac{1}{4}(A_{i-1} + 2A_i + A_{i+1})$. So the overall naive loop for the function is:

```cpp
for (int t = 1, t <= T; t++) {
    for (int i = 1; i < N - 1; i++) { // boundaries are fixed and thus skipped
    }
}
```

The 2-dimensional version defines $A$ as a matrix of size $M \times N$, simulating a metal grid, where $A_{i,j}$ is the square on the $i$’th row and $j$’th column. Again, the temperature on the grid’s boundaries is fixed, and on each of $T$ time steps, we update $A_{i,j}$ as the weighted average $A_{i,j} = \frac{1}{6}(A_{i,j-1} + A_{i,j+1} + A_{i-1,j} + A_{i+1,j} + 2A_{i,j})$. Thus the overall naive loop is:

```cpp
for (int t = 1, t <= T; t++) {
    for (int i = 1, i < M - 1; i++) {
        for (int j = 1; j < N - 1; j++) {
        }
    }
}
```

The obvious problem with both 1D and 2D versions described above is that the calculation occurs in place. For instance, when calculating $A_4$ for $t = 2$, we take the weighted average of values of $A_3$, $2A_4$ and $A_5$. This operation changes the value of $A_4$, so when we calculate $A_5$ next, the value of $A_4$ has been ‘corrupted’, and the model no longer correctly represents the flow of heat on the rod. We can handle this in multiple different ways (assuming 1D for now).
**Copy**  The copy method does what its name suggests. At each time step, the calculation is performed, and the result for each segment is stored in a separate array $B$ so as not to corrupt the working array $A$. When the entire array is finished, we then copy the information from the temporary array back to $A$. The code for each time step would look like this:

```c
for (int t = 1, t <= T; t++) {
    for (int i = 1; i < N - 1; i++) {
    }
    for (int i = 1; i < N - 1; i++) {
        A[i] = B[i];
    }
}
```

**Two-Calculations**  The two-calculation (‘twocalc’) method does something similar. It also creates a secondary array $B$, but instead of always doing calculations in $B$ then copying over the result to $A$, the method calculates the values of $B$ based on $A$, then switch to calculate the values of $A$ based on $B$, and alternates between the two. Doing so requires maintaining two arrays of the same size at all times, exchanging space for efficiency.

```c
for (int t = 1, t <= T; t+=2) {
    for (i = 1; i < N-1; i++) {
    }
    if (t < T) { // in case T is odd
        for (i = 1; i < N-1; i++) {
            A[i] = (B[i-1] + 2*B[i] + B[i+1]) * 0.25;
        }
    }
}
```

**Swap Rows**  The swap-rows method essentially combines the two arrays $A$ and $B$ into a single $2 \times N$ matrix. It is accessed through a C-macro $A(t, i)$, which returns or modifies the data entry $A_{t\%2,i}$ in the matrix. Thus, a calculation step on $A_4$ at time step 2 will invoke the following code:

$$A(2, 4) = (A(1,3) + 2*A(1,4) + A(1,5)) * 0.25;$$
which translates to


and the overall loop looks like:

```java
for (int t = 1, t <= T; t++) {
    for (int i = 1; i < N - 1; i++) { // boundaries are fixed and thus skipped
        A(t, i) = (A(t, i-1) + 2*A(t, i) + A(t, i+1)) \times 0.25;
    }
}
```

All three methods achieve the same goal, yet we expect that, due to the different ways in which each method handles data and instructions, this choice will have a non-negligible effect on the runtime performance. In fact, without even testing, we can assume that the `copy` method is going to be the worst of three on a modern machine, due to the excessive amount of extra memory operations. The other two operates mostly the same, except one uses an `if` clause to deal with loop index parity, while the other uses modular arithmetic for the same purpose.
5  Tiling

We went through some basic style of tiling in the introduction. For this exploration, we use 7 different version of tiling for the 1D problem: no tiling, 1-level Diamond tiling, 1-level Parallelogram\(^5\) tiling, 2-level Diamond-Diamond, 2-level Diamond-Pipeline, 2-level Pipeline-Pipeline and 2-level Pipeline-Diamond.

Below we use a notation similar to the syntax used for ISCC. For each version of tiling, we start with the vector \([t, i]\), which represents an operation on the \(i\)th element at time \(t\), in two level of loops, \(t\) and \(i\). We use the notation \([t, i] \rightarrow [t+i, t, i]\) to represent a linear transformation from \(\mathbb{N}^2\) to \(\mathbb{N}^3\), reordering the iteration order to execute all elements \((t, i)\) with the same \(t + i\) value first, while executing in the other-wise normal order among those that do. For each version of tiling we also show a sample graph (Fig. 4), showing the iteration order for a small example. The graph below is the iteration order for a untiled version of the code: calculation on each entire row is completed before calculation on the next row begins, until all rows are finished.

![Figure 4: Iteration order in untiled loop](image)

5.1 One Level Tiling

One level tiling is simply the practice of tiling the iteration space into tiles of the same size and shape (barring corner cases). For our particular example, as we have mentioned in the introduction, we cannot simply use rectangles for tile shape. This is because the way information flow through the iteration space: in the 1D jacobi problem, we see that the value of a segment \(A_i\) of the rod at time \(t\) is dependent on not just \(A_i\)’s value at \(t - 1\), but also the value of \(A_{i-1}\) and \(A_{i+1}\) at time \(t - 1\). If

\(^5\)also called ‘pipeline’
we are to perform a calculation on $A_i$, before the calculation for $A_{i-1}$ and $A_{i+1}$ are properly updated, the results would be erroneous. Were we to use rectangles, each rectangle requires information from tiles beneath it as well as to its left and right — so there is no way to actually start from a tile, unless the tile size has width $N$, and the tile is as large as the problem.

We show some graphs illustrating the iteration order when using those tilings, and we mark the tile boundaries with red lines.

**Parallelogram**  Parallelogram, or Pipeline tiling, is one of the more common tiling methods for 1D stencil code. Simply speaking, given the iteration space as a 2-dimensional plane, with one data dimension $x$ and one time dimension $t$, it cust the plan into parallelograms that are formed by the lines $x + t = k\sigma$ and $t = l\tau$, $k, l \in \mathbb{N}$, where $\sigma$ denotes the number of points along the slanted side and $\tau$, that of the flat side.

![Figure 5: Data flow pattern for Jacobi 1D][RKSR12]

![Figure 6: Iteration order of one level pipeline tiling](image)

Looking at the two graphs above (Fig. 5, 6), we see the merit of pipeline tiling in our problem. By executing tiles according to the order (both intra-tile and inter-tile) illustrated above, we are never in danger of violating the data dependency shown
in Figure 5. Each tile satisfies the data dependency internally, so as long as we go from tiles

Pipeline-tiled code are relatively easy to parallelize, albeit a concurrent start is not possible, due to the inter-tile dependency to each parallelogram’s left, below and bottom-left side. Each tile can be run only currently with two neighboring tiles that share only one vertex with it, either at its top left or bottom right corner. A chain of such parallelograms can be run together in the form of a ‘wavefront’.

In our experiments, the tiling is achieved by using a C-preprocessing tool, **ISCC**, that rewrites loops based on loop indices. Essentially, given a loop with the form

```c
for (int t = 1, t <= T; t++) {
    for (int i = 1; i < N - 1; i++) { // boundaries are fixed and thus skipped
        do_loop(t, i); // do_loop(t, i) = (A(t, i-1) + 2*A(t, i) + A(t, i+1)) * 0.25;
    }
}
```

ISCC will identify the loop core `do_loop` with indices `[t, i]`, recognize the iteration space `[T, N]`, and be able to rearrange iteration order by some affine transformation defined by the user. For pipeline tiling, we do the following affine transformations to our loop body:

1. Let $D$ be the slant-edge size, $E$ be the bottom-edge size.

2. $[t, i] \rightarrow [t+i, t, i]$, which slants only the data dimension. This means that all points $(t, i)$ with the same value of $t + i = k$, will be calculated before all other points $(t', i')$ with $t' + i' > k$, and among these points, the smaller $t - i$ is, the earlier it is executed. The skewing aligns the data iteration order better with our desired tile shape.

3. $[t+i, t, i] \rightarrow [wb, tb, wx, tx, t, i] : \begin{align*}
    wb &= t+i / D, \\
    wx &= t+i \% D, \\
    tb &= t / E, \\
    tx &= t \% E.
\end{align*}$ In this step, we divides the skewed space into parallelograms of bottom-length $D$ and height-length $E$. Both the inter-tile and intra-tile iteration orders satisfy the data dependencies.

4. $[wb, tb, wx, tx, t, i] \rightarrow [tb, cb, t, i] : \begin{align*}
    tb &= db + ab, \\
    cb &= db.
\end{align*}$ By adding $db$ and $ab$, we revert the skewing and obtain all the diamonds on the same horizontal line, while $db$ yields them in order from left to right. This allows better parallelization, as each core can handle one set of elements with the same $cb$ per block.
**Diamond Tiling**  Diamond tiling is another common tiling methods for 1D stencil code. Given the same iteration space as defined above, the diamond tiling method splits the plane into diamonds of a fitting size $\sigma$, and loop through the iteration space one tile at a time. The lines that perform the cuts are of the form $x + t = k\sigma$ and $x - t = l\sigma$, for $k, l \in \mathbb{N}$.

Diamond tiling also provides a good opportunity for parallelization. Due to the inter-tile dependency pattern, each diamond tile requires that the three tiles underneath it be completed. Therefore diamond tiles that share the same horizontal axis, such as those colored yellow in figure 4, can be performed concurrently. In fact, due to the fact that diamond tiles can be run concurrently

In this case, in order to achieve the diamond shaped tiles we desire, we perform the following affine transformations:

1. Let $D$ be the size of tile (the number of elements on the side).

2. $[t, i] \rightarrow [t+i, t-i, t, i]$, which skews the iteration space to its side. This means that all points $(t, i)$ with the same value of $t + i = k$, will be calculated before all other points $(t', i')$ with $t' + i' > k$, and among these points, the smaller $t - i$ is, the earlier it is executed. The skewing aligns the data iteration order better with our desired tile shape.

3. $[t+i, t-i, t, i] \rightarrow [db, ab, dx, ax, t, i]$ : $db = t+i / D$, $dx = t+i \% D$, $ab = t-i / D$, $ax = t-i \% D$. In this step, we divides the skewed space into squares of side-length $D$, which produces the diamonds we want. Both the inter-tile and intra-tile iteration orders satisfy the data dependencies.

4. $[db, ab, dx, ax, t, i] \rightarrow [tb, cb, t, i]$ : $tb = db + ab$, $cb = db$. By adding $db$ and $ab$, we revert the skewing and obtain all the diamonds on the same horizontal line, while $db$ yields them in order from left to right. This allows better parallelization, as each core can handle one set of elements with the same $cb$ per block.

Note that in the last step, after obtaining the loop order for the diamond tiles, we threw away $dx$ and $ax$, which recorded the order of iteration within diamonds after skewing. If we did not, and kept $[tb, cb, dx, ax, t, i]$ as the transformation result, we would obtain this iteration order:
Instead we use the original $t$ and $i$ for intra-tile ordering. Our experiments confirm [Won02] that this choice provides a considerable performance boost, as while the skewed iteration order is slightly better in terms of cache locality, the benefit is far out-weighted by the time cost wasted on repeated changing loops, visible above as the large amount of ‘zig-zagging’.

**Figure 7:** Iteration order with Skewing

**Figure 8:** Iteration order without Skewing

### 5.2 Two Level Tiling

Two level tiling is the practice of performing tiling again inside tiles. The two kinds of tiling above produces the four combinations of possible two level tiling we examine in this paper. When experimenting, to ensure that we minimize boundary cases, we define additional parameters $\pi$ and $\rho$ such that the outer tiles are created by lines of the form $x + t = k\pi\sigma$, $x - t = l\pi\sigma$, or $t = j\rho\tau$, and we use the lines as defined in the one-level cases for inner tiles. The intention of such complicated tiling is that perhaps we can align the outer tile with larger caches, and have the inner tiles be the size of smaller caches, so as to fully exploit the benefit of cache locality. We also
show graphs for them, where the green lines mark the boundaries of the inner-level tiles.

**Diamond-Diamond Tiling** There are two ways of thinking about two-level tiling. The first is to create large tiles, then subdivide each tile into smaller sub-tiles. The second is to create small tiles, then decide some kind of inter-tile iteration order so that a group of small tiles form a large tile.

1. Let $D$ be the inner diamond edge size, $E$ be the number of inner diamonds each outer diamond has.

2. $[t, i] \rightarrow [t+i, t-i, t, i]$

3. $[t+i, t-i, t, i] \rightarrow [db, ab, dx, ax, t, i] : \begin{align*}
    db &= t+i / D, \\
    dx &= t+i \% D, \\
    ab &= t-i / D, \\
    ax &= t-i \% D
\end{align*}$

4. $[db, ab, dx, ax, t, i] \rightarrow [dbb, abb, dbx, abx, t, i] : \begin{align*}
    dbb &= db / E, \\
    abb &= ab / E, \\
    dbx &= db \% E, \\
    abx &= ab \% E
\end{align*}$. This step determines the inter-tile execution order, fitting groups of smaller tiles into larger tiles, consisting of $E$ small tiles on each edge of the diamond.

5. $[dbb, abb, dbx, abx, t, i] \rightarrow [tbb, cbb, dbx, abx, t, i] : \begin{align*}
    tbb &= dbb + abb, \\
    cbb &= dbb
\end{align*}$. Again, this step parallelizes larger diamonds on the same horizontal line.

![Figure 9: Diamond two-level tiling](image)

**Pipeline-Pipeline** Like Diamond-Diamond, Pipeline-Pipeline groups a existing pipeline tiling into larger pipeline tiles.
1. Let \( D \) and \( E \) be the inner pipeline edge size, \( F \) and \( G \) be the number of inner pipelines each outer pipeline has on the sides.

2. \([t, i] \rightarrow [t+i, t, i]\)

3. \([t+i, t, i] \rightarrow [wb, tb, t, i] : \ wb = t+i / D, \ tb = t / E.\)

4. \([wb, tb, wx, tx, t, i] \rightarrow [wbb, tbb, wbx, tbx, t, i] : \ wbb = wb / F, \ wbx = wb \% F, \ tbb = tb / G, \ tbx = tb \% G.\)

5. \([wbb, tbb, wbx, tbx, t, i] \rightarrow [pbb, cbb, wbx, tbx, t, i] : \ pbb = wbb + tbb, \ cbb = wbb.\)

The outer core again is rescheduled for parallelization along the top-left to bottom-right wavefront. The inner tiles are not dealt with the same way as there are not much point optimizing the iteration order for parallelization if they only receive the attention of one CPU core. For machines with more CPU cores, it might be more beneficial to do one-level tiling but with smaller tile sizes.

\[\text{Figure 10: Pipeline two-level tiling}\]

**Diamond-Pipeline**  
Diamond-pipeline is the tiling scheme that first creates large diamond tiles as an outer shells, then tile the interior of individual diamond tiles with pipeline tiling. This tiling scheme is done in the hope of combining the performance of pipeline tiling with the concurrent start of diamond tiling, while also hopefully optimize cache locality. This does run the risk of trading loop complexity for cache performance, however, and at some point the trade-off will be too great to be worth the effort.

1. Let \( DE \) be the outer diamond edge size, \( D \) be inner pipeline edge sizes.
2. \([t, i] \rightarrow [t+i, t-i, t, i]\)

3. \([t+i, t-i, t, i] \rightarrow [dbb, abb, t+i, t, i] : \) \(dbb = t+i / DE, abb = t-i / DE\).

4. \([dbb, abb, t+i, t, i] \rightarrow [dbb, abb, wb, tb, t, i] : \) \(wb = t+i / D, tb = t / D\).

5. \([dbb, abb, wb, tb, t, i] \rightarrow [pbb, cbb, wb, tb, t, i] : \) \(pbb = dbb + abb, cbb = dbb\).

**Figure 11:** Diamond-Pipeline tiling

**Pipeline-Diamond**  Just like Diamond-pipeline tiling, but reversed in terms of the shape of outer and inner level tiles.

1. Let \(DF\) and \(EF\) be the outer pipeline edge size, \(F\) be inner diamond edge size.

2. \([t, i] \rightarrow [t+i, t-i, t, i]\)

3. \([t+i, t-i, t, i] \rightarrow [wbb, tbb, t+i, t-i, t, i] : \) \(wbb = t+i / DF, tbb = t / EF\).

4. \([wbb, tbb, t+i, t-i, t, i] \rightarrow [wbb, tbb, db, ab, t, i] : \) \(db = t+i / F, ab = t-i / F\).

5. \([wbb, tbb, db, ab, t, i] \rightarrow [pbb, cbb, db, ab, t, i] : \) \(pbb = wbb + tbb, cbb = wbb\).
Figure 12: Pipeline-Diamond tiling
6 Other Factors

Number of Cores  Theoretically, it is almost always better to use more cores if possible, and for our four-core machine, we should expect four-times the performance of the best single core performance with proper tiling. Practically, however, if the tile size is chosen inappropriately, it is conceivable that not all cores will be active at a time and provide less benefit than ideal. Also, if the performance of \( k \) cores is less than \( k \) times that of a single core, we may consider running separate programs each on their own cores, instead of running each of them on multiple cores in sequence.

Intra-tile and Inter-tile Iteration Order  While the central concept of tiling is changing the iteration order, as each tile is supposedly self-contained in terms of cache usage, we expect that inter-tile iteration order would not matter as much as some other factors. As for intra-tile iteration order, the question should be partially answered by the effect of two-level tiling.

One important factor is the balance between loop complexity and cache performance. As mentioned before, a data-skewed intra-tile iteration order can be slightly more beneficial to cache performance than that of a non-skewed one, but the trade-off, which comes in the form of excessive amount of loops, could greatly harm the overall performance.

Loop Unrolling  denotes the loop optimization technique that trades binary size for performance by explicitly spelling out more steps in a loop. For instance, the loop

```plaintext
for (int i=0; i<50; i++) {
    do_something(i);
}
```

can be unrolled into

```plaintext
for (int i=0; i<50; i+=5) {
    do_something(i);
    do_something(i+1);
    do_something(i+2);
    do_something(i+3);
    do_something(i+4);
}
```
The new code will compile into longer binary code, but at run time, less time is spent branching, and is devoted to actual calculation instead. The unrolling process is handled automatically by the compiler.

**Compiler**  Translators are traitors, and this also carries over to the digital world. A same piece of code can be compiled into rather different assembly code by two different compilers, and in turn have drastically different results.
7 Tests

In order to test the list of factors, we have performed these tests.

**Problem Parameters** We have performed tests where \( T = 100000 \), \( N \in \{1000, 2000, 5000, 10000, 20000, 50000\} \), and where \( N = 100000 \), \( T \in \{1000, 2000, 5000, 10000, 20000, 50000\} \), and where \( T = N \in \{1000, 2000, 5000, 10000, 20000, 50000\} \).

We also performed tests where \( T \) or \( N \) is 1000000, while the other factor is 50000, 100000 or 1000000.

**Number of Cores** For most of the above mentioned tests, we have performed the same tests on single-core and multi-core versions. In some cases the single core test results might not be present due to time concerns.

**Levels of tiling, tile shapes** We run all our tests on seven different versions: non-tiled, diamond-tiling, pipeline-tiling, and the four two-level tilings.

**Tile sizes** We have run tests on a varying group of tests, with the inner level tile sizes in \( \{4, 16, 40, 160, 400, 1600, 2000, 4000\} \), and the number of first level tiles on each edge of a second level tile in \( \{1, 4, 10, 20, 40, 100\} \).

**Intra-tile execution order** Some tests are run separately to test these on sufficiently large problems, and on known good tile sizes, some with skewed iteration order, some without.

**Inter-tile execution order** Tests are done in similar fashion to above.

**Loop Optimization** This is controlled via compiler options, so some separate tests are run with some of the compiler optimization options turned off.

**Loop Code Style** We have run some preliminary tests with the same parameters on multiple code versions. In almost all cases the twocalc version performs the best.

**Static/Parametric tile sizes** This is left untested.

**Compiler** We used ISCC to generate code for transformations given earlier. We have tested compiling most of the main tests using GCC-4.8.5 on Ubuntu 14.04.1. The flags used include
-O3 -static -fopenmp

We have also performed some tests using Intel ICC-16.0.2 (compatible with GCC-4.8.0), with the flags

-03 -Wall -Werror -funroll-loops -xHost -fno-alias -fno-fnalias
-fp-model precise -ipo
8 Results and Analysis

We collected the data and placed them in a MySQL database for easier examination. Due to the large amount of data (roughly 20000 entries), most of the data is not presented here other than via some graphs. However the code used for the tests and the resulting database is available upon request, and we plan to make it a downloadable appendix of a tech-report.

Problem Parameters  On ‘large’ problems — that is, problems where both T and N are on or above the scale of 1e4 — the problem parameters do not see to play a central role. We do note that, under specific circumstances, the overall shape of the problem can heavily bias towards a certain tiling. While pipeline tiling is the more flexible and efficient tiling in comparison to Diamonds, to achieve optimal performance, it does require that the time parameter be several times more than the tile size. This is due to the fact that while diamond-tiled code runs concurrently on a horizontal line, pipeline does so on a slanted edge.

To show this, we queried the database for situations where the parameter T is less than four times the tile size, and obtained the following result:

<table>
<thead>
<tr>
<th>sigma</th>
<th>prob-size</th>
<th>time-size</th>
<th>rate</th>
<th>ratio</th>
<th>limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1600</td>
<td>100000</td>
<td>5000</td>
<td>10612.6</td>
<td>3.032171316964286</td>
<td>3.1250</td>
</tr>
<tr>
<td>1600</td>
<td>100000</td>
<td>5000</td>
<td>10492.6</td>
<td>2.9978856026785716</td>
<td>3.1250</td>
</tr>
<tr>
<td>1600</td>
<td>100000</td>
<td>5000</td>
<td>10412.5</td>
<td>2.975</td>
<td>3.1250</td>
</tr>
<tr>
<td>1600</td>
<td>100000</td>
<td>5000</td>
<td>10390.8</td>
<td>2.96879994196429</td>
<td>3.1250</td>
</tr>
</tbody>
</table>

Table 1: Small T compared to Tile Size

where ratio denotes the ratio of the actual performance against the expected performance of a single core running at a decent rate (3500 MF), juxtaposed with the number of tiles that can fit in vertically at once, denoted by limit, sorted by the maximum performance. We see that the results match one another, staying below the limit. With diamonds, however, we have
Table 2: Small T compared to Tile Size, w/ Diamonds

where diamonds seem to perform well regardless of the limitations posed by T. While the height of the problem is only enough for one tile, it runs perfectly well with all four cores running most of the time.

In the events of a very small N, however, due to the way both algorithms work, pipeline falls further behind due to the slanted parallelism, while diamonds, with its simpler parallelism, seemingly runs as well as the problem allows it to run.

Table 3: Pipeline, Restricted by N

Thus it can be said with some confidence that, in the events of extreme problem shapes such as these, with either T or N vastly smaller than the other, diamond tiling is the more appropriate method to deliver high performance. For single core, however, as parallelism is not invoked, the performance is not greatly impacted by the shape.

Loop Body  By investigating among some 30000 entries in the database for cases where all else being equal, and T and N are at least 10000, 7141 entries have Twocalc running strictly faster than Swaprows, while only 586 were the other way around.
Below is a table showing the extrema of the ratio between Twocalc and Swaprows’ performances.

<table>
<thead>
<tr>
<th>sigma</th>
<th>pi</th>
<th>time-size</th>
<th>prob-size</th>
<th>sr-rate</th>
<th>tc-rate</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>16</td>
<td>40</td>
<td>100000</td>
<td>50000</td>
<td>7638.84</td>
<td>3859.32</td>
<td>0.5052</td>
</tr>
<tr>
<td>16</td>
<td>40</td>
<td>50000</td>
<td>100000</td>
<td>7571.08</td>
<td>3823.24</td>
<td>0.5049</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>100000</td>
<td>100000</td>
<td>7847.69</td>
<td>3954.28</td>
<td>0.5038</td>
</tr>
<tr>
<td>16</td>
<td>40</td>
<td>100000</td>
<td>100000</td>
<td>7681.67</td>
<td>3827.29</td>
<td>0.4982</td>
</tr>
</tbody>
</table>

Table 5: Worst ratio, Twocalc : Swaprows

We notice that while both have their share of good and bads, the best performance for Twocalc does not occur at the point where it has the greatest advantage: when Twocalc performs at above 14000 MF (which we see is quite achievable from other queries), the performance advantage range only from 17% to 40%. We also note that while sometimes Swaprows perform twice as good as Twocalc, neither performs above 10000 MF, meaning that they do not have their potential fully exploited.

Likewise, we performed performance comparisons to Swaprows and Copy. There are 257 cases where Copy out-performs Swaprows, but 5119 cases that are otherwise.

<table>
<thead>
<tr>
<th>sigma</th>
<th>pi</th>
<th>time-size</th>
<th>prob-size</th>
<th>cp-rate</th>
<th>sr-rate</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>20</td>
<td>50</td>
<td>50000</td>
<td>100000</td>
<td>6339.18</td>
<td>4622.25</td>
<td>2.2296</td>
</tr>
<tr>
<td>20</td>
<td>50</td>
<td>100000</td>
<td>100000</td>
<td>4620.66</td>
<td>10306.3</td>
<td>2.2304</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
<td>100000</td>
<td>100000</td>
<td>4639.48</td>
<td>10373.3</td>
<td>2.2358</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
<td>50000</td>
<td>100000</td>
<td>4520.9</td>
<td>10372.2</td>
<td>2.2942</td>
</tr>
</tbody>
</table>

Table 6: Best ratio, Twocalc : Swaprows

<table>
<thead>
<tr>
<th>sigma</th>
<th>pi</th>
<th>time-size</th>
<th>prob-size</th>
<th>cp-rate</th>
<th>sr-rate</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>20</td>
<td>50</td>
<td>50000</td>
<td>100000</td>
<td>6339.18</td>
<td>4622.2</td>
<td>0.7291</td>
</tr>
<tr>
<td>20</td>
<td>50</td>
<td>100000</td>
<td>100000</td>
<td>6342.41</td>
<td>4620.6</td>
<td>0.7285</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
<td>100000</td>
<td>100000</td>
<td>6496.25</td>
<td>4639.4</td>
<td>0.7141</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
<td>50000</td>
<td>100000</td>
<td>6498.91</td>
<td>4521.</td>
<td>0.6956</td>
</tr>
</tbody>
</table>

Table 7: Worst ratio, Swaprows : Copy
We see that Copy performs better occasionally, but usually performs worse. We also see again that when the ratio between Swaprows and Copy’s performance is maximized, Swaprows’ performance is not maximized. We see that if we query for when both have a performance above 7000MF (average two cores running), the range of ratio of Swaprows’ performance over Copy’s is only from 0.83 to 1.73. Also worth noting is that if we look at cases where Copy performs above 10000MF, it never performs better than Swaprows.

We believe that this evidence enough that in almost all circumstances, it is favorable to use Twocalc over Swaprows, and in turn over Copy.

### Number of Cores

We queried our database for all situations where all else being equal, and T and N at least 10000, for the ratio between the performance of the single-core version and that of the multi-core version, with single-core performing better. We see that sometimes the multi-core version could be worse than the single core versions. This seem abysmal, but querying for also the tiling for these runs show that all the worst performers are two-level tiling, and all the pi-values, which decides how many base-level tiles there are in a outer-level tile. Essentially, it defines humongous outer-level tile, and since the parallelization schemes one core to each outer-level tile, only one CPU is running in those ‘multi-core’ runs. If we instead query for cases where the tiling is not one of the four two-level tilings, there are no cases where single-core code out-performs multi-core code.
On the other hand, the ratio of performance of multi-core against that of single-core, where all else are equal but multi-core performs better, ranges from about identical to 4.41:1, a value even above the number of cores there are in the machine. It seems not to be the case that the computer miraculously gained extra (portion of) cores, but rather the single core version performs miserably, while the multi-core version avoids the pitfall to some extent. The best performance of multi-core versions do not occur at the point of highest advantage against single-core. When both are performing well, we see that the ratio between the two drop closer to 4, and is more explainable as experimental variation.

<table>
<thead>
<tr>
<th>tiling</th>
<th>sigma</th>
<th>pi</th>
<th>time-size</th>
<th>prob-size</th>
<th>sc-rate</th>
<th>mc-rate</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>pipelined-ts-two-level</td>
<td>1000</td>
<td>100</td>
<td>100000</td>
<td>100000</td>
<td>2935.78</td>
<td>1838.58</td>
<td>0.6262</td>
</tr>
<tr>
<td>pipelined-ts-two-level</td>
<td>1000</td>
<td>100</td>
<td>50000</td>
<td>100000</td>
<td>2989.69</td>
<td>1843.85</td>
<td>0.6167</td>
</tr>
<tr>
<td>pipelined-ts-two-level</td>
<td>1000</td>
<td>100</td>
<td>100000</td>
<td>50000</td>
<td>2997.03</td>
<td>1844.35</td>
<td>0.6153</td>
</tr>
<tr>
<td>pipelined-ts-two-level</td>
<td>1000</td>
<td>100</td>
<td>50000</td>
<td>50000</td>
<td>2994.94</td>
<td>1842.93</td>
<td>0.6153</td>
</tr>
<tr>
<td>pipelined-ts-two-level</td>
<td>1000</td>
<td>100</td>
<td>100000</td>
<td>10000</td>
<td>2993.3</td>
<td>1839.84</td>
<td>0.6146</td>
</tr>
</tbody>
</table>

Table 9: Worst performance, Multi-core : Single-core

<table>
<thead>
<tr>
<th>tiling</th>
<th>sigma</th>
<th>pi</th>
<th>sc-rate</th>
<th>mc-rate</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>pipelined-ts-two-level</td>
<td>1000</td>
<td>100</td>
<td>2935.78</td>
<td>1838.58</td>
<td>0.6262</td>
</tr>
<tr>
<td>pipelined-ts-two-level</td>
<td>1000</td>
<td>100</td>
<td>2989.69</td>
<td>1843.85</td>
<td>0.6167</td>
</tr>
<tr>
<td>pipelined-ts-two-level</td>
<td>1000</td>
<td>100</td>
<td>2997.03</td>
<td>1844.35</td>
<td>0.6153</td>
</tr>
<tr>
<td>pipelined-ts-two-level</td>
<td>1000</td>
<td>100</td>
<td>2994.94</td>
<td>1842.93</td>
<td>0.6153</td>
</tr>
<tr>
<td>pipelined-ts-two-level</td>
<td>1000</td>
<td>100</td>
<td>2993.3</td>
<td>1839.84</td>
<td>0.6146</td>
</tr>
</tbody>
</table>

Table 10: Worst performance, Multi-core : Single-core with tiling info

<table>
<thead>
<tr>
<th>sigma</th>
<th>pi</th>
<th>time-size</th>
<th>prob-size</th>
<th>sc-rate</th>
<th>mc-rate</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>50</td>
<td>50000</td>
<td>50000</td>
<td>2024.16</td>
<td>8755.89</td>
<td>4.3256</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>50000</td>
<td>50000</td>
<td>1992.01</td>
<td>8725.11</td>
<td>4.3800</td>
</tr>
<tr>
<td>200</td>
<td>100</td>
<td>100000</td>
<td>50000</td>
<td>1992.01</td>
<td>8777.69</td>
<td>4.4064</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>50000</td>
<td>50000</td>
<td>1993.08</td>
<td>8791.58</td>
<td>4.4110</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>100000</td>
<td>50000</td>
<td>1990.31</td>
<td>8784.19</td>
<td>4.4134</td>
</tr>
</tbody>
</table>

Table 11: Best performance, Multi-core : Single-core
Following the discussion of the previous portion, we see that it is possible that under certain circumstances, simply adding CPU cores will not improve the performance by much, as the performance is bottlenecked by the parallelism of the tiles. In general, however, when all else being equal, the performance of the multi-core version is about $k$ times that of the single-core version, where $k$ is the number of cores present in the machine.

The effect of inter-core communication is not quite testable, due to only testing on a single type of CPU.

**Level of Tiling**  For the effect of second level tiling, we queried instances of multi-core and single-core performance of pipeline-tiled code and two-level pipeline-tiled code (with a non-trivial second level tile size). Querying the database for such cases revealed that in the best case, the performance improvement obtained by using two-level tiling maxes out at 13% above single-level code where all else are equal. In the worst case, however, two-level tiling performs a whopping 83% worse than one-level tiling. However, in the worst cases, the pi-value is chosen poorly at 100 or similarly big numbers.
<table>
<thead>
<tr>
<th>sigma</th>
<th>pi</th>
<th>time-size</th>
<th>prob-size</th>
<th>l1-rate</th>
<th>l2-rate</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>10000</td>
<td>10000</td>
<td>8675.21</td>
<td>10233.1</td>
<td>1.1795</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>10000</td>
<td>10000</td>
<td>9509.48</td>
<td>11065.9</td>
<td>1.1636</td>
</tr>
<tr>
<td>1600</td>
<td>1</td>
<td>10000</td>
<td>10000</td>
<td>9981.77</td>
<td>11161.6</td>
<td>1.1181</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>50000</td>
<td>1000000</td>
<td>1701.35</td>
<td>1891.8</td>
<td>1.1119</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
<td>50000</td>
<td>50000</td>
<td>11136.9</td>
<td>1821.32</td>
<td>0.1635</td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
<td>1000000</td>
<td>50000</td>
<td>11147.5</td>
<td>1822.96</td>
<td>0.1635</td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
<td>20000</td>
<td>50000</td>
<td>11123.3</td>
<td>1818.07</td>
<td>0.1634</td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
<td>10000</td>
<td>20000</td>
<td>10966.2</td>
<td>1779.89</td>
<td>0.1623</td>
</tr>
</tbody>
</table>

Table 14: Best to worst, ratio of Twocalc two-level : one-level diamonds

<table>
<thead>
<tr>
<th>sigma</th>
<th>pi</th>
<th>time-size</th>
<th>prob-size</th>
<th>l1-rate</th>
<th>l2-rate</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>20000</td>
<td>100000</td>
<td>10320.3</td>
<td>11804.2</td>
<td>1.1437</td>
</tr>
<tr>
<td>40</td>
<td>4</td>
<td>10000</td>
<td>10000</td>
<td>8542.92</td>
<td>9766.33</td>
<td>1.1432</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>20000</td>
<td>100000</td>
<td>12052.3</td>
<td>13753.5</td>
<td>1.1411</td>
</tr>
<tr>
<td>4000</td>
<td>1</td>
<td>10000</td>
<td>100000</td>
<td>3555.29</td>
<td>3919.25</td>
<td>1.1023</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>10000</td>
<td>50000</td>
<td>10638.6</td>
<td>1823.91</td>
<td>0.1714</td>
</tr>
<tr>
<td>1000</td>
<td>50</td>
<td>50000</td>
<td>1000000</td>
<td>10950.6</td>
<td>1845.56</td>
<td>0.1685</td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
<td>50000</td>
<td>1000000</td>
<td>10951.6</td>
<td>1845.61</td>
<td>0.1685</td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
<td>100000</td>
<td>1000000</td>
<td>11034.5</td>
<td>1845.5</td>
<td>0.1672</td>
</tr>
</tbody>
</table>

Table 15: Best to worst, ratio of Twocalc two-level : one-level pipeline

If we ask for cases where the second level tile count is no greater than 20, we see a very slightly improvement for the worst case scenario. The improvement for pipeline is very minor in comparison to that of diamonds', since pipelines are more sensitive the the size of the problem due to how its parallelization scales.
Tile Shapes  We queried for the top performers of all test cases, and all the tests that achieved a performance of 15000 MF or more have the inner most tile-shape of pipeline: the more uniform loop length inside the tiles certainly help improve the performance.  All else being equal, apart from tile sizes, the performance of diamonds peaks at 14400 MF for T=100000, N=100000, while pipeline peaks at 15500 MF; diamonds 14360 MF for T=10000, N=100000, while pipeline does 15000 MF; for T=100000, N=10000, diamond does 12700 MF, while pipeline does 14900 MF.  For very large N (1e8) and moderate T, diamond does 14500 MF, while pipeline does 15000 MF.  It seems that pipeline is overall better by about 3% - 9%, if both are performing well.

Again, in cases where the problem is particularly ‘flat’, i.e., when N is very large, and T is as low as 1000 or 2000, thus nearing tile-sizes, diamond-tiling perform much better than pipeline-tiling.  This would be due to its parallelization scheme.

Table 16: Best to worst, ratio of Twocalc two-level : one-level diamonds, Pi < 20

<table>
<thead>
<tr>
<th>sigma</th>
<th>pi</th>
<th>time-size</th>
<th>prob-size</th>
<th>l1-rate</th>
<th>l2-rate</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>10000</td>
<td>10000</td>
<td>8675.21</td>
<td>10233.1</td>
<td>1.1795</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>10000</td>
<td>10000</td>
<td>9509.48</td>
<td>11065.9</td>
<td>1.1636</td>
</tr>
<tr>
<td>1600</td>
<td>1</td>
<td>10000</td>
<td>10000</td>
<td>9981.77</td>
<td>11161.6</td>
<td>1.1181</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>50000</td>
<td>1000000</td>
<td>1701.35</td>
<td>1891.8</td>
<td>1.1119</td>
</tr>
</tbody>
</table>

Table 17: Best to worst, ratio of Twocalc two-level : one-level pipeline, Pi < 20

<table>
<thead>
<tr>
<th>sigma</th>
<th>pi</th>
<th>time-size</th>
<th>prob-size</th>
<th>l1-rate</th>
<th>l2-rate</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>20000</td>
<td>100000</td>
<td>10320.3</td>
<td>11804.2</td>
<td>1.1437</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>20000</td>
<td>100000</td>
<td>12052.3</td>
<td>13753.5</td>
<td>1.1411</td>
</tr>
<tr>
<td>1600</td>
<td>1</td>
<td>20000</td>
<td>100000</td>
<td>11350.5</td>
<td>12386.2</td>
<td>1.0912</td>
</tr>
<tr>
<td>1600</td>
<td>1</td>
<td>20000</td>
<td>100000</td>
<td>9785.93</td>
<td>10605.4</td>
<td>1.0837</td>
</tr>
</tbody>
</table>

Tile Shapes  We queried for the top performers of all test cases, and all the tests that achieved a performance of 15000 MF or more have the inner most tile-shape of pipeline: the more uniform loop length inside the tiles certainly help improve the performance.  All else being equal, apart from tile sizes, the performance of diamonds peaks at 14400 MF for T=100000, N=100000, while pipeline peaks at 15500 MF; diamonds 14360 MF for T=10000, N=100000, while pipeline does 15000 MF; for T=100000, N=10000, diamond does 12700 MF, while pipeline does 14900 MF.  For very large N (1e8) and moderate T, diamond does 14500 MF, while pipeline does 15000 MF.  It seems that pipeline is overall better by about 3% - 9%, if both are performing well.

Again, in cases where the problem is particularly ‘flat’, i.e., when N is very large, and T is as low as 1000 or 2000, thus nearing tile-sizes, diamond-tiling perform much better than pipeline-tiling.  This would be due to its parallelization scheme.
that stretches primarily in one dimension only. Pipeline’s parallelization scheme, in comparison, runs diagonally, extending into the time-dimension, therefore if it is not given sufficient space in both dimensions, it will not run well in parallel.

**Tile Sizes** Sorting the performance of all diamond one-level tests by performance show that $\sigma = 1000$ as consistently the best performing tile size, followed by: 1600, 2000, 400, 200, 4000, 160, 100, 40 and etc. The same trend holds all variants of the loop body.

For diamond two-level, the optimal tile size is slightly more flexible: again 1000 is the optimal base tile size. There does not seem to be much correlation between second level tile size and the performance, however.

**Intra-tile Execution Order** This is one of the less debatable factors. For two-level tiling, the inner-level tiling has a much greater effect on the performance than the outer level, so the answer would most certainly coincide with that for one-level tiling. For one-level diamond tiling, we can either follow the normal iteration flow of $[t, i]$, or the skewed iteration order $[t + i, t - i]$. We have discussed the effect of these choices, but we would like to back up our claim with actual results: For $T=50000$, $N=50000$ diamonds two-level code, all else being equal, the skewed version has a performance of 942 MF, while the unskewed version has 2948 MF. For $T=10000$, $N=1000000$, the skewed does 940 MF, the unskewed does 2945 MF. For $T=1000000$, $N=10000$, the skewed does 943 MF, the unskewed does 2940 MF. Even at optimal tile-size ($\sigma = 1000, \pi = 10$), the unskewed does only 960 MF, while the unskewed does 3600 MF.

**Compiler** We ran a small set of tests using the untiled version with both ICC and GCC, using the following special scenarios:

1. large problems, where $N = T = 100000$
2. ‘slab’ problems, where $T = 1000$ and $N = 100000$
3. ‘pillar’ problem, where $T = 1000000$, and $N = 1000$.

We are specially interested in the pillar case, where the problem’s data dimension is small enough to fit in L1 CPU cache. We hope that by using simpler code, we can obtain a fairer comparison. First, we have the single core performance of code compiled by ICC and GCC in each scenario:
<table>
<thead>
<tr>
<th>Test</th>
<th>Twocalc(MF)</th>
<th>Swaprows(MF)</th>
<th>Copy(MF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large, GCC</td>
<td>3307</td>
<td>3385</td>
<td>1827</td>
</tr>
<tr>
<td>Large, ICC</td>
<td>3101</td>
<td>3287</td>
<td>1850</td>
</tr>
<tr>
<td>Slab, GCC</td>
<td>3109</td>
<td>3143</td>
<td>1832</td>
</tr>
<tr>
<td>Slab, ICC</td>
<td>2970</td>
<td>3178</td>
<td>1812</td>
</tr>
<tr>
<td>Pillar, GCC</td>
<td>3470</td>
<td>3517</td>
<td>2839</td>
</tr>
<tr>
<td>Pillar, ICC</td>
<td>2532</td>
<td>3533</td>
<td>2799</td>
</tr>
</tbody>
</table>

**Table 18:** Single-core Original(ISCC) Performance

And the multi-core version:

<table>
<thead>
<tr>
<th>Test</th>
<th>Twocalc(MF)</th>
<th>Swaprows(MF)</th>
<th>Copy(MF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large, GCC</td>
<td>9057</td>
<td>8671</td>
<td>4581</td>
</tr>
<tr>
<td>Large, ICC</td>
<td>4351</td>
<td>8937</td>
<td>4647</td>
</tr>
<tr>
<td>Slab, GCC</td>
<td>8658</td>
<td>8691</td>
<td>4477</td>
</tr>
<tr>
<td>Slab, ICC</td>
<td>4291</td>
<td>8588</td>
<td>4680</td>
</tr>
<tr>
<td>Pillar, GCC</td>
<td>3279</td>
<td>3066</td>
<td>1727</td>
</tr>
<tr>
<td>Pillar, ICC</td>
<td>1881</td>
<td>2986</td>
<td>1714</td>
</tr>
</tbody>
</table>

**Table 19:** Multi-core Original(ISCC) Performance

This shows that for single-core tests and no tiling, ICC and GCC performs on-par with one another. Note that while sometimes they differ by up to 200MF, the test results for a single test can also vary by that amount. The only true difference would be for single core pillar, where GCC performs more than a whopping 30% better than ICC when using Twocalc.

For multi-core, the general trend is that GCC and ICC performs about the same for Swaprows and Copy, but ICC performs much worse when it comes to Twocalc, which is the main focus of our test runs. For this reason, we elected to primarily test with GCC instead of ICC.

We also performed some tests using the original code\(^6\), on Twocalc.

\(^6\)the source code without applying any code transformation using ISCC, while the ‘untiled’ code tested above is obtained after passing it through ISCC, albeit without applying any transformations.
<table>
<thead>
<tr>
<th>Test</th>
<th>Rate(MF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large, GCC</td>
<td>3396</td>
</tr>
<tr>
<td>Large, ICC</td>
<td>3069</td>
</tr>
<tr>
<td>Slab, GCC</td>
<td>3339</td>
</tr>
<tr>
<td>Slab, ICC</td>
<td>3020</td>
</tr>
<tr>
<td>Pillar, GCC</td>
<td>3840</td>
</tr>
<tr>
<td>Pillar, ICC</td>
<td>2486</td>
</tr>
</tbody>
</table>

**Table 20**: single-core Original(w/o ISCC) Twocalc Performance

Here we see expected behavior with GCC: it performs extremely well on Pillar due to the data fitting inside cache, while the rest still performs decently. ICC, however, does not perform as well in all situations. We believe that this shows that GCC and ICC are capable of similar level of performance, but ICC seems to be having problem in particular with optimizing Twocalc code generated using ISCC.
9 Deviations

In addition to the effect each factor has in the best/worst case scenario, we are also exploring the degree to which we can fix a bad choice in certain factors by optimizing in other factors. Essentially, this section attempts to answer this question: what if we have to use the Swaprows version, instead of Twocalc, for a problem? How much performance do we lose for each variable dimension that we misstep in?

We are restricting the number of variables due to the sheer amount of possible combinations of all the dimensions. We only looking at loop body code style, tilings ‘diamonds’, ‘pipeline’, and a small set of tile-sizes, for a given typical problem of non-trivial size in both dimensions. We are examining the performance drop from by moving away from optimal choices in a dimension(called a ‘misstep’ in that dimension), and possible performance regains by changing to more suitable tile sizes(called a ‘suboptimal’ in that dimension). Again, we look at the performance variations in six different scenarios: large problems (that is at least 50000 on each dimensions), slab problems (small T but large N) and pillar problems (small N but large T), on both multi-core and single core versions.

**Multi-core Large**  After the query, we found out that the overall best performing test is Twocalc, with pipeline two-level tiling, the performance of which deteriorates somewhat for all possible missteps. Even allowing for changing tile sizes after, none of the suboptimal versions perform above it, though one-level pipeline does become close enough that the difference in performance can be ignored (less than 1%). Diamond-based tilings perform slightly worse by 10-15%, but are still decent in comparison. We illustrate these facts with the following graph (Fig. 13), where we compare the results of sub-optimals and the improvement from the missteps to the optimal performance.
Multi-core Slab  The slab is defined as a problem with very large N, but much smaller T. In our case we used T = 1000 and N = 100000. We found that the best performing variation is Twocalc, with diamonds tiling (displays as two level, but second level tile count is 1). That said, both two level diamonds and hybrid diamond-pipeline performs well, while all other pipeline-based tiling performs very badly using the same set of tile sizes, though the performance issue can be alleviated significantly by using more appropriate tile sizes. (See Fig. 14)

While it may seem strange that diamond-pipeline performs well despite being a pipeline based tiling, one should remember that each small pipeline tile is bounded by a larger diamond tile, which defines its concurrent behavior. The outer tile (and its inter-tile iteration order) thus avoids the pipeline start-up problem and provides very good performance\textsuperscript{7}.

\textsuperscript{7}In the graph, copy-diamonds-two-level displays as all orange as we did not obtain data for that particular tile size and problem setting; the same goes for other graphs with that feature.
Multi-core Pillar  The pillar is the opposite of a slab; it is a problem with a very large T and small T. In our case we used $T = 100000$ and $N = 1000$. The best performing variation is Twocalc pipeline one-level, while twocalc diamonds follows closely. (See Fig. 15)

One might think that result is somewhat misguided, since for such small N, it is probably better to not use any tiling. Experiment reveals however that for this scenario, non-tiled version does not allow for much parallelization, so its performance is capped to a single core at a time. For multi-core pillar, the untiled version for Twocalc, Swaprows and Copy achieves $3300$MF, $3200$MF and $1800$MF respectively. If compiled with ICC, the three achieves $2400$MF, $3100$MF and $1700$MF.
**Single-core**  While we have test results for different shapes of single-core tests as well, it seems that they do not differ all that much in comparison to multi-core tests. In three out of four of these scenarios pipeline-diamonds perform the best, while the other tilings perform imperceptibly worse even without fixing tile sizes, as long as the loop body is Twocalc; switching to Swaprows reduces performance by about 18% on average, while Copy reduces it even more. (See Fig. 16-19)

![Figure 16: Performance deviation of 1e5-by-1e3 (pillar) problems on single-core](image1)

![Figure 17: Performance deviation of 1e3-by-1e5 (slab) problems on single-core](image2)
From those tests, we gained further proof that Copy is slower than the other two loop-body versions, falling behind in all tilings tests, both in terms of missteps and sub-optimals in the loop-body dimension. Swaprows is always faster than Copy, but mostly slower than Twocalc. Sub-optimals in the loop-body dimension from twocalc can be as bad as achieving only 45% of the optimal.
10 Conclusions

Problem Size  We see that, if the problem size is too small, the performance is bad across the board, but that is hardly a concern, as ‘small’ in our case would mean ‘taking less than 1 second to finish’, and is rather trivial for our purpose. Most versions of the algorithm seem to stabilize beyond the problem parameters T=1e4 and N=1e4, so we can observe the effects of increasing either T and N beyond this range. We see that in most cases, pipeline tiling performs better, though in cases where T is very small (on the scale of level 1 cache) diamonds come out ahead.

Style of Loop Body  By the numerous results from above, we can say with some certainty that the style in which the loop body is written is never a trivial matter, and a poorly written loop body may cause a performance drop so great that no tiling or other optimization can alleviate.

Number of Cores  From the results we can see that, given 3600 MF as about the average peak performance observed from a single core version (the highest recorded is original on very small tile sizes, at 3800 MF), simply adding 3 more usable cores can, under the best of circumstances, improve the performance by almost 3 times. This improvement is somewhat to be expected: if it is possible to have the cores operate simultaneously and independently on full power, this should be the case. However to achieve such improvement the conditions required can be quite delicate: presumably, this requires that demands to the level 3 cache from all cores be satisfiable at once. Usually per-core performance does not drop much when going from multi-core to single core.

Level of Tiling  It seems that in most cases, the level of tiling does not play much of a role. Though sometimes hybrid tiling can combine the benefits of both type of tiling: diamond-pipeline hybrid tiling allows better concurrent start using the outer diamonds while mostly preserving the speed performance of pipeline tiling in the base level. Deviation from one level to two level for non-hybrid tiling usually does not cause as much of a performance drop as would a bad loop body.

Tile Shape  Again, pipeline seems to perform well in most cases other than if the problem is of the slab shape. Deviation on this front on large problems can cause about 1000MF, or about 9% of performance drop.
**Tile sizes**  We determined that both diamond and pipeline tiling seems to have a ‘sweet spot’ in terms of tile sizes to use, which from the data we gathered (that is by no means all-encompassing) seem to be about 1000 and 100 respectively. Reasonable deviation from a tiling’s optimal tile size can cause a sharp drop in performance (we recorded drops as much as 80% in comparison to the optimal). That said, this should be a relatively easy problem to solve for a auto-tiling compiler.

**Intra-tile Execution Order**  Specifically related to diamond tiling with skewed and unskewed intra-tile order, our experiments confirmed that skewed code performs much worse than unskewed code, scoring as low as 25% to 30% the rate of the latter. Therefore we did not perform many tests using this option.

**Compiler**  We performed batches of tests using both Intel’s ICC and GCC, and in all cases the GCC performs as well as, if not much better than ICC does (in the cases of Twocalc). We have determined that this is probably due to ISCC being more optimized to work with GCC.

**Closing Note**  We provide in this thesis some of the experimental results on quantified effects of factors affecting performance of stencil code. We wish to promote the idea that the tiling is not the only decisive factor in the performance of stencil code. Indeed, we see how sometimes optimization to factors like the loop body can far outweigh the benefit of optimization to the tiling scheme, and we believe that automatic compiler optimization should be able to take these factors into consideration during code compilation.
Acknowledgment

I am very grateful for professor David Wonnacott for providing me with the opportunity and, in the case of summer 2014, funds, to work with him on his research; I am indebted for all the instructions and help he kindly provided in writing this thesis. I am also thankful to professor John Dougherty for reading my drafts and providing valuable comments.

I am also thankful to my colleague in research, Tian Jin, for assistance granted in times of need.

I wish to also express my appreciation for Sanjay Rajopadhye and Yun Zou at Colorado State University for providing valuable help in the process of this research.

I also would like to thank the infrastructure in the Koshland Integrated Natural Science Center at Haverford for not causing three rains in the Computer Lab during the entire period of my research.
References


A Some MySQL Queries Used

-- Query for problem size, pipeline: slab
select sigma, prob_size, time_size, rate, rate / 3500 as ratio, time_size / sigma as lim from run where tiling = 'pipelined-ts' and omp = 1 and time_size / sigma < 4 order by rate desc;

-- Query for problem size, diamond: slab
select sigma, prob_size, time_size, rate, rate / 3500 as ratio, time_size / sigma as lim from run where tiling = 'diamonds-ti' and omp = 1 and time_size / sigma < 4 order by rate desc;

-- Query for problem size, pipeline: pillar
select sigma, prob_size, time_size, rate, rate / 3500 as ratio, prob_size / sigma as lim from run where tiling = 'pipelined-ts' and omp = 1 and prob_size / sigma < 4 order by rate desc;

-- Query for problem size, diamond: pillar
select sigma, prob_size, time_size, rate, rate / 3500 as ratio, prob_size / sigma as lim from run where tiling = 'diamonds-ti' and omp = 1 and prob_size / sigma < 4 order by rate desc;

-- Query for number of times Twocalc is better than Swaprows
select clist.sigma, clist.pi, clist.time_size, clist.prob_size, clist.id, clist.rate, tlist.id, tlist.rate from (select * from run where test_name = 'swaprows') as clist left join (select * from run where test_name = 'twocalc') as tlist on clist.tiling = tlist.tiling and clist.omp = tlist.omp and clist.sigma = tlist.sigma and clist.pi = tlist.pi and clist.time_size = tlist.time_size and clist.prob_size = tlist.prob_size and clist.time_size > 9999 and clist.prob_size > 9999 where clist.rate < tlist.rate; -- 7141

-- Query for number of times Twocalc is worse than Swaprows
select clist.sigma, clist.pi, clist.time_size, clist.prob_size, clist.id, clist.rate, tlist.id, tlist.rate from (select * from run where test_name = 'swaprows') as clist left join (select * from run where test_name = 'twocalc') as tlist on clist.tiling = tlist.tiling and clist.omp = tlist.omp and clist.sigma = tlist.sigma and clist.pi = tlist.pi and clist.time_size = tlist.time_size and clist.prob_size = tlist.prob_size and clist.time_size > 9999 and clist.prob_size > 9999 where clist.rate > tlist.rate; -- 586
-- Query for performance ratio when Twocalc is better than Swaprows

```sql
select clist.sigma, clist.pi, clist.time_size, clist.prob_size, clist.rate, tlist.rate, left(tlist.rate / clist.rate, 6) as ratio from (select * from run where test_name = 'swaprows') as clist left join (select * from run where test_name = 'twocalc') as tlist on clist.tiling = tlist.tiling and clist.omp = tlist.omp and clist.sigma = tlist.sigma and clist.pi = tlist.pi and clist.time_size = tlist.time_size and clist.prob_size = tlist.prob_size and clist.time_size > 9999 and clist.prob_size > 9999 where clist.rate < tlist.rate order by ratio asc;
```

-- Query for performance ratio when Twocalc is worse than Swaprows

```sql
select clist.sigma, clist.pi, clist.time_size, clist.prob_size, clist.rate, tlist.rate, left(tlist.rate / clist.rate, 6) as ratio from (select * from run where test_name = 'swaprows') as clist left join (select * from run where test_name = 'twocalc') as tlist on clist.tiling = tlist.tiling and clist.omp = tlist.omp and clist.sigma = tlist.sigma and clist.pi = tlist.pi and clist.time_size = tlist.time_size and clist.prob_size = tlist.prob_size and clist.time_size > 9999 and clist.prob_size > 9999 where clist.rate > tlist.rate order by ratio desc;
```

-- Query for number of times Swaprows is better than Copy

```sql
select clist.sigma, clist.pi, clist.id, clist.time_size, clist.prob_size, clist.rate, tlist.id, tlist.rate from (select * from run where test_name = 'copy') as clist left join (select * from run where test_name = 'swaprows') as tlist on clist.tiling = tlist.tiling and clist.omp = tlist.omp and clist.sigma = tlist.sigma and clist.pi = tlist.pi and clist.time_size = tlist.time_size and clist.prob_size = tlist.prob_size and clist.time_size > 9999 and clist.prob_size > 9999 where clist.rate < tlist.rate; -- 5119
```

-- Query for number of times Swaprows is worse than Copy

```sql
select clist.sigma, clist.pi, clist.id, clist.time_size, clist.prob_size, clist.rate, tlist.id, tlist.rate from (select * from run where test_name = 'copy') as clist left join (select * from run where test_name = 'swaprows') as tlist on clist.tiling = tlist.tiling and clist.omp = tlist.omp and clist.sigma = tlist.sigma and clist.pi = tlist.pi and clist.time_size = tlist.time_size and clist.prob_size = tlist.prob_size and clist.time_size > 9999 and clist.prob_size > 9999 where clist.rate > tlist.rate; -- 257
```
-- Query for performance ratio when Swaprows is better than Copy
select clist.sigma, clist.pi, clist.time_size, clist.prob_size,
clist.rate, tlist.rate, left(tlist.rate / clist.rate, 6) as ratio from
(select * from run where test_name = 'copy') as clist left join
(select * from run where test_name = 'swaprows') as tlist on
clist.tiling = tlist.tiling and clist.omp = tlist.omp and clist.sigma
= tlist.sigma and clist.pi = tlist.pi and clist.time_size =
tlist.time_size and clist.prob_size = tlist.prob_size and
clist.time_size > 9999 and clist.prob_size > 9999 where clist.rate <
tlist.rate order by ratio asc;

-- Query for performance ratio when Swaprows is worse than Copy
select clist.sigma, clist.pi, clist.time_size, clist.prob_size,
clist.rate, tlist.rate, left(tlist.rate / clist.rate, 6) as ratio from
(select * from run where test_name = 'copy') as clist left join
(select * from run where test_name = 'swaprows') as tlist on
clist.tiling = tlist.tiling and clist.omp = tlist.omp and clist.sigma
= tlist.sigma and clist.pi = tlist.pi and clist.time_size =
tlist.time_size and clist.prob_size = tlist.prob_size and
clist.time_size > 9999 and clist.prob_size > 9999 where clist.rate >
tlist.rate order by ratio desc;

-- Query for multicore vs singlecore, multicore performs better
select clist.sigma, clist.pi, clist.time_size, clist.prob_size,
clist.rate, tlist.rate, left(tlist.rate / clist.rate, 6) as ratio from
(select * from run where omp = 0) as clist left join (select * from
run where omp = 1) as tlist on clist.tiling = tlist.tiling and
clist.test_name = tlist.test_name and clist.sigma = tlist.sigma and
clist.pi = tlist.pi and clist.time_size = tlist.time_size and
clist.prob_size = tlist.prob_size and clist.time_size > 9999 and
clist.prob_size > 9999 where clist.rate < tlist.rate order by ratio
asc;

-- Query for multicore vs singlecore, single performs better
select clist.sigma, clist.pi, clist.time_size, clist.prob_size,
clist.rate, tlist.rate, left(tlist.rate / clist.rate, 6) as ratio from
(select * from run where omp = 0) as clist left join (select * from
run where omp = 1) as tlist on clist.tiling = tlist.tiling and
clist.test_name = tlist.test_name and clist.sigma = tlist.sigma and
clist.pi = tlist.pi and clist.time_size = tlist.time_size and
clist.prob_size = tlist.prob_size and clist.time_size > 9999 and
clist.prob_size = tlist.prob_size and clist.time_size > 9999 and
clist.prob_size > 9999 where clist.rate > tlist.rate order by ratio desc;

-- Query for multicore vs singlecore, multi performs better at over 10000
select clist.sigma, clist.pi, clist.time_size, clist.prob_size, clist.rate, tlist.rate, left(tlist.rate / clist.rate, 6) as ratio from (select * from run where omp = 0) as clist left join (select * from run where omp = 1) as tlist on clist.tiling = tlist.tiling and clist.test_name = tlist.test_name and clist.sigma = tlist.sigma and clist.pi = tlist.pi and clist.time_size = tlist.time_size and clist.prob_size = tlist.prob_size and clist.time_size > 9999 and clist.prob_size > 9999 and tlist.rate > 10000 where clist.rate < tlist.rate order by ratio asc;

-- Query for multicore vs singlecore, multi better, but single at least 3500
select clist.sigma, clist.pi, clist.time_size, clist.prob_size, clist.rate, tlist.rate, left(tlist.rate / clist.rate, 6) as ratio from (select * from run where omp = 0) as clist left join (select * from run where omp = 1) as tlist on clist.tiling = tlist.tiling and clist.test_name = tlist.test_name and clist.sigma = tlist.sigma and clist.pi = tlist.pi and clist.time_size = tlist.time_size and clist.prob_size = tlist.prob_size and clist.time_size > 9999 and clist.prob_size > 9999 and clist.rate > 3500 where clist.rate < tlist.rate order by ratio asc;

-- Non-twolevel multicore; hint, there are no cases.
select clist.sigma, clist.pi, clist.time_size, clist.prob_size, clist.rate, tlist.rate, left(tlist.rate / clist.rate, 6) as ratio from (select * from run where omp = 0) as clist left join (select * from run where omp = 1) as tlist on clist.tiling = tlist.tiling and clist.test_name = tlist.test_name and clist.sigma = tlist.sigma and clist.pi = tlist.pi and clist.time_size = tlist.time_size and tlist.tiling in ('orig0', 'diamonds-ti', 'pipelined-ts') and clist.prob_size = tlist.prob_size and clist.time_size > 9999 and clist.prob_size > 9999 where clist.rate > tlist.rate order by ratio desc;

-- One-level / Twolevel diamonds, one, then two
select clist.sigma, clist.pi, clist.time_size, clist.prob_size, clist.rate, tlist.rate, left(tlist.rate / clist.rate, 6) as ratio from (select * from run where tiling = 'diamonds-ti') as clist inner join
(select * from run where tiling = 'diamonds-ti-twolevel') as tlist on
clist.omp = tlist.omp and clist.test_name = tlist.test_name and
clist.sigma = tlist.sigma and clist.pi = tlist.pi and clist.time_size
= tlist.time_size and clist.prob_size = tlist.prob_size and
clist.time_size > 9999 and clist.prob_size > 9999 order by ratio desc
limit 4;

-- One-level / Twolevel pipeline, one, then two
select clist.sigma, clist.pi, clist.time_size, clist.prob_size,
clist.rate, tlist.rate, left(tlist.rate / clist.rate, 6) as ratio from
(select * from run where tiling = 'pipelined-ts') as clist inner join
(select * from run where tiling = 'pipelined-ts-twolevel') as tlist on
clist.omp = tlist.omp and clist.test_name = tlist.test_name and
clist.sigma = tlist.sigma and clist.pi = tlist.pi and clist.time_size
= tlist.time_size and clist.prob_size = tlist.prob_size and
clist.time_size > 9999 and clist.prob_size > 9999 order by ratio desc
limit 4;

-- One-level / Twolevel diamonds, one, then two, small pi
select clist.sigma, clist.pi, clist.time_size, clist.prob_size,
clist.rate, tlist.rate, left(tlist.rate / clist.rate, 6) as ratio from
(select * from run where tiling = 'diamonds-ti') as clist inner join
(select * from run where tiling = 'diamonds-ti-twolevel') as tlist on
clist.omp = tlist.omp and clist.test_name = tlist.test_name and
clist.sigma = tlist.sigma and clist.pi = tlist.pi and clist.time_size
= tlist.time_size and clist.prob_size = tlist.prob_size and
clist.time_size > 9999 and clist.prob_size > 9999 and tlist.pi < 20
order by ratio desc limit 4;

-- One-level / Twolevel pipeline, one, then two, small pi
select clist.sigma, clist.pi, clist.time_size, clist.prob_size,
clist.rate, tlist.rate, left(tlist.rate / clist.rate, 6) as ratio from
(select * from run where tiling = 'pipelined-ts') as clist inner join
(select * from run where tiling = 'pipelined-ts-twolevel') as tlist on
clist.omp = tlist.omp and clist.test_name = tlist.test_name and
clist.sigma = tlist.sigma and clist.pi = tlist.pi and clist.time_size
= tlist.time_size and clist.prob_size = tlist.prob_size and
clist.time_size > 9999 and clist.prob_size > 9999 and tlist.pi < 20
order by ratio desc limit 4;