University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Diamond Tiling: A Tiling Framework for Time-iterated Scientific Applications.

Daniel Orozco and Guang Gao

CAPSL Technical Memo 091
December 21st, 2009

§University of Delaware
{orozco, ggao}@capsl.udel.edu

Abstract

This paper fully develops Diamond Tiling, a technique to partition the computations of stencil applications such as FDTD. The Diamond Tiling technique is the result of optimizing the amount of useful computations that can be executed when a region of memory is loaded to the local memory of a multiprocessor chip.

Diamond Tiling contributes to the state of the art on time tiling techniques in that it merges the following characteristics: (1) it optimally reuses the amount of computations that can be executed per region of memory loaded, (2) this optimization for locality is done regardless of code structure, (stencil computations with any loop structure can be optimized), the data dependencies between the computations are used to partition the program instructions, (3) the program partitions (tiles) resulting from applying Diamond Tiling are fully parallel without the need to execute redundant computations (4) code generation is simple, and it can be easily incorporated in an optimizing compiler and (5) the technique presented here is applicable to N dimensional stencil computations.

Experimental evidence to support our claims is gathered using FDTD, a commonly used stencil application running on the recently developed Cyclops-64 processor. The results obtained show that stencil applications using Diamond Tiling have a lower running time and total number of off-chip memory operations than other state of the art tiling techniques.

1 Introduction

This paper addresses the problem of how to optimize a class of scientific computing programs called *stencil computations* running on many core processor architectures. This class of applications perform many read-modify-write operations on a few data arrays. The main challenge faced by stencil applications is the limitation of off-chip memory access, both in terms of bandwidth and memory latency.

Tiling [15, 16, 8, 7] is a commonly used technique to optimize stencil applications. Tiling transformations attempt to reduce the number of off-chip memory accesses by exploiting locality in the program. To that effect, a region of values is loaded to the local processor memory where increased bandwidth an better latency is available. Previous tiling techniques [6, 10] have achieved suboptimal results: Some of those approaches do not take advantage of the nature of the algorithm [7], some of them only work for one-dimensional applications [10], others require redundant computations [6], and so on.

This paper develops the details of Diamond Tiling. The diamond tiling technique is a tiling technique that results from answering the question: *What program partitioning will result in maximum locality for stencil applications running on many core processors?*.

The main contributions of Diamond Tiling, outlined on the abstract, are influenced by a simple idea: Program partitioning should not be the result of code transformations upon the original code, they should be a natural consequence of the data dependencies between the computations of a program. The source code of the program is only used to generate a full data dependency graph of the computations, and the partitions are generated based on what region of the data dependency graph provide the lowest use of off-chip memory. This

approach fundamentally departs from other tiling techniques where the program partition is heavily influenced by the original structure of the source code.

The partitions resulting from applying Diamond Tile are the result of observing how much on-chip memory is available in a processor chip, and finding what are all the computations that can be executed when a region of the data arrays of the program is loaded to it. Several steps of read-modify-write operations can be executed locally. However, the data dependencies at the boundaries of the region loaded reduce the number of computations, producing a diamond shape. To cover all the multidimensional iteration space, several diamond displacements are needed. This characteristic is explained in detail in later sections.

The effectiveness of Diamond Tiling was tested by running FDTD on Cyclops-64 with several other tiling techniques. FDTD is a common stencil application used to simulate electromagnetic wave propagation, and Cyclops-64 is a many core processor developed by IBM. The background chapter provides more information on FDTD and Cyclops-64.

The experimental results confirm that Diamond Tiling is a very effective way to optimize stencil applications. For the application and the architecture tested, Diamond Tiling provided the lowest running time, the lowest total amount of off-chip memory operations and the best performance among all other tiling techniques, surpassing recent, state of the art tiling techniques by many times in some cases.

The rest of the paper is organized as follows: Section 2 formally describes the problem addressed by this paper, Section 3 provides relevant background on the field, Section 4 describes how Diamond Tiling is the result of optimizing locality for stencil computations for many core processors while Section 5 describes the issue of generating code from the partitions obtained. Section 6 describes the experimental framework used and Section 7 presents experimental evidence showing the effectiveness of Diamond Tiling. The paper concludes with conclusions and future work on Sections 8 and 9.

2 Problem Formulation

The lack of off-chip memory bandwidth will be the limiting factor for performance of regular scientific computing codes for future many core processor architectures. Current many core architectures already suffer from this limitation [10, 11].

The limited amount of off-chip memory bandwidth is a natural result of the fact that the number of physical pins on the surface of the processor chip has a limit on the order of a thousand, but the number of computational cores inside the chip continues to increase. Even for multiprocessors available today, the ratio of bandwidth available to the Floating Point Units to off-chip bandwidth is already 40 to 1 [10]. Floating point operations pay a low price in terms of latency, available resources and power. The relative cost of executing floating point operations continues to drop while the relative price of memory operations continues to go up.

Such combination of factors drive program optimizations to regard memory operations as

precious and it makes locality one of the most important optimization goals. An effective way to optimize for locality is to partition the computations of an application in groups called "Tiles". Tiling, or partitioning the application into tiles, is an effective way to optimize for locality: The advantages of temporal locality can be exploited by reusing data.

Current tiling techniques follow the constraints of current paradigms. Traditional approaches are limited to search for possible partitions from a limited set of transformations on the starting source code. Newer approaches attempt to go beyond the limitations of source code, but their results are constrained to heuristics to transform the code or by simplistic code generation approaches.

A more fundamental approach is followed by Diamond Tiling, the technique proposed in this paper: It seeks to reduce the total running time of stencil applications. Since the question *"What should be done to reduce the total application running time?"* is too complex to address directly, Diamond Tiling build on the observation that the bottleneck for the applications under study running on many core processors is the available memory bandwidth from the cores in a chip to the DRAM memory. The following question is a natural attempt to reduce the effect of the bottleneck and is the focus of this study:

"What is the most effective way to reduce off-chip bandwidth through the use of on-chip memory?"

The analysis presented here is not bounded by the limitations of current paradigms such as the polyhedral model [5], or by restricting the results to a combination of simple loop transformations. The solution provided here completely isolates locality optimization and code generation as the fundamental problems. This work shows that introducing constraints related to code generation is not necessary at the optimization step and it shows that optimized tiles can always be built by a suitable code generator, which is also described.

Answering the question presented here will be a first step towards the more complex problem of finding the program transformations that will result in the lowest total aggregated off-chip memory traffic.

The approach taken to solve the problem is to observe the data dependencies between the instructions executed by the program. Then, the application can load a region of memory that enables a group of those instruction for execution.

The observations on the data dependency graph and the values that can be computed given a region of memory loaded into on-chip memory enable finding the answer to the question presented here. To solve the main question, a closely related question is approached: *What are the regions of memory that have to be loaded into on-chip memory such that their computational shadows completely cover the data dependency graph of the application?.*

The goal of this paper is attained through optimizing memory operations to off-chip memory. As a result, locality and synchronization overhead will also be optimized. The overall effect of optimizing applications this way is that the ultimate objective of reducing the total running time of the application will be achieved. As described before, the conclusions drawn here apply

to stencil computations running on many core architectures.

3 Background and Related Work

3.1 Tiling

Compiler transformations that focus on optimizing memory locality of programs are commonly referred to as *tiling* transformations. Tiling takes advantage of temporal and spatial locality found in programs by loading a certain region of memory into the higher levels of the memory hierarchy. A tiled program experiences a faster execution due the lower latency and higher bandwidth available to memories that are closer to the processor.

Research on how to effectively tile an application is extensive [15, 16, 8, 12, 1, 6, 11, 10, 17] and a full survey of tiling methods could be a publication in itself.

Some papers [6, 8, 9], optimize the number of synchronizations and data communication required between processors, in an attempt to address a wider range of applications where the bottleneck can be driven by synchronization or messages between processor elements. When applied to stencil computations, those approaches do not produce an optimal result since bandwidth to main memory is also optimized as a side effect of optimizing synchronization and communication.

A previous tiling approach that considers the code generation step of the optimization [1] attempts to optimize stencil applications for many core architectures but their results are limited by a combination of code transformations related to the polyhedron model, and does not effectively exploit the existence of a time loop.

3.2 Stencil Applications and FDTD

The work presented here applies to a common class of applications characterized by one or more arrays where an *update* operation is performed on their elements multiple times. The loops that implement the update operations are fully parallel, regular, they are enclosed by an outer time loop, and each one of the iterations only requires data that is close in time and space to the particular item being updated. Such applications are typically referred to as *stencil applications* and they typically result from implementations of physical phenomena through regular discretization of physical variables such as time and space.

Many stencil applications are the result of solving partial differential equations using finite differences: Heat Propagation, Particle Transport, Electromagnetic Propagation, Mass Transport and others.

The Finite Difference Time Domain (FDTD) [18] technique is a common way to solve Maxwell's Equations to simulate the propagation of electromagnetic waves. The resulting algorithm is excellent to illustrate our techniques since it is easy to understand, it is widely used,

and the main computational kernel requires little support code. Although other methods to solve electromagnetic wave propagation exist, the implementation described by Taflove [14] is chosen for the experiments and the examples of this paper due to its simplicity and effectiveness.

3.3 Tiling for Stencil Applications

Due to the frequent accesses to memory, stencil applications are particularly sensitive to the Memory Wall problem, causing a body of work on optimizations and tiling for stencil computations. A short review of the most relevant ones is presented here. To the best knowledge of the authors there is no other publication that combines all the results presented here. *e.g.* some program representations may generate program partitions such as the one presented here, but the papers that present such program partitions do not attempt to generate such partitions or do not present results of their effectiveness. The work presented here also includes a full description of how the tiles generated by Diamond Tiling can be easily integrated with code generation.

Previous approaches at tiling for stencil applications include Time skewing in the context of bandwidth optimization [17], split tiling [2] and overlapped tiling [6, 2]. Time skewing as presented in [17] does not provide experimental evidence of its advantages or of the feasibility of implementation, Split tiling [6] continues the work of time skewing but it fails to address the problems of the code generation step, and overlapped tiling [6], although simpler to implement, requires redundant computations and more off-chip memory operations.

Tiles with Diamond shapes were previously presented in [10] to maximize the reuse of on-chip memory to address. Although maximization of memory reuse was achieved, the technique presented only works for 1 dimensional stencil applications, and does not describe the details of the boundaries between tiles.

The Data Dependency Graph of the application has been used to drive the locality optimization [12], but its results are constrained by the limitations of code generation.

Other time tiling approaches follow other paths and are not mentioned here in the spirit of brevity.

3.4 Many Core Architectures

Many Core Architectures are chosen as the focus of the work of this paper due to their particular vulnerability to the Memory Wall problem. The authors expect that the trend in computer architecture will continue to flow towards architectures with more cores and a strong interconnection network inside the chip, and many simple processor cores. It is also the belief of the authors, that the field of High Performance Computing will lean towards removing complexity out of the execution cores in favor of more parallelism.

Cyclops-64 [4] was chosen as the testbed architecture. Cyclops-64, designed by IBM, is a chip multiprocessor that features a powerful on-chip interconnection network, hardware support

Chip Features			
Processor Cores	80	On-Chip Interconnect	Full Crossbar
Frequency	500 MHz	Off-Chip Interconnect	Proprietary
Hardware Barriers	Yes	Hardware Mutex	Yes
Hardware Signal - Wait	Yes	Addressable Memory	2GB
On-Chip Memory	User Managed 5MB	Off-Chip Memory Banks	4 x 64 bits
Instruction Cache	320 KB		
Processor Core Features			
Thread Units	2, single issue	Local Memory	32KB
Floating Point Unit	1, pipelined	Registers	64 x 64 bit

Table 1: Cyclops-64 Features

for barriers and signal-wait operations, a fully connected crossbar network that connects its 80 cores, and on-chip hardware for inter chip communication and global reduction operations.

Table 1 summarizes the features of the Cyclops-64 many core chip.

The Cyclops-64 system was designed to achieve an aggregated peak computational rate of 1PFLOPS when using the full system size of 13824 processor chips. A single Cyclops-64 chip has a peak performance rate of 80 GFLOP per second, the chip runs at 500 MHz and each one of its 80 cores has two single-issue thread units that share a fully pipelined floating point unit. The peak rate of 80 GFLOPS can only be achieved if the floating points unit execute Multiply-Add instructions continuously.

One of the big challenges to achieve high performance for Cyclops-64 is the great amount of locality required to achieve the peak computational rate. The Floating Point unit present in each core has a direct connection to the register file, allowing it to consume 2 double precision numbers and producing a third double precision number per cycle. On the other hand, the total off-chip memory bandwidth is only 4 x 64 bits or 4 double precision values per cycle. The Cyclops-64 chip can consume 160 double precision values in one cycle taking into account that each one of the 80 Floating point units can consume 2 double precision values per cycle. However, only 4 double precision values are available from memory each cycle, in average. In other words, the overwhelming disparity between available bandwidth to off-chip memory and the floating point units from the register file means that in average, only one out of $160/4 = 40$ instructions can be a memory operation.

Effectively using the on-chip memory in Cyclops-64 is key to achieving good performance. This paper finds the best possible use of the on-chip memory for stencil applications.

4 Data Locality for Stencil Computations

The problem of fully computing an application can be observed from the point of the data dependencies of all the computations in the program. In theory, this data dependency graph

(DDG) can be obtained by fully unrolling all loops of the program and by finding the dependencies between all the values computed, but in practice it can be obtained directly from the source code.

Tiling an application is equivalent to partitioning the DDG. Fully analyzing what is the best way to partition the DDG of an arbitrary application is a complex problem that escapes the scope of this paper. However, it is possible to analyze the full DDG of stencil applications because their dependencies are regular, greatly simplifying the situation.

It is known that bigger tiles lead to better performance results [10]: they provide better locality and result in less off-chip memory instructions providing an overall effect of faster running speed and lower execution time.

This paper addresses the problem of tile maximization. The procedure is best explained by considering the code for FDTD 1D shown in Figure 1. In the figure, `NT` represents the number of timesteps in the problem, `N` is the size of the arrays, `E`, `H`, `Ca` and `Cb` are arrays representing the physical properties of the problem, of which `Ca` and `Cb` are constants arrays in the computation.

If complete unrolling of the program is done, the data dependency graph will look like the one shown in Figure 2.

```
1  for t in 0 to NT-1
2
3    for i in 1 to N-1
4      E[i]=Ca[i]*E[i]+ Cb[i]*
5               ( H[i] - H[i-1] )
6    end for
7
8    for i in 1 to N-1
9      H[i]+=E[i]-E[i+1]
10   end for
11
12 end for
```

Figure 1: Kernel for FDTD 1D

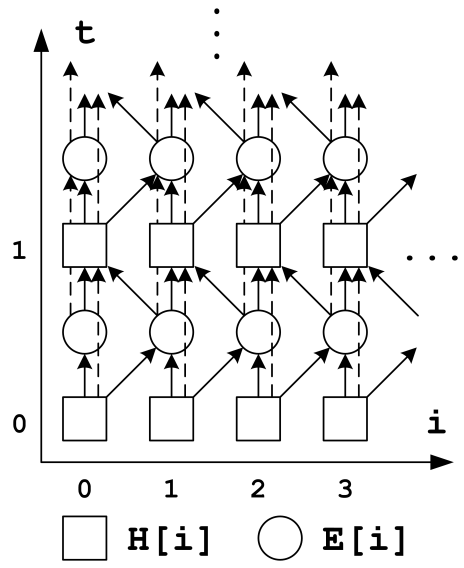

Figure 2: DDG for FDTD 1D

When running in a many core architecture, each one of the cores can load a region of values as shown in Figure 3. The whole application space is partitioned in parts that equal the size of the on-chip memory. The decision for this initial tile partitioning intends to maximize tile size and is used by most tiling techniques.

Once a region of values has been loaded, it is relatively simple to compute what instructions

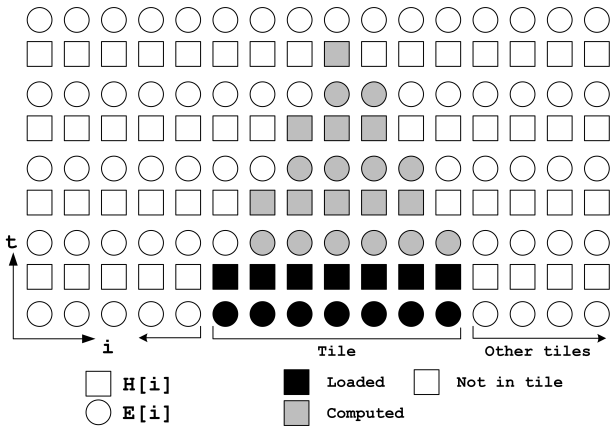

Figure 3: Values that can be computed from tile

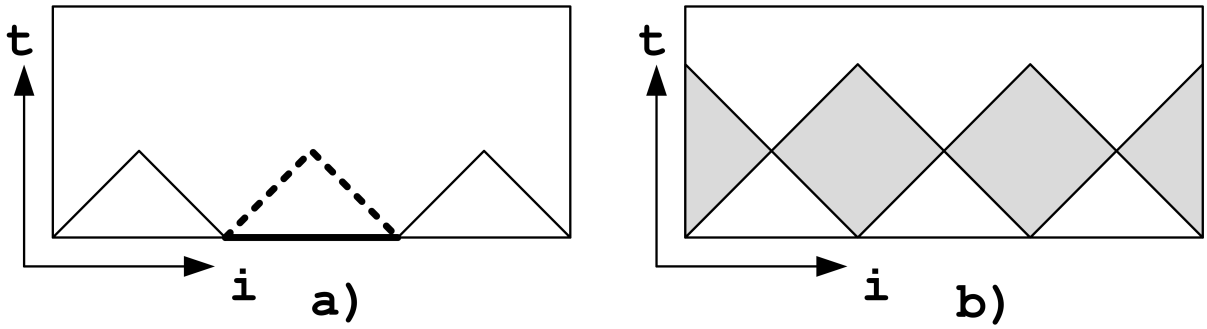

Figure 4: Diamond Tiles

in the DDG can be computed without additional memory operations. Figure 3 shows in black the values loaded and it shows in gray the values that can be computed.

At the end of the computation, regions such as the ones shown in Figure 4 a) have been computed in the DDG. Thick black lines show the values loaded and thick dashed lines show the values that are stored back to memory.

The next tiling step can compute the shaded regions as seen in Figure 4 b). Such computation can be done with a tile with the same size as before, and can compute a full diamond.

This example can be extended to N Dimensions. The ideas remain the same:

1. Partition the memory in tiles of a size equal to the available on-chip memory.
2. Find and compute the values of the DDG that can be computed from loading the data in the tile.
3. Synchronize to avoid race conditions. A barrier is enough to achieve this, but other synchronization techniques can be used.

4. Find the matching values that can be computed. Compute them.

5. Continue until all the program has been computed.

The details of each one of those steps implies some development. *e.g.* finding what values of the DDG can be computed requires some analysis on the dependencies. This is not a difficult task, it can be accomplished by looking at the values available and then looking at what values are enabled in the DDG. For the case of FDTD 1D, all values of the E array within the tile are enabled except the one on the low boundary (because of the read to `H[i-1]` which lies outside of the tile), and all the values of the H array in the tile are enabled except the one on the high boundary. So, a simple equation can be used to express the region of values of `E` and `H` that can be computed as a function of the timestep. For example, in the case of the `E` array, the functions that describe the boundaries for a loop that computes the values of `E[i]` are: $B_L = B_{L0} + t$ and $B_H = B_{H0} - t - 1$. B_L and B_H are the boundaries and B_L0 and B_H0 are the tile boundaries as shown in Figure 2.

Using the DDG to maximize tiles provides a tiling partition that surpasses other tiling techniques in terms of running time, number of tiles, and performance of individual tiles. This tiling technique is named *Diamond Tiling* due to the shape of the tiles. The next section briefly describes how Diamond Tiling is implemented.

5 Code Generation for Diamond Tiling

Diamond Tiling is made useful by the fact that code can be readily generated. Seeking clarity over formality, a careful, informal explanation of Diamond Tiling is presented here instead of a full mathematical treatment of all the possible variations of code. The discussion will be limited to providing the foundations for code generation in a way that it allows a full implementation (that includes all details).

The issue is best approached by considering the simple example of FDTD 2D (Figure 5). The size of the region of memory to be loaded (tile size) is determined by the available on chip memory, and the data to be loaded is given by a simple partition of the data arrays.

Finding what elements of the DDG can be computed is a more interesting problem: It can be addressed by observing: (1) what nodes of the DDG can be computed from the first time iteration with the data in the tile and (2) what is the change between the values computed and the values originally available. The boundaries for the loops computing the values can be extrapolated from the information of (1) and (2) for each timestep. It is possible to extrapolate since the DDG has a very regular pattern.

The height of the tile follows from the boundaries, since it can be found when the *low* and *high* boundaries for the tile meet, leaving zero elements to compute. This method works for multiple dimensions as well: the height of the tile can be computed by finding when the functions for the *low* and *high* boundaries of the loops intercept.

```
for t in 0 to NT - 1

  for i in 0 to N-1
    for j in 0 to N-1
      dHy = - ( Hy[ i - 1 ][ j ] - Hy[ i ][ j ] ) * k1;
      dHx = - ( Hx[ i ][ j - 1 ] - Hx[ i ][ j ] ) * k1;
      Ez[i][j] += k2*( dHy - dHx - S[i][j] * Ez[i][j] );
    end for
  end for

  for i in 0 to N-1
    for j in 0 to N-1
      Hx[ i ][ j ] -= k3 * ( Ez[ i ][ j + 1 ] - Ez[ i ][ j ] );
      Hy[ i ][ j ] += k3 * ( Ez[ i + 1 ][ j ] - Ez[ i ][ j ] );
    end for
  end for

end for
```

Figure 5: Kernel for FDTD 2D

After all the first tiles have been computed and reasonable synchronization has been introduced (perhaps a barrier), the next tiling step can proceed. Keeping the same mapping of tiles to addresses for the next tiling step is not effective: The next tiles should cover all the nodes in the DDG that were not computed, and their locations will be a displaced version of the previous locations. The displacement direction can be chosen arbitrarily along one of the spatial dimensions of the problem.

The region of values that can be computed for the newly displaced tiles can be easily determined at compile time, and they will form a diamond. This process can continue, each time displacing along a new direction, and returning to the original one.

A full example of how the FDTD 2D computational kernel (Figure 5) is tiled for Cyclops-64 is shown below for illustrative purposes.

5.1 Diamond Tiling for FDTD 2D in Cyclops-64.

The code for FDTD 2D is shown in Figure 5. For this example, it is assumed that all variables are double precision floating point numbers. In the code `NT` represents the number of timesteps in the problem, `N` is the size of the arrays, `Ez`, `Hx`, `Hy` and `S` are two dimensional arrays representing the physical properties of the problem, of which `S` is a constant array. The output of the program is the arrays `Ez`, `Hx` and `Hy`.

The first step to tile the code is to find a tile size. In the case of Cyclops-64, approximately 2MB of on-chip memory is available to the programmer. The remaining memory is used by the stack frame of the thread units and by kernel variables. Four arrays are used: `Ez`, `Hx`, `Hy`, and `S`. All available memory is used to hold them, so 0.5MB are used for each one of them, and represents approximately 62500 double precision values, or an array of approximately 250 by 250 elements.

Once the right size for the tiles is found, code to load and offload the data to main memory can be written. The next step is to find what values can be immediately computed from the values loaded. The computation of `Ez[i][j]` requires reading `Hx[i][j-1]` and `Hy[i-1][j]`. Since the boundaries for the local arrays loaded go from 0 to 249, the values that can be computed for `Ez` are `[1...249][1...249]`. Similarly, considering the reduced available values for `Ez`, and the original boundaries of the tile, the values for `Hx` and `Hy` can be computed as `[1...249][1...248]` and `[1...248][1...249]` respectively.

The possible ranges of values for the variables during future timesteps can be extrapolated: The limits for Ez are reduced due to the availability of values of `Hx` and `Hy` and this reduction continues. If `t` is the time variable, then the limits for `Ez` can be computed as `[1+t...249-t][1+t...249-t]`, and similarly, the limits for `Hx` and `Hy` are respectively `[1+t...249-t][1+t...248-t]` and `[1+t...248-t][1+t...249-t]`. The maximum height of the tile can be found by finding what value of `t` will make the range zero.

In general, it is possible to cover the full N Dimensional space with diamonds, but there will be 2^N different diamond displacements. Indeed, aligning diamonds at the same boundaries is not enough. This point can be easily illustrated with the FDTD 1D case. Two kinds of diamonds are needed, and they are displaced exactly half of a diamond with respect with their neighbors.

That line of thought can be expanded to reach the conclusion that 4 kinds of tiles are required for FDTD 2D. The first one was briefly presented, and the other 3 kinds have to consider that some of the values have been already computed. As done before, the boundaries can be found, and, for example, the second kind of diamond, displaced along the `i` dimension, will have boundaries for `Ez` of `[249-t(+1)...1+t+(250)][1+t...249-t]`, and the same tile height.

Assembling the 4 diamond kinds together forms a solid *tiled iteration* that covers all the range of `i` and `j` in the original problem, and has a height as calculated.

A full algorithm that can be integrated into a compiler can be easily written by expanding the one presented here. It is worth mentioning that the diamond tiling transformation requires allocation of additional temporary arrays when elements that are needed by later diamonds are overwritten. Determining whether or not required elements are overwritten is trivial from simple observation of the DDG, and some code to copy those elements can be easily integrated into the diamond tiling framework. The FDTD examples shown here do not require additional storage since elements needed by later diamond tiles are not overwritten.

6 Experiments

A number of experiments were conducted using the Cyclops-64 processor (described in Section 3) to find the validity of Diamond Tiling.

The FDTD application in 1, and 2 dimensions was used to compare Diamond Tiling to other traditional and state of the art tiling techniques.

The tiling techniques used were:

Naive: No tiling was done. All values are read and written directly from and to DRAM. The code used follows Figures 1 and 5.

Rectangular: Traditional tiling was used. Each one of the perfectly nested loops in the computation is fully tiled using all the available on-chip memory.

Skewed: The full iteration space of the application was skewed to enable tiling along the time dimension [16], and then it was partitioned using all the available on-chip memory.

Overlapped: Tiling along the time dimension and the spatial dimensions is done at the expense of redundant computations and more memory operations [6].

Diamond: Diamond Tiling, as described in this paper was used.

The code for each one of the tiling approaches was hand written. The programs were compiled with gcc for Cyclops-64 version 4.3.2, -O3 was used to automatically optimize other aspects of the code. The required synchronization between the computational steps of each tiling approach was done using the hardware supported barriers available on Cyclops-64. The experiments were run using the FAST simulator [3] and the timing results presented have been normalized to the execution of a problem of size 10000 with 4000 timesteps for FDTD in 1 dimension and a problem of size $1000x1000$ with 500 timesteps for FDTD in 2 dimensions.

Timing information about the execution time and about floating point operations per second was obtained using the hardware counters available in the Cyclops-64 processor. Execution time and floating point operation count were gathered for the computational part of the program. The initialization and finalization stages of the program represent a minority of the execution time and code size and they do not participate in tiling. For that reason, no timing information about them was gathered.

7 Results

The results of executing FDTD in 1 and 2 dimensions using several tiling techniques are presented in Figure 2.

The results confirm the hypothesis that Diamond Tiling is an effective way to tile stencil computations. For the two applications considered in Table 2, the use of Diamond Tiling resulted in a lower total execution time.

FDTD 1D			
Tiling	**Runtime (s)**	**DRAM Accesses**	**GFLOPS**
Naive	143.36	19G	0.001
Rectangular	2.73	242.04M	0.09
Skewed	0.13	162K	1.91
Overlapped	0.22	326K	1.10
Diamond	0.06	161K	3.73
FDTD 2D			
Tiling	**Runtime (s)**	**DRAM Accesses**	**GFLOPS**
Naive	20.25	2.1G	0.40
Rectangular	15.48	1.52G	0.52
Skewed	2.81	140M	2.85
Overlapped	8.52	292M	0.94
Diamond	2.21	87.5M	3.62

Table 2: Performance of several tiling approaches

Figure 2 also shows the total number of off-chip memory operations required to execute the program. The figure shows how Diamond Tiling provides the lowest number of off-chip memory operations of all the tiling techniques in the experiments. As explained before, this decrease in the total number of memory operations greatly contributes to decreasing the total running time of the application.

Traditional tiling approaches such as the widely used rectangular tiling have a performance that is far below the performance of Diamond Tiling. This is due to the fact that traditional tiling approaches do not always consider the advantages of tiling across several loops and are limited by the ability to generate code. Needless to say, Diamond Tiling and all other tiling approaches performs far better than a naive implementation. Diamond Tiling has a much lower running time and it has far less off-chip memory operations.

The execution speed of FDTD when skewed tiling is used is comparable (although slower) than that of Diamond Tiling. Skewed tiling, however, is limited in that its low execution time is likely to hold only for single processor chip executions due to the wavefront dependencies between tiles that limit their available parallelism at the start and end of the computation.

The recent Overlapped Tiling [6] overcomes the limitation of low parallelism between tiles at the expense of more memory operations and redundant computations. The price paid for parallelism results in a lower execution speed and more off-chip memory operations. For large tiles, such as the ones used in the experiment, diamond tiles require less memory operations while still enabling full parallelism between tiles. Also, it does not require redundant computations.

In summary, the experimental data presented here supports the idea that Diamond Tiling is an excellent technique to tile stencil applications.

8 Conclusions

This paper presents Diamond Tiling, a tiling technique for stencil applications. The main idea behind tiling technique is that the data tiles are extracted from the dependencies of the program regardless of how the code was written. Diamond Tiling constitutes and advance on the state of the art because it optimizes the locality of tiles for stencil applications. The experimental results shown here confirm these claims and shows that Diamond Tiling provides better performance than other currently used tiling techniques.

This paper presents an improvement over a previous publication [10] by the authors in the sense that Diamond Tiling was extended from 1 dimension to multiple dimensions, the issue of code generation was addressed, and results comparing diamond tiling to other state of the art and commonly used techniques are provided.

Diamond Tiling has excellent properties that make it preferable over other tiling techniques for stencil computations: (1) It enables full parallelism between tiles without the expense of redundant computations or additional memory operations, (2) it provides optimum utilization of memory loaded to on-chip memory, lowering the total off-chip memory operations, and (3) generating code to implement the diamond tiles is simple and it can be readily implemented as part of an automatic optimizing compiler.

As explained throughout the paper, it is simple to generate code to tile N dimensional applications. One of the characteristics of Diamond Tiling is that it requires 2^N different alignments for the tiles to cover all the N dimensional space. This limits the applicability of Diamond Tiling when small tiles are used, since the overhead of managing 2^N different alignments has to be overcome by the increased locality of the diamonds. Analysis done by the authors [10] show that diamond tiles are useful when the tile width is greater than 2^N.

The positive impact of Diamond Tiling increases as more memory becomes available: for an N dimensional problem with a processor with enough memory to hold a tile of width w, the number of elements computed in a diamond tile is of $O(w^N)$ while the number of memory operations required is $O(w^{N-1})$. In other words, as the tile size increases the computations per memory operation increase as well.

The amount of memory operations will play an even more critical role in the performance of the application if its instructions are heavily optimized. The speed of the programs as they were run (using only the -O3 flag at compile time) are already dependent on the speed to complete the required memory operations.

In summary, Diamond Tiling is an effective way to tile stencil computations such as FDTD. Experimental evidence shows that when it is used, it can lower the running time and it is easy to generate code for it. The Diamond Tiling technique is useful for many core processors with large amounts of computing power and limited amount of off-chip memory bandwidth. It is the belief of the authors that the disparity between available computing power and on-chip memory bandwidth will continue to increase, making diamond tiling of the utmost importance for stencil applications running on many core architectures.

9 Future Work

Future work for diamond tiling will address the issue of tiling for systems with multiple many core processor chips.

The effectiveness of using barrier synchronization as opposed to point to point synchronization will be studied. Consumer-producer synchronization primitives such as phasers [13] can potentially reduce any execution noise in the stencil applications.

The greater problem of fully addressing the optimization of an arbitrary application from the point of view of the data dependency graph is still an open problem.

The authors will continue research to extend the fundamental ideas of Diamond Tiling to other applications beyond the scope of stencil applications.

10 Acknowledgment

This work was supported by NSF (CNS-0509332, CSR-0720531, CCF-0833166, CCF-0702244), and other government sponsors.

We express our gratitude to Chen Chen for his valuable feedback on the quality of writing, to all the members of the CAPSL group at University of Delaware for their valuable feedback, and to our reviewers for their uninterested help to produce high quality science.

References

[1] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 1–10, New York, NY, USA, 2008. ACM.

[2] U. Bondhugula, M. Baskaran, A. Hartono, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Towards effective automatic parallelization for multicore systems. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–5, April 2008.

[3] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Fast: A functionally accurate simulation toolset for the cyclops-64 cellular architecture. *Workshop on Modeling, Benchmarking and Simulation (MoBS), held in conjunction with the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*, 2005.

[4] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Toward a software infrastructure for the cyclops-64 cellular architecture. In *High-Performance Computing in an*

Advanced Collaborative Environment, 2006. HPCS 2006. 20th International Symposium on, pages 9–9, May 2006.

[5] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program.*, 34(3):261–317, 2006.

[6] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective automatic parallelization of stencil computations. *SIGPLAN Not.*, 42(6):235–244, 2007.

[7] Monica S. Lam and Michael E. Wolf. A data locality optimizing algorithm. *SIGPLAN Not.*, 39(4):442–459, 2004.

[8] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 228–237, New York, NY, USA, 1999. ACM.

[9] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 201–214, New York, NY, USA, 1997. ACM.

[10] Daniel Orozco and Guang Gao. Mapping the fdtd application for many core processor. *International Conference on Parallel Processing ICPP*, 2009.

[11] Daniel Orozco and Guang Gao. Tiling techniques to map applications to multi-core systems. *CAPSL Technical Memo Number 87*, 2009.

[12] Lakshminarayanan Renganarayana, Manjukumar Harthikote-matha, Rinku Dewri, and Sanjay Rajopadhye. Towards optimal multi-level tiling for stencil computations. In *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS*, 2007.

[13] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 277–288, New York, NY, USA, 2008. ACM.

[14] A. Tavlove. *Computational Electrodynamics*. 1995.

[15] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *SIGPLAN Not.*, 26(6):30–44, 1991.

[16] M. Wolfe. More iteration space tiling. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664, New York, NY, USA, 1989. ACM.

[17] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 171–180, 2000.

[18] Kane Yee. Numerical solution of inital boundary value problems involving maxwell's equations in isotropic media. *Antennas and Propagation, IEEE Transactions on*, 14(3):302–307, May 1966.