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

# Locality Optimization of Stencil Applications using Data Dependency Graphs

*Daniel Orozco, Elkin Garcia, Guang R. Gao*

**CAPSL Technical Memo 101**
October, 2010



University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

# Contents

# List of Figures



# List of Tables

**Abstract**

This paper proposes tiling techniques based on data dependencies and not in code structure.

The work presented here leverages and expands previous work by the authors in the domain of non traditional tiling for parallel applications.

The main contributions of this paper are: (1) A formal description of tiling from the point of view of the data produced and not from the source code. (2) A mathematical proof for an optimum tiling in terms of maximum reuse for stencil applications, addressing the disparity between computation power and memory bandwidth for many-core architectures. (3) A description and implementation of our tiling technique for well known stencil applications. (4) Experimental evidence that confirms the effectiveness of the tiling proposed to alleviate the disparity between computation power and memory bandwidth for many-core architectures.

Our experiments, performed using one of the first Cyclops-64 many-core chips produced, confirm the effectiveness of our approach to reduce the total number of memory operations of stencil applications as well as the running time of the application.

# 1 Introduction

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

Tiling, described in [1, 2, 3, 4, 5, 6] and in many other publications, 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 a program. To that effect, frequently used data is loaded to the local processor memory where increased bandwidth and better latency are available.

Previous tiling techniques, when applied to *stencil applications*, achieve suboptimal results because they either do not take advantage of the algorithm [4], they only work for one-dimensional loop structures [6], they require redundant computations [5], or, in general, a good tiling solution can not be found because the particular code written by the programmer does not fit the tiling technique.

Previous research by the authors looked at the problem of tiling stencil applications from the side of the data produced in the application regardless of the source code used to produce such data. The question posed was *"What program partitioning will result in maximum locality for stencil applications running on many-core processors?"*, and a partial answer was provided for a particular instance of a 1 Dimensional Stencil application. Evidence on the advantage of the technique proposed was produced using a simulator [6].

The limitation of memory bandwidth on many-core applications has been addressed to other levels in the memory hierarchy for some linear algebra applications [7, 8]. They have proposed alternatives to find optimum tiling to the register level and mechanism for hiding memory latency.

The main contributions of this paper, outlined in the abstract, are influenced by a simple idea: Program tiling and/or 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 should only be used to generate a full data dependency graph for the computations. The data dependency graph is then used to formulate an optimization problem whose solution is the tiling strategy that, in our case, would result in the least use of off-chip memory resources. This approach fundamentally departs from other tiling techniques where the program partition is heavily influenced by the original structure of the source code.

The tiling strategies resulting from a dependency-graph-based optimization for stencil computations can be collectively designated as *Diamond Tilings*. A very limited form of Diamond Tiling was successfully demonstrated in the past [6] without a detailed description of the optimization problem or a general solution. This paper fully develops the idea of how to generate Diamond Tilings based on the data dependencies of an application. We show guidelines to

implement tiles that increase reuse of data in the case of parallel stencil applications.

The key idea behind the development of the Diamond Tiling class of tilings is simple: Maximum locality can be obtained if all local, *enabled* computations are executed. In here, the term *enabled* is borrowed from the dataflow field, and it refers to a computation whose input data is available, and whose input data resides in on-chip memory. This thought is translated into an optimization problem that is solved in the context of the implementation possibilities such as memory available (that limits the total size of a tile), synchronization primitives and so on. The main difference between Diamond Tiling and other tiling techniques is that Diamond Tiling is not concerned with the original structure of the code. The source code is only used to generate a complete data dependency graph of the application, but it is not required in itself to do the tiling transformation. Many programming models can benefit from the strategies used in the development of Diamond Tiling: Dataflow programs, serial programs, parallel programs and any other kind of program where a data dependency can be extracted.

The effectiveness of Diamond Tiling is presented in our results. Diamond Tiling was compared against other tiling techniques using an application that models the propagation of electromagnetic waves (using the FDTD algorithm) in multiple dimensions. The Cyclops-64 many-core processor developed by IBM [9] is used as our testbed architecture. When DRAM bandwidth is the bottleneck of the application, Diamond Tiling provides the shortest running time, the smallest 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 provides relevant background on the field, including details on stencil applications and the Cyclops-64 processor. Section 3 formally describes the problem addressed by this paper. Section 4 provides a mathematical proof of the optimality of Diamond Tiling to reduce the amount of DRAM operations, and Section 5 shows how to apply the mathematical results of Section 4 to implement an application. The experiments used to test the effectiveness of Diamond Tiling and their results are described in Sections 6 and 7. The paper concludes with conclusions and future work on Sections 8 and 9.

# 2 Background

This section provides a brief introduction of the knowledge required to understand the paper. This section covers Stencil Applications, Cyclops-64 and many-core Architectures, DRAM memory limitations in many-core architectures, and traditional tiling approaches.

## 2.1 Stencil Applications

Stencil Applications are applications whose main computation consists of repeatedly updating an array with a copy of a previous version of the array. Stencil applications are of great use for science and engineering, and they typically solve partial differential equations. Common

stencil applications include simulations of Heat Propagation, Particle Transport, Electromagnetic Propagation, Mass Transport and others.

The Finite Difference Time Domain (FDTD) [10] technique is a common algorithm to simulate the propagation of electromagnetic waves through direct solution of Maxwell's Equations. FDTD was chosen to illustrate the techniques presented here since it is easy to understand, it is widely used, and it can be easily written for multiple dimensions.

### 2.2 Cyclops-64 and Many-core Architectures

Cyclops-64 [9] is a many-core processor chip recently developed by IBM. Cyclops-64 allows simultaneous execution of 160 hardware threads. Cyclops-64 shows a new direction in computer architecture: It has a user-addressable on-chip memory and no data cache, it has direct hardware support for synchronization (barriers as well as hardware sleep and awake) and a rich set of atomic operations. The main features of Cyclops-64 chip are shown in Table 1.

| **Chip Features** | | | |
|---|---|---|---|
| Processor Cores | 80 | On-Chip Interconnect | Full Crossbar |
| Frequency | 500 MHz | Off-Chip Interconnect | Proprietary |
| Hardware Barriers | Yes | Instruction Cache | 320 KB |
| Hardware Sleep - Awake | Yes | Addressable Memory | 1GB |
| On-Chip Memory | User Managed 5MB | Off-Chip Memory Banks | 4 x 64 bits |
| **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

Cyclops-64 was chosen as the testbed for this application since it has a large amount of parallelism (160 threads simultaneously) and it is highly programmable (each thread runs its own code).

### 2.3 DRAM Limitations for Many-core Architectures

For many-core architectures, the disparity between the availability of DRAM and the availability of floating point units is perhaps the most limiting factor for performance. This disparity is only likely to increase with each new processor chip generation: it becomes increasingly easier to build more and more floating point units inside the chip, but the physical connections that supply memory from outside the chip improve only marginally.

Current many-core architectures already suffer from this limitation. As an illustration, the ratio of bandwidth available to the Floating Point Units to off-chip bandwidth is already 60 to 1 in Cyclops (Table 1): There are 80 Floating Point Floating units that consume 2 and produce 1

64-bit values per cycle while there are only 4 memory banks that can supply 1 64-bit value per cycle each.

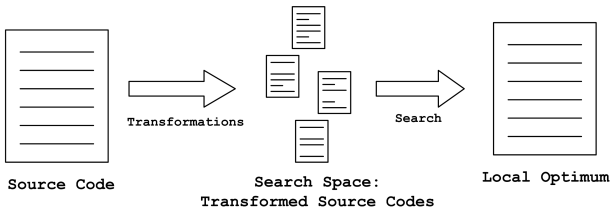
Floating point operations pay a low price in terms of latency, available resources and power, while, on the other hand, memory operations on DRAM have a long latency, use more power, and their total bandwidth is limited. For those reasons, memory operations to DRAM are the determinant factor in the cost of computations for scientific programs.

## 2.4 Tiling

The limitations of DRAM memory operations for many-core architectures 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 they are constrained by simplistic code generation approaches.

Some tiling techniques (Figure 1) apply a set of transformations to the source code provided and search for the best locality in the transformed programs. This approach was demonstrated with success in the past, but it is constrained by the fact that the search space is inherently limited by the programs that can be generated from the initial source code.




Traditional approaches find the best tiling technique from a fixed set of transformations. For example, they find the affine transformation that produces the best result on the source code.

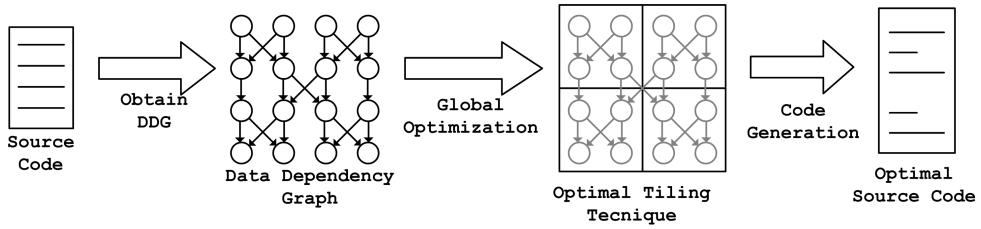
Figure 1: Traditional Approaches to Tiling

Tiling techniques that are not constrained by the structure of the original source code have been proposed [11], but their usability is still restricted to programs with a specific structure, and their expressing power is still limited.

# 3 Problem Formulation

The previous sections described the main techniques for tiling. Most of them are the result of applying clever transformations on the source code of the application.

We propose instead focusing on the fundamental problem of tiling, regardless of code structure or expressiveness (Figure 2): *"What is the most effective tiling strategy to reduce off-chip bandwidth through the use of on-chip memory?"*




Tiling, in this paper, is the result of an optimization done directly on the data dependency graph of the program. This is different to traditional tiling techniques where a set of proposed techniques is evaluated to find a local optimum.

Figure 2: Tiling using Data Dependency Graphs

It is not claimed in this paper that it is always possible or convenient to find a solution to the tiling problem as presented in Figure 2, or that such a solution exists for all possible programs running all possible inputs on all possible architectures. For example, solution of the problem requires knowing the full dependency graph of the application, which can be practically or theoretically impossible for some applications.

The focus of this paper is specific: The problem of finding an optimum tiling is solved for stencil applications (See Section 2) running on parallel processors. Under those conditions the optimization problem can be approached because:

- The computations in the program are independent of the input data,
- it is possible to count all the computations in the program and
- it is possible to build a data dependency graph providing the relations between the computations on the program.

The rest of the paper presents the solution in detail.

# 4 Optimal Tiling for Stencil Computations

The previous section formulated the problem of tiling optimality for a program in general. This section addresses the problem for the particular case of parallel stencil computations. Parallel

in this context means that a tile can be fully computed when its memory has been loaded; no communication with off-chip memory or with other processors is required.

```
1 for t in 0 to NT-1
2
3   for i in 1 to N-1
4     E[i] = k1*E[i] +
5         k2 * ( 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 3: Kernel for FDTD 1D

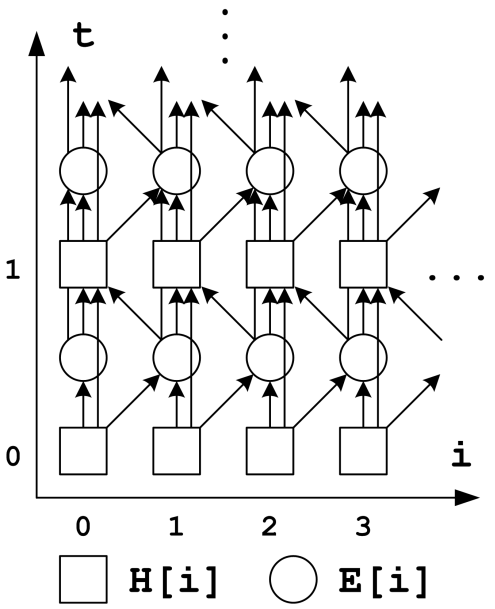



Figure 4: DDG for FDTD 1D

The optimal tiling problem is approached as a formal optimization problem. The optimization problem maximizes the ratio of computations to memory operations for all possible tiling approaches.

Tiling of memory addresses and computations is an integer optimization problem on the computation space. Figure 4 shows the computation space for an implementation of FDTD (Figure 3). The problem is an integer optimization problem since all computations are associated with a spatial grid and a computation step. The problem is extended to allow continuous values for the memory addresses as well as the number of computations (Figure 5). The problem is then solved in the continuous space. As will be seen in the equations below, the optimum solution of the continuous problem is integer, which guarantees that it is also a solution of the integer optimization problem.

The rest of this section provides definitions of the variables used, some theorems and a final remark that ultimately results in the optimal tiling technique.

**Definition 1** *$D_T$ is the computation space. The elements of $D_T$ are the values computed by the stencil application. Each one of the elements of $D_T$ is associated with a vector $x_T = (x_{T,0}, x_{T,1}, ..., x_{T,M-1}, t)$ where $M$ is the number of spatial dimensions in the stencil problem, and $t$ represents time in the computation. All components in $x_T$ are integers because they are associated with a spatial grid and a computation step.*

As an example, consider the implementation of the FDTD algorithm in 1 dimension shown

in Figure 3. The elements of $D_T$ for the implementation shown in Figure 3 are represented by circles and squares in Figure 4. In the graph shown in Figure 4, $x_{T,0}$ corresponds to the $i$ direction and $t$, the last coordinate of the vector, corresponds to the $t$ direction. For simplicity of the explanation, consider that the circle and the square with the same coordinates in space and time are merged and are referred to simply as one *element* of the computation space.

**Definition 2** *Also, each element of $D_T$ is associated with a set of vectors $d_i = (d_{i,0}, d_{i,1}, ..., d_{i,(M-1)}, d_{i,t})$, $i = 0, ..., p$ that represent the direction of the dependencies in the computation space. In Figure 4 each arrow represents each $d_i$ These vectors are also illustrated in Figure 5.*

**Definition 3** *$D$ is the extended continuous space of $D_T$. In the same way, a point in $D$ can be associated with a vector $x = (x_0, ..., x_M - 1, t)$.*

**Definition 4** *$T$ is a region (e.g. a tile) of the continuous computation space $D$.*

Figure 5 shows the extended computation space $D$, and an arbitrary tile $T$.

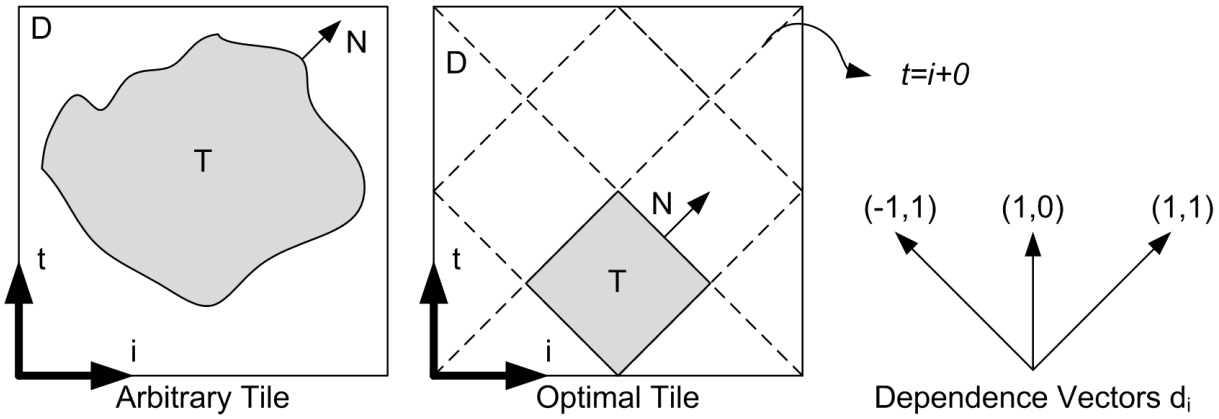



Figure 5: Continuous computation space, tiles, and dependences for FDTD 1D

**Definition 5** *$S(T)$ represents the elements that have to be loaded to compute $T$ and $V(T)$ represents the elements that can be computed in tile $T$. $S(T)$ and $V(T)$ can be interpreted as the surface and volume of $T$ respectively.*

**Definition 6** *$N$, or $N(x)$, $x \in S(T)$ is a unit vector perpendicular to the surface $S(T)$ at point $x$. The orientation of $N$ is towards the inside of $T$ if $S(T)$ is associated with loads or it is towards the outside of $T$ if $S(T)$ is associated with stores.*

**Definition 7** *Vertical bars $|\quad|$ are used to indicate the cardinality of a set in $D_T$. $|S(T)|$ and $|V(T)|$ can be interpreted as the area of the surface and volume of $T$ respectively.*

**Definition 8** *$m(T)$ is the amount of memory used by a particular tiling $T$ and $m_{max}$ is the maximum amount of on-chip memory available.*

The optimization problem for finding the tiling in the continuous space is:

$$\max_{T} f(T) = \frac{|V(T)|}{|S(T)|} \tag{1}$$

$$s.t. \quad m(T) \leq m_{max}, \quad T \text{ is a parallel tiling} \tag{2}$$

The solution can be found by the following theorems:

**Theorem 4.1** *If $T$ is a valid, parallel tiling, then $N \cdot d_i \geq 0$ for $i = 0, 1, ..., p$.*

**Proof** This condition ensures the dependencies points inwards. It implies, computations inside $T$ do not require loading data from other tiles or from memory and can be computed in parallel.

**Theorem 4.2** *If $T^*$ is an optimum solution of the optimization problem, then $T^*$ is a convex tile.*

**Proof** Considering a feasible non-convex tile $T_{nc}$ and the smallest convex tile $T_c$ that fully contains $T_{nc}$. $T_c$ is also feasible. $f(T_{nc}) \leq f(T_c)$ because $|S(T_{nc})| \geq |S(T_c)|$ and $|V(T_{nc})| \leq |V(T_c)|$. Since for any feasible non-convex tile there exists a feasible convex tile with a better or equal value of $f(T)$, it follows that $T^*$ is convex.

**Theorem 4.3** *If $T^*$ is an optimum solution of the optimization problem, then, at each point along the boundaries of $T^*$, $N$ is perpendicular to at least one of the dependency vectors $d_i$ at that point.*

**Proof** Consider all the tiles $T$ for which $|S(T)|$ is a particular constant value. For those tiles the optimization problem of Equation 1 is equivalent to maximization of $|V(T)|$ under the constraints of Equation 2.

Maximization of $|V(T)|$ is equivalent to maximization of a multidimensional integral[1] of $-N \cdot \hat{t}$ along the surface of T. Which is maximum when $N \cdot \hat{t}$ is minimum at all points along the surface of the tile T.

Since theorem 4.1 implies that $N$ lies in a cone extending in the direction of $+\hat{t}$. The minimum value of $N \cdot \hat{t}$ is when $N \cdot d_i = 0$, for some $i$ and thus, $N$ is perpendicular to at least one of the dependence vectors.

[1]The full details of the derivation of the integral are omitted due to space constraints.

**Theorem 4.4** *The Tiling technique that solves the optimization problem of Equations 1 and 2 are tiles where the computation space $D_T$ is partitioned with planes that extend in the direction of the dependences.*

**Proof** Theorem 4.3 states that an optimum tile is formed by planes whose normal vectors are orthogonal to the dependence vectors. It follows that the planes extend in the direction of the dependence vectors. Because $N \cdot d_i = 0$ is required. The only shape that satisfies this requirement are diamonds. Thus the name given to the technique. This is also true in the computation space $D_T$ where the optimum $T^*$ is a diamond restricted to planes in a grid.

**Theorem 4.5** *The optimum tile has a size that tries to use all the available memory.*

**Proof** The function $f(T)$ is a ratio between volume and surface of the tile, and this ratio increases with the amount of memory used. The maximum tile size is bounded by the constraint described in Equation 2.

Theorems 4.4 and 4.5 are enough to uniquely describe the tiling partition (Figure 5). The tile partitions extend in the direction of the dependences (Theorem 4.4) and the size of the tile should be as big as possible (Theorem 4.5).

Note that the results of this section find the global optimum of the described problem. The solution found here does not address other constraints not specified. For example, the result obtained in this section does not consider the effect of latency, only the effects of bandwidth because, as argued, bandwidth is likely to be the most important limiting factor, and other limitations such as latency can be hidden using techniques such as double buffering.

The following sections provide examples and results of using the theorems of this section to minimize the number of off-chip memory operations.

# 5 Implementation

The process to obtain the optimum tile size and shape for an application was described and solved for stencil applications in Section 4. This section provides and informal description of the results of previous sections that seeks clarity over formality. The complete details of the particular implementation used for the experiments has been prepared as a separate publication and it can be found in [12].

The main conclusion of Section 4 is that the optimum tiling technique is ultimately defined by the direction of the dependences in the code. Those dependencies ultimately define how the application is tiled.

Consider the application of Figure 3, and its corresponding expanded data dependency graph (Figure 4). For the elements in the data dependency graph, the vectors that represent the

dependencies are $(-1,1)$, $(0,1)$, and $(1,0)$. Following the optimality result, the boundaries for the optimum tiling extend in the direction of the dependencies. The planes that form the tiles would be $t = i + c$, $t = -i + c$ and $t = c$, where $c$ is a constant that is associated with the location of the tile, and $t$ and $i$ are the time and space indexes respectively. The normal vectors $N$, found using linear algebra on the planes, are $(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$, $(0,1)$ and $(-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$ respectively for the planes. As explained before, the planes of the optimum tile have the lowest possible value of $N \cdot \hat{t}$. Since $N \cdot \hat{t}$ is $(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}) \cdot (0,1) = \frac{1}{\sqrt{2}}$, $(0,1) \cdot (0,1) = 1$ and $(-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}) \cdot (0,1) = \frac{1}{\sqrt{2}}$ respectively for the planes $t = i + c$, $t = c$, $t = -i + c$, then $t = i + c$, and $t = -i + c$ are chosen as the partition planes because of their lower value of $N \cdot \hat{t}$.

The resulting planes partition the iteration space as shown in Figure 5. The planes are spaced so that each tile uses the maximum memory available.

Stencil applications in any number of dimensions can be similarly tiled.

# 6 Experiments

A number of experiments were conducted using the Cyclops-64 processor (described in Section 2) to test our claims of optimality for Diamond Tiling.

A simulation (using the FDTD technique) of an electromagnetic wave propagating both 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:

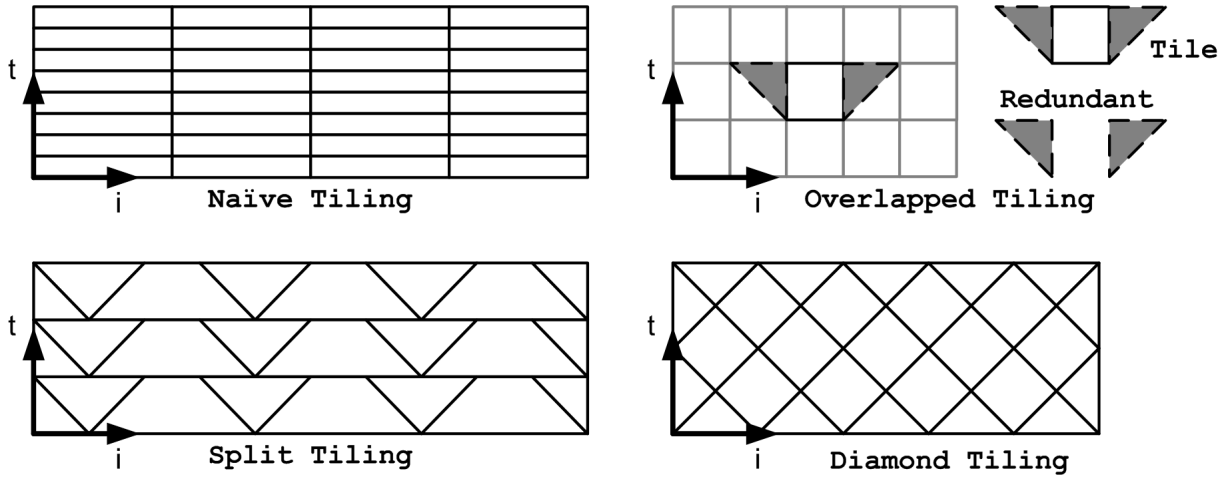



Figure 6: Tiling techniques used in the experiments

**Naïve:** The traditional rectangular tiling was used. Each one of the perfectly nested loops in the computation is fully tiled 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 [5].

**Split:** Two or more tile shapes are used to fully partition the iteration space such that time tiling with no redundant computations is achieved [5].

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

Figure 6 shows a conceptual view of the tiling techniques when applied to a 1 dimensional problem. The experiments use tiling in both one and two dimensions.

Tiling was done at the on-chip memory level. Register tiling and tiling at other levels was not done because the focus of our study is the limitation in the memory operations between DRAM and the on-chip memory in many-core processors. Further levels of tiling are likely to accentuate the results, but such experiments fall outside of the scope of our work.

The code for each one of the tiling approaches was hand written in C. The programs were compiled with ET International's Compiler for Cyclops-64 version 4.8.17 with optimization flags -O3.

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 one of the first Cyclops-64 chips produced.

The 1 dimensional implementation computes 4000 timesteps of a problem of size 10000 while the 2 dimensional application computes 500 timesteps of a problem of size $1000 \times 1000$. The tile sizes used were 256 and $24 \times 24$ for the implementations in 1 and 2 dimensions respectively. These tile sizes were chosen so that they used all the available on-chip memory.

Information about the execution time, floating point operations and memory accesses was obtained using the hardware counters available in the Cyclops-64 processor.

All results are reported 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 they are not included in the results.

# 7 Results

The result of executing the testbed application (FDTD) in 1 and 2 dimensions using several tiling techniques are presented in Figures 7 to 12.

The results confirm the hypothesis that Diamond Tiling is an effective way to tile stencil computations. For the two applications considered in section 6, the use of Diamond Tiling resulted in a lower total number of DRAM memory operations, and for the cases in which the bandwidth was the limiting factor of the application, Diamond Tiling resulted in the best performance.

Figures 7 and 8 show the total number of off-chip memory operations required to execute the program. They show how Diamond Tiling provides the lowest number of off-chip memory

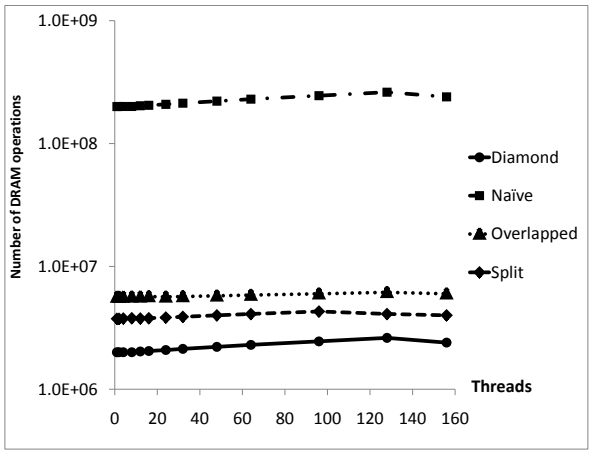



Figure 7: FDTD 1D: Operations on DRAM

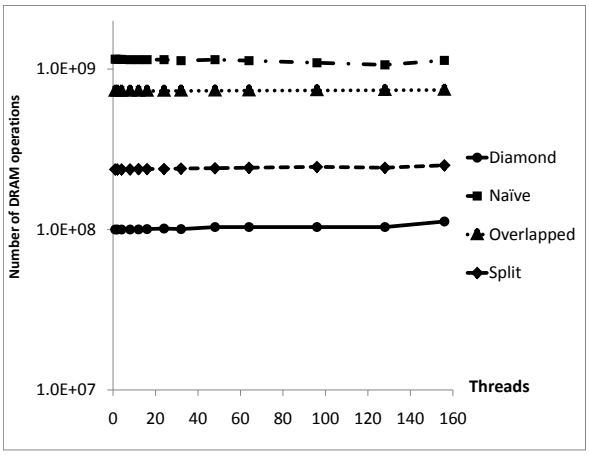



Figure 8: FDTD 2D: Operations on DRAM

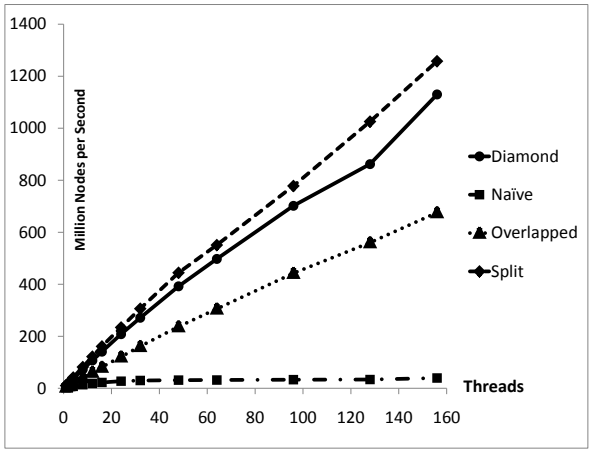



Figure 9: FDTD 1D: Performance

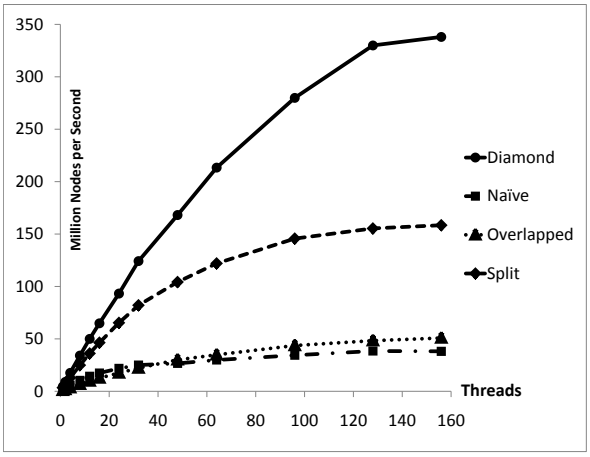



Figure 10: FDTD 2D: Performance

operations of all the tiling techniques in the experiments. As explained before, this decrease in the total number of memory operations greatly contributes to decrease the total running time of the application as can be detailed on Figures 9 and 10 using the number of nodes processed per second as a metric. As a result of the reduction in off-chip memory operations and the increasing of performance, the average bandwidth used is decreased, as can be seen on Figures 11 and 12. A better utilization of off-chip bandwidth is achieved particularly for 2D-FDTD using Diamond Tiling. In all cases, the maximum amount of bandwidth available was 2 GB/s.

Traditional tiling approaches such as the widely used rectangular tiling (our naïve implementation) 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. Diamond Tiling has a much lower running time and it has far less off-chip memory operations.

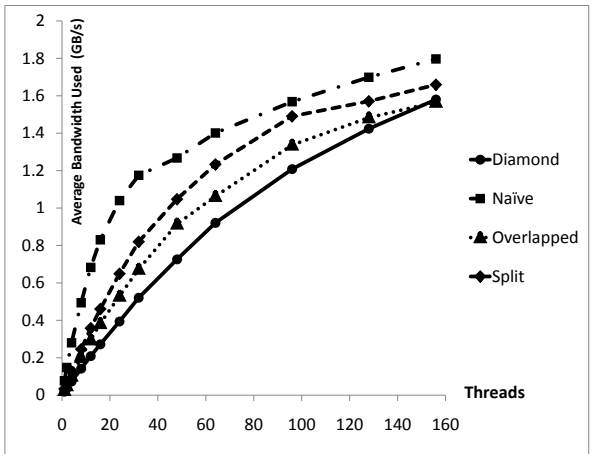



Figure 11: FDTD 1D: Bandwidth used

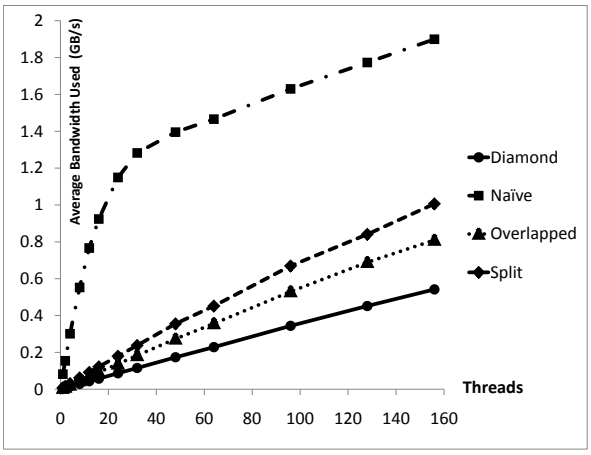



Figure 12: FDTD 2D: Bandwidth used

The recent Overlapped Tiling [5] 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, Diamond Tiling 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 in a parallel execution environment.

# 8 Conclusions

This paper presents a technique for data locality optimization that is based on the analysis of the data dependency graph of the application.

The most important contribution of this work is that it approaches the problem of tiling as a mathematical problem of optimality rather than as a transformation. Although the general approach of tiling a generic application through mathematical optimization of a performance function is not always feasible, it is nevertheless useful for Stencil Applications.

Section 4 showed a formal description of the problem and developed it to reach a simple conclusion: In stencil computations, the iteration space should be partitioned into Diamond-shaped tiles. This result was obtained for the case of parallel applications. If the parallel restriction is dropped, it may be possible to find other tiling techniques with better performance. For example if the execution is restricted to be serial, skewed tiling [2] may have a better performance than Diamond Tiling.

The resulting Diamond Tiling has excellent characteristics: (1) It produces fully parallel tiles, (2) it provides optimum utilization of the off-chip DRAM bandwidth, and (3) it is easy to generate

code for it. Additionally, the positive impact of Diamond Tiling increases when combined with other optimizations and possible architecture changes such as better synchronization or more memory.

The results of this paper extend the results of previous research by the authors. In a previous publication, Diamond Tiling was conjectured to be optimal for FDTD in 1 dimension [6]. This paper formally confirms such claim and extends the result to general stencil applications in any number of dimensions.

This paper also opens the door for formal optimization of other applications where a full data dependency graph is possible to obtain. Or, at the very least, it provides an alternative to locality optimization from source code.

It is possible that the importance of optimization in a mathematical way will increase in future generations of many core architectures. If current trends in computer architecture continue, more and more parallelism will be available while data movement will become even more expensive.

# 9 Future Work

The main contributions of this paper are significant to stencil applications. Future work in locality using data dependency graphs can focus in extending its applicability to a larger number of applications, or to integrate other techniques into the formal framework proposed.

So far, the results presented here only address the difficulties of DRAM memory accesses of one many-core chip. The issue of how to partition a distributed memory application across multiple many-core chips using data dependency graphs is a future topic of research.

The tradeoff between barrier synchronizations and point to point synchronizations needs to be evaluated as well. Consumer-producer synchronization primitives such as phasers [13] can potentially reduce execution noise in stencil applications.

Future work will also focus on providing a generic mathematical framework for optimization. In the case of the stencil application optimized here, all the optimization steps were hand-designed by the authors. A more attractive approach would be to design a consistent methodology that could be applied to any application without being restricted to a particular class of applications. Whether or not that is possible is unknown to the authors at the time of publication.

## References


[1] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," *SIGPLAN Not.*, vol. 26, no. 6, pp. 30–44, 1991.

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

[3] A. W. Lim, G. I. Cheong, and M. S. Lam, "An affine partitioning algorithm to maximize parallelism and minimize communication," in *ICS '99: Proceedings of the 13th international conference on Supercomputing*. New York, NY, USA: ACM, 1999, pp. 228–237.

[4] M. S. Lam and M. E. Wolf, "A data locality optimizing algorithm," *SIGPLAN Not.*, vol. 39, no. 4, pp. 442–459, 2004.

[5] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," *SIGPLAN Not.*, vol. 42, no. 6, pp. 235–244, 2007.

[6] D. Orozco and G. Gao, "Mapping the fdtd application for many core processor," *International Conference on Parallel Processing ICPP*, 2009.

[7] I. E. Venetis and G. R. Gao, "Mapping the LU Decomposition on a Many-Core Architecture: Challenges and Solutions," in *Proceedings of the 6th ACM Conference on Computing Frontiers (CF '09)*, Ischia, Italy, May 2009, pp. 71–80.

[8] E. Garcia, I. E. Venetis, R. Khan, and G. Gao, "Optimized dense matrix multiplication on a many-core architecture," in *Proceedings of the Sixteenth International Conference on Parallel Computing (Euro-Par 2010)*, Ischia, Italy, 2010.

[9] J. del Cuvillo, W. Zhu, Z. Hu, and G. 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*, May 2006, pp. 9–9.

[10] K. Yee, "Numerical solution of inital boundary value problems involving maxwell's equations in isotropic media," *Antennas and Propagation, IEEE Transactions on*, vol. 14, no. 3, pp. 302–307, May 1966.

[11] P. Feautrier, "Automatic parallelization in the polytope model," in *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*. London, UK: Springer-Verlag, 1996, pp. 79–103.

[12] D. Orozco and G. Gao, "Diamond Tiling: A Tiling Framework for Time-iterated Scientific Applications," in *CAPSL Technical Memo 91*, University of Delaware, 2009.

[13] J. Shirako, D. M. Peixotto, V. Sarkar, and W. 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*. New York, NY, USA: ACM, 2008, pp. 277–288.