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

Mapping the FDTD Application to Many-Core Chip Architectures

Daniel Orozco and Guang Gao

CAPSL Technical Memo 087
March 3rd, 2009
Revised March 17th, 2009

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

Abstract

This paper reports a study of mapping the Finite Difference Time Domain (FDTD) application to the IBM Cyclops-64 (C64) many-core chip architecture [1]. C64 is chosen for this study as it represents the current trend in computer architecture to develop a class of many-core architectures with distinct features e.g. software manageable on-chip memory hierarchy (vs. a hardware-managed data cache), high on-chip bandwidth, fine grain multithreading and synchronization, among others.

Major results of our study include:

1. A good mapping of FDTD can effectively exploit the on-chip parallelism of C64-like architectures and show good performance and scalability.

2. Such performance improvement is derived by employing a number of code optimization techniques - such as time skewing and split tiling that judiciously exploit the architecture features described in (1).

3. High performance requires maximum reuse of on-chip memory, which is obtained by tiling with non conventional tile shapes.

4. Such code optimization techniques we used in (2) and tiling such as the one used in (3) should be implementable within a reasonable compilation framework - opening a new set of possibilities for compiler optimizations.

1 Introduction

The FDTD method, proposed by Yee[2], and explained in detail by Tavlove [3], provides a numerical algorithm to solve partial differential equations using time differences as approximations to partial derivatives. FDTD algorithms constitute the computational kernel of many scientific computing applications.

The amount of calculations required in practical FDTD computations make it unfeasible to achieve reasonable running times if only one processor is used. Efficient parallel execution is desired when running FDTD and other similar scientific applications, but porting serial code to a parallel architecture is not a straightforward task.

Little literature related to mapping applications with data dependencies common to FDTD to parallel architectures with many cores has been published. Mainly, we try to extend traditional parallelization techniques [4, 5, 6], to allow better results. It is paramount to redesign tiling [7] to take advantage of the memory behavior, since in the past, tiling techniques were heavily optimized to handle hardware-managed on-chip cache memories. If a fully manageable memory is used instead, we present a clever technique that tiles the iteration space taking into account the direction of the data dependences. Tiling shapes are not necessarily square as is common in most implementations by other authors.

Full control of the memory hierarchies of an architecture can be exploited to design a tile shape and size that will have an optimal performance for a given problem. In here, we chose the Cyclops 64 architecture [1], a revolutionary architecture developed by IBM, that features fully manageable memory hierarchies.

We analyze the data dependencies of FDTD and propose a number of loop transformations that enable efficient tiling. We show how tiling size and shape affects the performance of the finished program. Then, a number of incremental optimizations in the style proposed by [8] are used to boost the performance of our code. We take advantage of the high amount of on-chip parallelism and the versatility of the control flow instructions available in the Cyclops-64 chip to optimize tiling for our problem.

In our results, we show that on-chip memory reuse is vital to achieve high performance. Our analysis shows that tiling is the right approach to obtain good performance through memory reuse.

Our proposed techniques follow a sequence of transformations that only depend on the application's data dependences and in the architecture features. For that reason, they are suitable to be incorporated into many-core compilers.

The paper is organized as follows: Section 2 presents background on stencil applications and their optimizations. Section 3 defines the problem addressed in this paper. Section 4 shows a first initial implementation of the FDTD algorithm and presents a number of optimizations that lower the overall running time of the application when used together. Section 5 presents our results and talks about the effectiveness of our approach. Section 6 presents other related work in the field. Finally, Section 7 presents our conclusions and possible directions for our future work.

2 Background

This section introduces previous knowledge in stencil-type computations such as FDTD and discusses previous results on stencil optimizations.

2.1 Stencil Computation Techniques

Stencil Computations represent a set of important scientific applications. A number of very common applications fall into the category of stencil computations such as explicit integration methods for solution of partial differential equations such as FDTD [3], multimedia and image processing applications that use neighbor pixels as inputs to filters [9], particle transport codes [10] and others.

A characteristic shared by most stencil applications is their low computation to communication ratio in their naive versions, forcing application developers and compilers to resort to techniques such as tiling [7] or loop skewing [11, 12] and time skewing [13]. Previous approaches that use skewing show too many redundant memory computations [12] or produce tiles intended to be executed serially that exhibit little parallelism. Attempts to use cache oblivious algorithms [14] in the skewed versions take advantage of the cache behavior but do not address the interaction between multiple processing units in a single chip.

More recent approaches [4] try to achieve tile parallelism by redundantly computing the required dependences for each tile, or by executing the tiles in a wavefront style. Those approaches have their drawbacks since many floating point operations are wasted in redundant computations, and although wavefront execution of the tiles grants some parallelism, the start and end of the computation pay a significant cost. Furthermore, to be able to exploit the parallelism found in a wavefront execution of tiles, extra program complexity has to be added.

Split tiling [4] alleviates the problem of redundant computations, but its mathematical treatment is obscure, previous work on it does not present satisfactory performance results, and it fails to evaluate the impact of tiling shapes on the total performance of the applications.

2.2 The Finite Difference Time Domain Algorithm

The Finite Difference Time Domain algorithm (FDTD) is an iterative solver for the electromagnetic equations widely known as Maxwell's equations. The FDTD algorithm is used in a wide range of applications spanning from medical research [15] to military development [16].

For simplicity, we have chosen the 1 dimension version of FDTD. The techniques presented here can be similarly applied to 2 or 3 dimensions.

The computational kernel of FDTD can be represented by the pseudo code shown in Figure 1. It consists of the update of two arrays, `E` and `H` using two constants (`k1` and `k1`) that represent the physical environment. A *timestep* corresponds to an update of the `E` and `H` arrays and translates to a time increment in the simulated physical problem.

FDTD problems of interest in real life usually have a large number of timesteps. The size of the arrays used are only limited by the computational power and can be larger that 10^9 elements.

```
S1:  for t = 1 to NT
S2:     for i = 1 to N
S3:        E(i) += k1*H(i) + k2*H(i-1)
S :    end for
S4:     for i = 1 to N
S5:        H(i) += E(i) - E(i+1)
S :    end for
S : end for
```

Figure 1: Computational kernel for FDTD 1D, Naive version

2.3 Previous Attempts at Parallel FDTD

A number of previous approaches have been employed to execute the FDTD algorithm in parallel [17], [6], [18], [5].

Commercial implementations known to the authors rely on the fact that for most practical applications N is a very large number and focus on parallelizing the inner loops in the computational kernel while executing the outer time loop sequentially. This approach, however, suffers from great bandwidth requirements.

Some of previous approaches at parallelism focus on programmability [6] or scalability in cluster systems [5]. It is however a common issue to see that very little attention has been paid to bandwidth. Several vendors of hardware accelerated FDTD algorithms [19] are still limited in their results by the total bandwidth to main memory. To the authors' knowledge, the algorithms used in the supercomputers around the world do not explicitly employ time skewing or tiling shapes other than rectangular tiling along the iterator's dimensions.

2.4 Challenges in Many-Core Architectures

Bandwidth to main memory is steadily losing ground to floating point units, which can be built by the thousands on a chip and can run at multiple gigahertz.

The discussion on how to use on-chip memory remains open. A majority of the users that understand the problem think that an automatic cache system is better by some abstract judgment. However, when parallelizing an application where efficient coordination of the threads is paramount, compilers and users continuously struggle to take full advantage of the capabilities of multi-core chips where cache coherency is maintained by automatic hardware.

How to use user-manageable on-chip memories present in many-core architectures is still a challenge today. Better techniques for data reuse in on-chip memories have to be developed by leveraging on the high versatility that they provide. On-chip memory reuse is an initial step to addressing the very limited amount of bandwidth between the multiprocessor chip and the main memory.

The computer architecture field has presented us with new promising features since fully manageable on-chip memories found in recently developed projects expose many opportunities for bandwidth reduction. Users or the compilers can eliminate altogether the coherency problem by manually managing on-chip memories and introducing synchronization where needed to maintain a memory state that is meaningful for the application at hand as opposed to letting the hardware execute billions of unnecessary memory synchronizations.

New synchronization techniques and better techniques for tiling and data locality can be developed if the constraints of cache line sizes or cache line associativity are taken away. Tiling could be done along any direction and with any size and shape if the on-chip memory is fully manageable.

Further research must be done on synchronization primitives as there is not a simple way to orchestrate thousands of threads on a chip. An initial step has been given with the introduction of hardware barriers in recent multi-core chips such as Cyclops-64 [1]. However, how to efficiently implement fine grain synchronization remains an open issue.

Compiler technology has to be advanced to match the advances in many-core architectures. It is still not very clear how to use previous knowledge in serial optimizations when applied to parallel optimizations. Our approach provides a few steps towards the solution of the problem by presenting a successful optimization approach that can be integrated into parallel compilers. Our proposed compiler optimizations leverage in the compiler's knowledge and control of the underlying architecture.

We take advantage of our knowledge in the Cyclops-64 [1] architecture and the FDTD application to present a full case study of how bandwidth and synchronization can be done in a multi-core architecture.

Cyclops-64 is a revolutionary architecture developed by IBM. It has been designed the name Blue Gene Cyclops (BG/C) and it targets the petaflop supercomputing market with a peak performance in excess of 1 PFLOPS. Cyclops-64 is an architecture that features 80 processing cores in a chip, with two thread units per core, one 64-bit floating point unit, a user-manageable on-chip memory of 32KB per thread unit and a high-bandwidth on-chip crossbar network with a total bandwidth of 384 GBytes/s. Each Cyclops-64 chip has four memory banks for a total off chip memory bandwidth of 16GBytes/s. Each core (containing two thread units and one floating point unit) unit can issue 1 double precision floating point Multiply Add instruction per cycle, for a total performance of 80 GFLOPS per chip when running at 500MHz. The architecture has 64 general purpose, double precision registers. The chip contains a special signal bus that allows threads to perform a barrier operation in a few tens of cycles and features other fast synchronization hardware operations like thread sleep and thread wakeup.

The Cyclops-64 architecture was selected due to its high programmability and its lack of on-chip memory cache. All memory inside the chip is fully manageable by the user.

3 Problem Formulation

In the previous sections we have introduced a number of challenges faced by stencil-type applications such as FDTD. New architecture features and the growing gap between memory and processor speed have created a number of open problems that need to be solved.

The problems that will be addressed by this paper include:

- What is a good mapping of stencil applications such as FDTD to many-core architectures?
- What are suitable optimizations that will take advantage of the features of many-core architectures exposed to the user?
- What is the effect of data reuse on FDTD and what kind of tiling techniques are required to maximize it?
- What kind of compiler framework would be required to implement tiling and loop transformations as presented here into many-core compilers?

The rest of the paper addresses those questions and presents experimental results that show our findings.

4 Algorithm and Optimizations

A number of transformations can be applied to the algorithm to make it suitable for tiling, thus reducing the total bandwidth required. Such transformations as well as an analysis of the effects of tiling on data reuse and bandwidth are presented in this section.

4.1 Transformations on the Base Algorithm

The performance of a naive execution of FDTD, where all the instances of statement `S3` from Figure 1 are executed in parallel (followed by a parallel execution of the instances of `S5`) will be limited by the requirements of bandwidth to main memory.

Cache usage or fast on-chip local memories can be used, but ultimately, they will not solve the bandwidth limitation since the amount of memory loads per computed array element stays constant.

Time skewing along with loop tiling can be used to alleviate the bandwidth problem while finding a good mapping to a parallel architecture. We seek to use transformations that would result in a tiling space that will exhibit enough parallelism as to use simultaneously all the processing units inside a many-core chip.

Unfortunately, the data dependences and the structure of the loops shown in Figure 1 do not allow direct transformations into efficient forms of tiling. A transformation inspired by the Single Static Assignment form [20] can be used to relax the data dependencies found in Figure 1 at the expense of more memory usage. t is added as a dimension to the data arrays as shown in Figure 2.

```
S1:  for t = 1 to NT
S2:   for i = 1 to N
S3:      E(i,t) = E(i,t-1)+
S3:           k1*H(i,t-1)+k2*H(i-1,t-1)
S :   end for
S4:   for i = 1 to N
S5:      H(i,t) = H(i,t-1) +
S5:           E(i,t) - E(i+1,t)
S :   end for
S :end for
```

Figure 2: A Single Assignment transformation

To simplify the following explanations we define the *node* `EH(i,t)` as the pair of values `E(i,t)` and `H(i,t)`.

A function `h` operating on *nodes* can be used to better represent the data dependencies of the application, as seen in Figure 3. This representation can be obtained by noting that `E(i,t)` and `H(i,t)` depend on previous values of `EH` computed at time `t-1`. It should be pointed that values of `H(i,t)` require two values of `E` at time `t` that can be computed from the arguments of function `h`.

```
S1:  for t = 1 to NT
S2:     for i = 1 to N
S3:        EH(i,t) = h(EH(i-1,t-1),
S3:            EH(i,t-1),EH(i+1,t-1))
```

Figure 3: Program written as nodes of values

The representation in Figure 3 can be used to derive parallel tiling techniques such as the split tiling presented in Figure 4. Highlighted in black is one of the tiles that each processing element would have to execute. The tiling presented in Figure 4 represents an improvement over other tiling techniques since tiles do not require redundant computations as in [4] and tiles do not have to be executed serially as in [7].

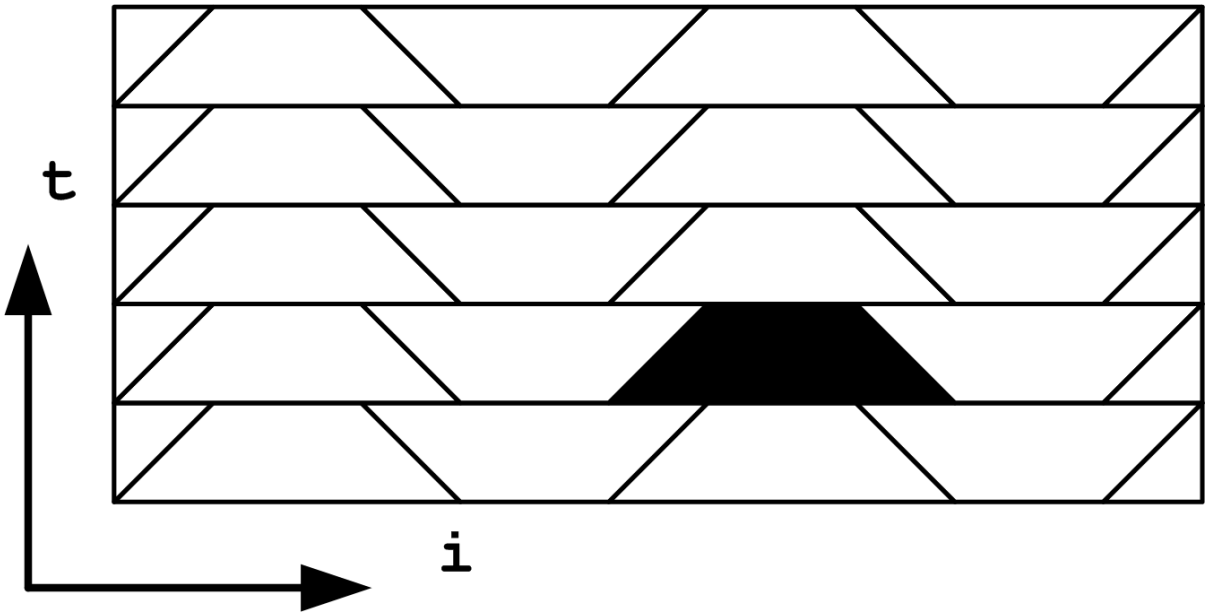

Figure 4: Split tiling on a time-skewed loop

The tiling shape of Figure 4 reduces the bandwidth requirements of the program since nodes computed inside the tile do not require memory load operations from main memory. However, without further loop transformations the bandwidth of the application is still prohibitive since all values of `EH(i,t)` have to be sent back to main memory.

An in-place computation of `EH(i,t)` can be done to avoid sending all values back to memory. To do so, arrays containing the current nodes being computed and the previous nodes computed must be kept in on-chip memory.

4.2 Maximization of Data Reuse

The main limiting factor of the FDTD algorithm is its requirements for main memory bandwidth. Increasing data reuse of on-chip memory can increase the performance of the algorithm by reducing its dependence on main memory bandwidth.

In this paper we propose the use of tiles with diamond shapes as shown in Figure 6.

Diamond-shaped tiles cause values kept in the working arrays to belong to different time iterations. This is not a problem since some startup and finishing code can be added to compute the half-diamonds required to start and finish the computation of FDTD. Tiles on the boundaries along the i dimension are also triangular in shape and are handled by additional code that can run in parallel with other tiles aligned to the same value of time.

Figure 5 shows the logical positioning of the working arrays at the start of the tile computation (marked "a" in the figure) and at the end of the tile computation (marked "b" in the figure). The boxes shown in the figure represent nodes in the iteration space. The lightly and darkly shaded boxes represent the working node arrays. Boxes with dots represent nodes computed between the moments "a" and "b" shown in the figure and constitute the nodes that would be computed during the execution of a tile. As can be seen in the figure, all elements of the arrays do not share the same position in time. For that reason, startup and finishing codes are required to reach a particular time boundary.

If the two arrays shown in Figure 5 are kept in on-chip memory where bandwidth to the processor is much larger than the bandwidth of main memory to the processor, an increase of the perceived bandwidth is experienced.

To analyze this, consider the FDTD application tiled as shown in Figure 6. Let $2c$ be the size of the array allocated in on-chip memory and let c_{mem} be the number of memory operations that would be required, per node, if no time skewing is used. Also, consider that for most real problems, both N and NT are large.

Figure 6 shows the tile shape (diamond) that can produce the most useful computation without communicating with main memory given a region of memory already loaded into on-chip memory. The black highlighted areas represent the memory locations that need to be loaded. A total of two arrays of size $2c$ (previous and current time iteration) must be loaded to compute the diamond region.

The diamond tile with the dimensions shown in Figure 6 has a total of $2(c^2 - 2c)$ nodes. The reuse of the working array after it has been loaded into on-chip memory results in less main memory operations per node – if diamond tiling is used. Equation 1 shows the total number of memory operations to main memory, per node, if diamond tiling is used.

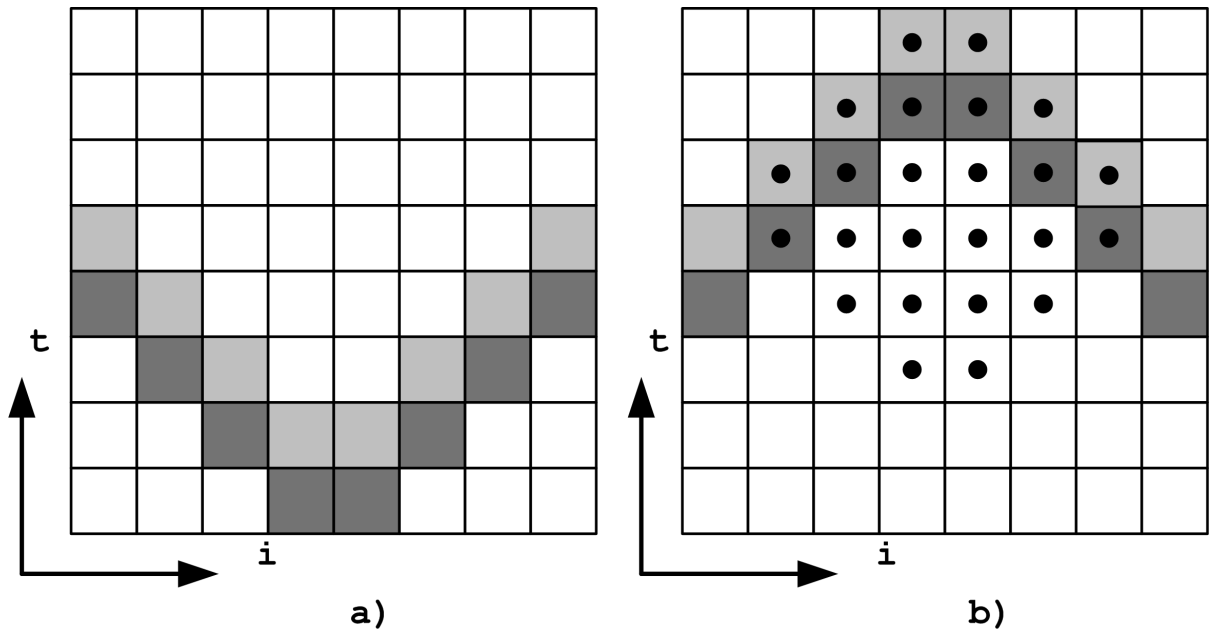

Figure 5: Logical array positioning in tile

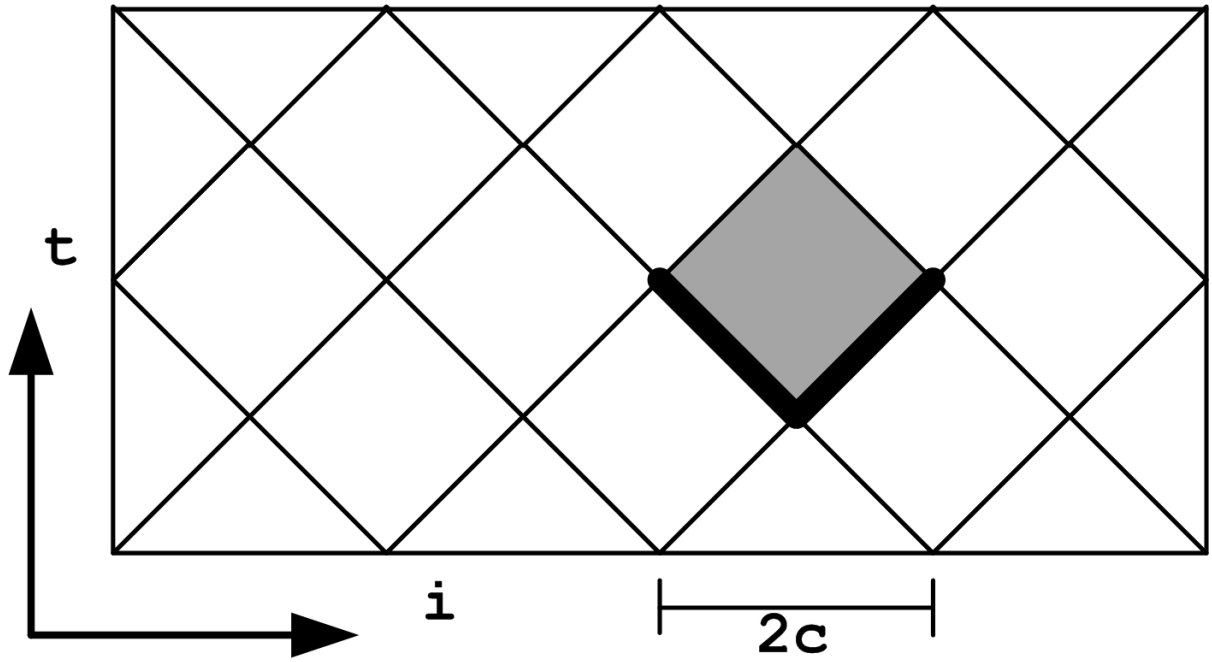

Figure 6: Diamond Tiling

$$\frac{2c \cdot c_{mem}}{c^2 - 2c} \tag{1}$$

The important conclusion about Equation 1 is that it multiplies the required memory band-

width to main memory by $2c/(c^2 - 2c)$, which is a number smaller than 1 for $c > 4$. This result holds as long as there is enough on-chip memory bandwidth to support the desired computation rate plus a one-time load of the arrays (with size $2c$) and their respective store to memory. The best results are obtained when $2c$ is chosen as large as possible given the amount of on-chip memory.

The concept of reuse, presented in Equation 1 can be extended to include other tile shapes, and in particular, it can be applied to a many-core architecture. The following sections will leverage on the initial results presented here to find a reuse equation that can be used in a broader range of cases.

4.3 Parallelization of FDTD on a Many-Core Chip

After the transformations proposed, the algorithm can be executed in a many-core chip. The application data can be partitioned into *chunks* so that each chunk occupies all on-chip memory. A chunk is formally defined as the largest consecutive partition of application data that will fit in on-chip memory. The chunk size will be denoted by M.

To simplify our discussion, consider that M is a multiple of $2P$, where P is the number of processing elements in the chip.

When the application executes, each processor in the chip will load, compute and offload a full diamond tile. The size of the arrays that need to be transferred from and to main memory are, per processor, $2c = M/P$.

A *phase* is the computation of all tiles that are aligned at a particular time value. A phase is composed of the load of one or more chunks, the computation of the tiles inside the chunks, and the offloading of the chunks to main memory.

A barrier synchronization is used at the end of each *phase* to ensure that all processor elements have finished. Once this is done, time is increased and the process repeats with another phase.

Odd phases load their chunks aligned with the boundary of the problem while even phases load their chunks with a displacement of c to match the diamond positions with the previous phase.

Special code is required to begin and end the application since the first and the last phase have triangle tiles instead of diamond tiles, as can be seen in Figure 6.

This algorithm keeps all processing elements working all the time.

4.4 Analysis of Tile Shape

Given the nature of the dependences between nodes, the iteration space can be also partitioned as shown in Figure 7.

Tiles can have several shapes that can be described using a parameter p. p is defined as the reduction factor in tile height and is computed as the ratio of the tile height to the maximum possible tile height (which is c). Figure 8 shows various tile shapes and their corresponding p value.

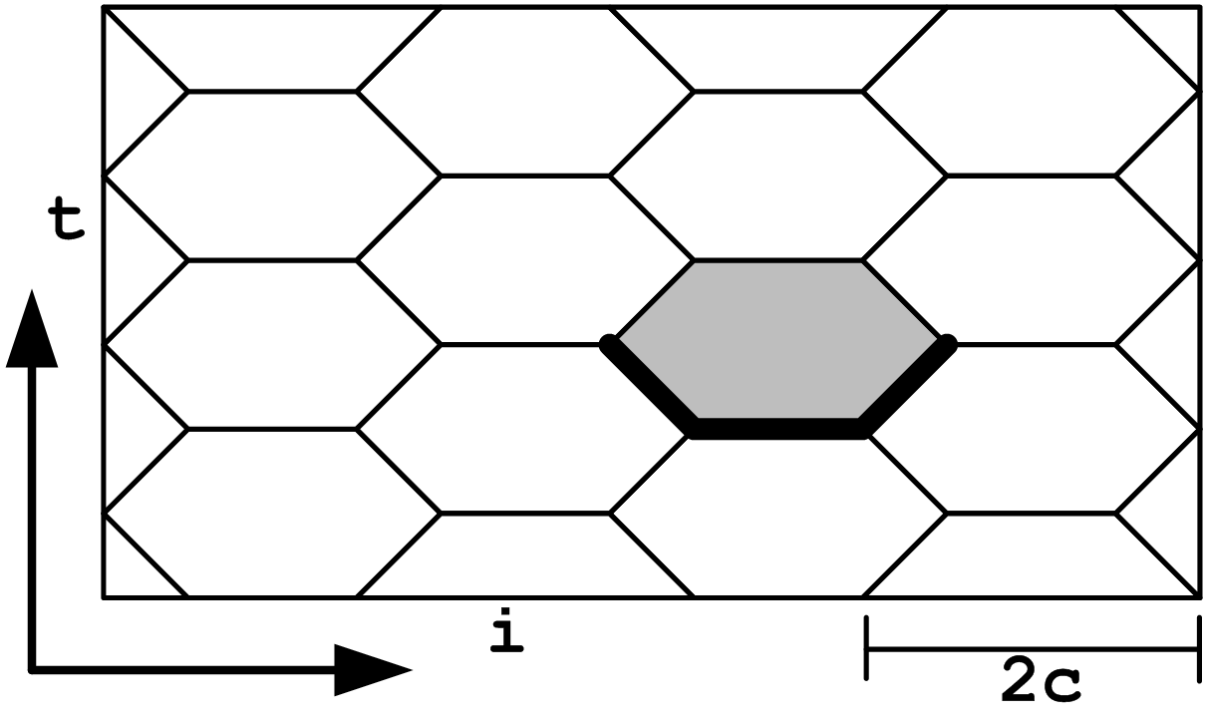

Figure 7: Tiling with varying tile shape

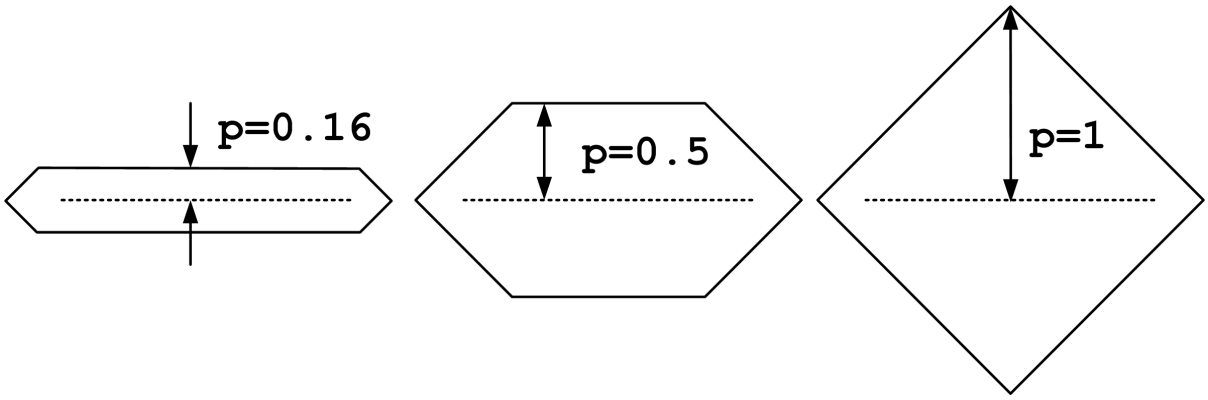

Figure 8: Tile with parametric shape

Tiles aligned at a particular time value can start concurrently, and they require the load of 2 node arrays of size $2c$, their logical location indicated by the thick dark line at the bottom of the shaded tile shown in Figure 7.

To simplify our analysis, we will assume that the array sizes are continuous. The discretiza-

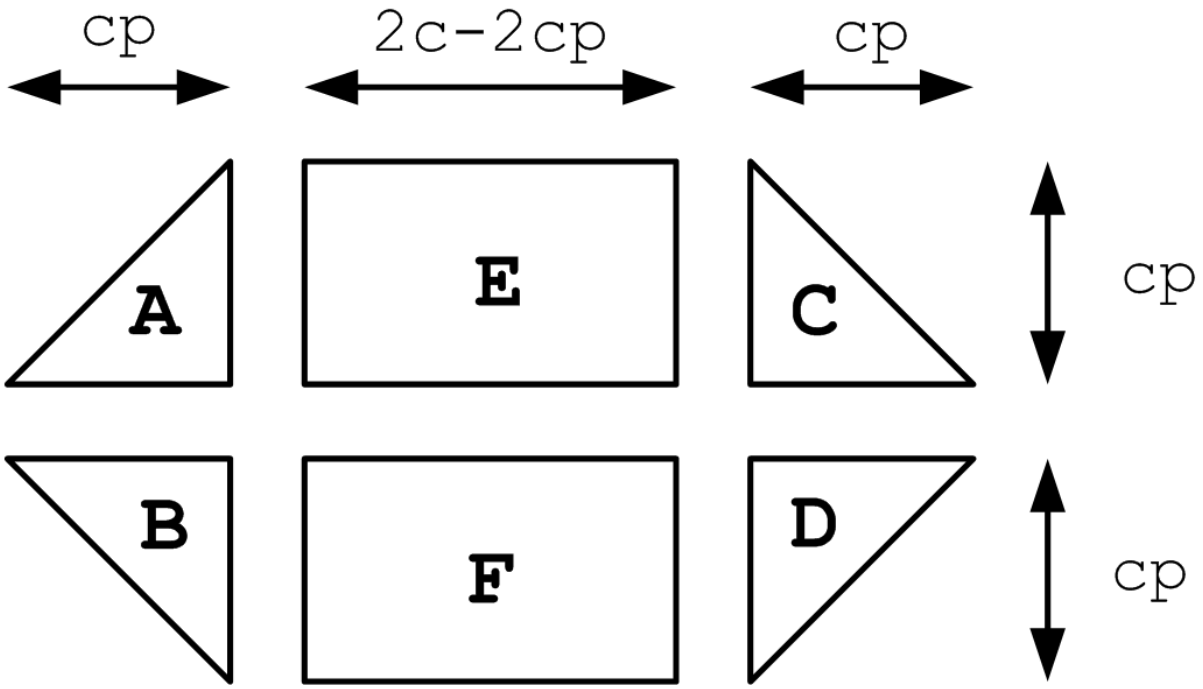

Figure 9: Tile dimensions

tion error introduced by this assumption is negligible for problem sizes found in real life.

Given the original fact that the FDTD application is bounded by memory bandwidth, we claim that the best tile shape will be the one that has the best reuse of on-chip memory.

We define the reuse r of on-chip memory as the ratio of nodes computed to the number of nodes loaded in a tile.

The approach that we take to calculate r is to find an expression for the total number of nodes in a tile and an expression for the number of nodes loaded.

The later value is trivial since the number of nodes loaded is $4c$. That value corresponds to loading two arrays of size $2c$, one for the node corresponding to the last time step computed and one corresponding to the previous one.

To find the number of nodes computed, consider that the height of the tile can be approximated as cp, since by definition, p is the ratio of the tile height to the maximum possible tile height.

The number of nodes in the tile can be approximated by the area of the tile. Figure 9 can be used to better explain how to calculate the area of the tile.

Regions A, B, C, and D in the figure have each an area of $(cp)^2/2$, while regions E and F have an area of $cp(2c - cp)$. The total area A for the tile is:

$$A = 4\frac{(cp)^2}{2} + 2cp(2c - 2cp)$$
$$A = 4c^2p - 2c^2p^2$$
$$A = 2c^2(2p - p^2)$$

Since the reuse r was defined as the ratio of memory loads to nodes computed, we can rewrite the reuse as

$$r = \frac{A}{4c}$$
$$r = \frac{2c^2(2p - p^2)}{4c}$$

Simplifying, we can express the reuse by Equation 2.

Intuitively, the reuse is given by the area of the tile divided by its width. In particular, a tile with a particular width value will maximize its reuse if its area is maximized (when $p = 1$).

$$r = \frac{c(2p - p^2)}{2} \tag{2}$$

Large values of r result in the best application performance. Equation 2 has the implication that the required bandwidth to main memory is scaled by $1/r$, and consequently, the best results are obtained if large values of c are used and a value of $p = 1$ is selected. In other words, **the diamond shape is the best tiling shape in terms of performance and bandwidth for the data dependencies in FDTD**.

The width of the tile, $2c$, is important in the sense that it both reduces the memory bandwidth required and distributes the cost of loading and storing array values to main memory. For that reason, the tile width should be chosen as large as the on-chip memory would allow. Figure 10 presents a theoretical cycle cost of executing a node for two values of $2c$ as a function of the tile shape. It can be seen in the figure that for low values of p, the bandwidth to main memory limits the performance of the application. As p gets larger, the number of cycles required to compute a node asymptotically approach the floating point cost plus the control flow cost of computation.

The analysis presented here confirms that diamond shape tiling along with a tile as big as possible will provide greater data reuse that is reflected as better performance in FDTD and other applications with similar dependencies.

5 Results

We have found that our optimization techniques have produced excellent performance results when applied to the FDTD computation.

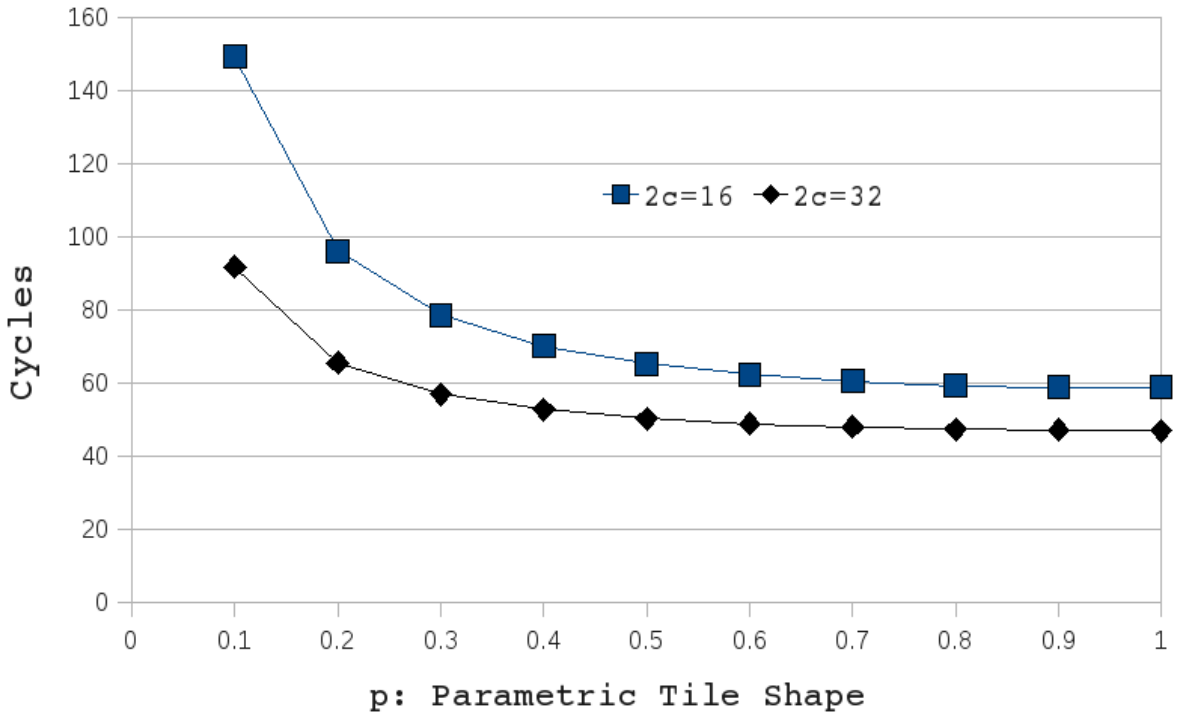

Figure 10: Cycle cost per node computed

The following algorithms were tested in Cyclops-64. The compiler used was *gcc* version 3.4 for Cyclops-64, with compilation flags `-O3`. Timing was done using the hardware counters present in the Cyclops-64 chip, and it only measures the execution of the computational loops. No timing information was gathered for the initialization of the required arrays or for memory allocation steps.

- Naive DRAM: FDTD was computed exactly as described in Figure 1. All arrays resided in off chip memory (DRAM).
- Triangle: The iteration space was divided into triangles. The execution proceeds in "steps" where each step calculates a constant number of time iterations. Each "step" is tiled in triangles so that each triangle can be computed entirely after loading its base. Intuitively, a triangle tile is equivalent to only calculating half of a diamond tile.
- Diamond: The iteration space was split into diamonds. Diamonds aligned along a particular time value are mutually independent an where executed in parallel. Figure 6 shows an example of diamond tiling. The highlighted black boundaries show the memory needed to start the computation.
- Varying Shape: Although the Diamond tile represents the best reuse of on-chip memory, the control flow instructions are more abundant in the regions close to the top and bottom of the diamond. The varying shape tiling does not use complete diamond tiles. Instead,

the tiles have a hexagonal shape that resembles a diamond without a tip. In this sense, less control flow instructions are used at the expense of less memory reuse.

The performance results are shown in Figures 11 and 13.

As expected, the Naive version has a very low performance. Off-Chip bandwidth is not enough to load the required operands from main memory fast enough.

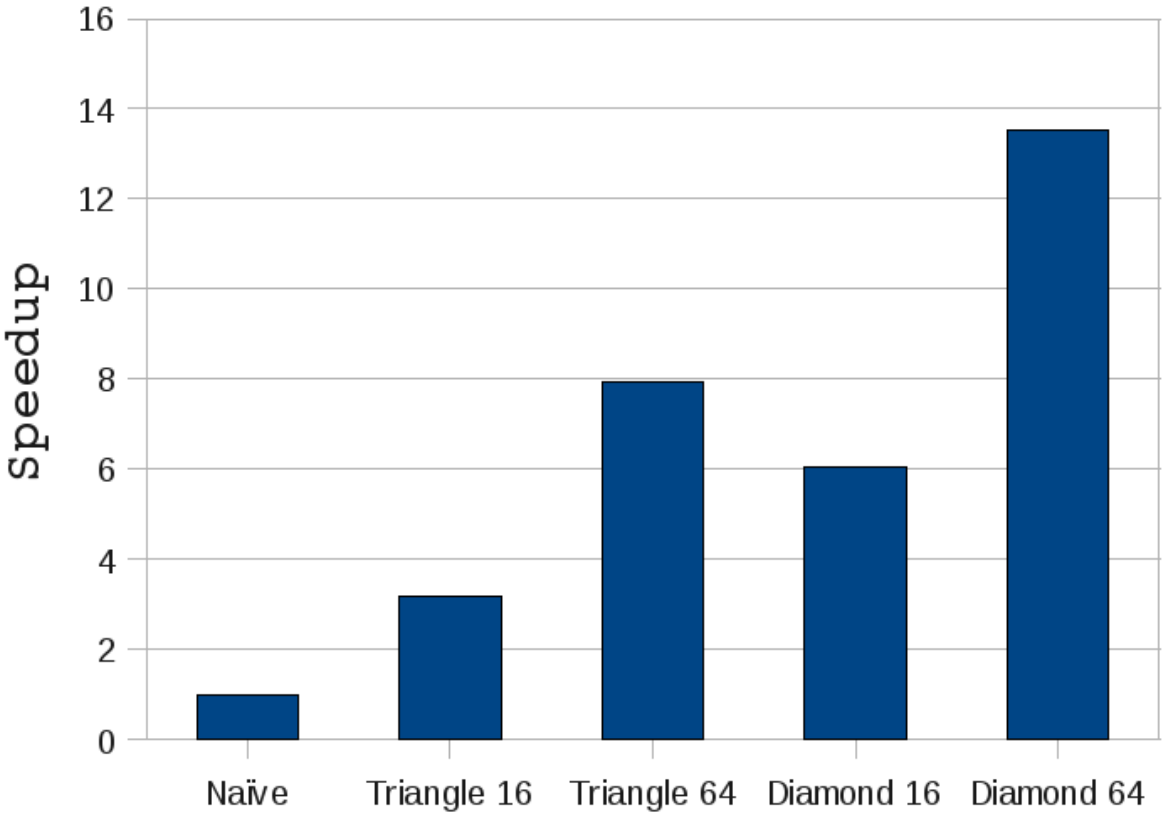

Figure 11: Relative speedup of several tiling approaches in Cyclops-64

The triangle version is the first approach at data reuse. The iteration space was partitioned as shown in Figure 12. The thick black lines represent the memory that has to be loaded for a tile. The thick line is shown for the two kinds of triangles used.

Figure 11 shows that reuse of memory is of paramount importance to achieve better performance. Two tile sizes are shown for triangle tiling. It can be seen that as the tile size increases, the overall performance of the application is increased greatly. Those first results suggests that full reuse of the memory loaded plays a key role in the execution of the application.

The Diamond version results, also shown in Figure 11, confirm our predictions about how data reuse is fundamental to achieve greater performance. Using diamond tiling produces twice as much data reuse when compared to triangle tiling, and is reflected in an overall increase of performance that closely follows the amount of data reuse.

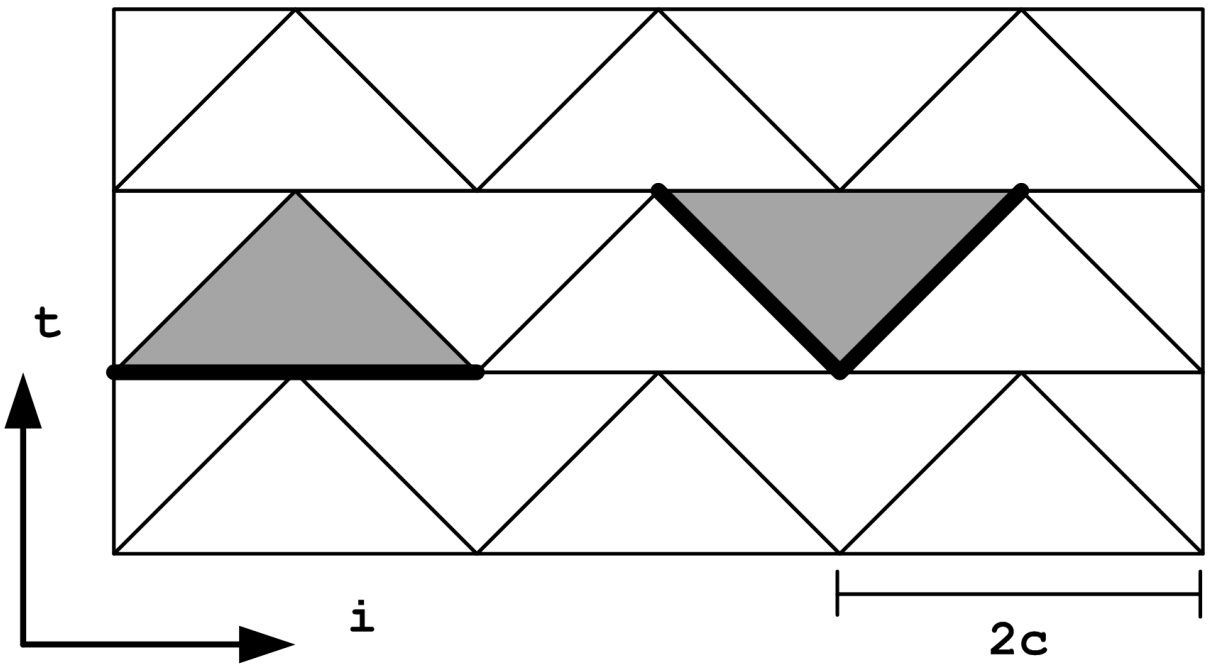

Figure 12: Triangle Tiling

To verify our hypothesis about the performance of tiles as a function of their shape, we have produced the varying shape version, where tiling shape was parameterized as shown in Figures 7 and 8. Speedup results confirming our analysis are presented in Figure 13.

It should be noted that at the bottom and top of each diamond tile, there are more control flow instructions per node executed. To test the effects of this relative increase in instructions to floating point operations, we ran the FDTD application with several tile shapes. The results of our experiments, shown in Figure 13, show that for larger tile sizes, the extra amount of control flow instructions does not play a key role in the performance of the application. Instead, the amount of data reuse keeps dominating the performance.

The results presented here have been obtained by applying previously known compiler optimizations to the FDTD application. However, there is a significant difference with previous approaches since optimizations such as tiling or loop skewing, when applied in many core architectures, have different results than when they are used in traditional multi core architectures. The evidence provided by the results of our optimizations shows that loop transformations and new tiling shapes such as diamonds can be integrated into many-core compilers to produce good performance improvements. To do so, each transformation needs to be associated with new cost functions that would reflect the characteristics of each optimization as shown here.

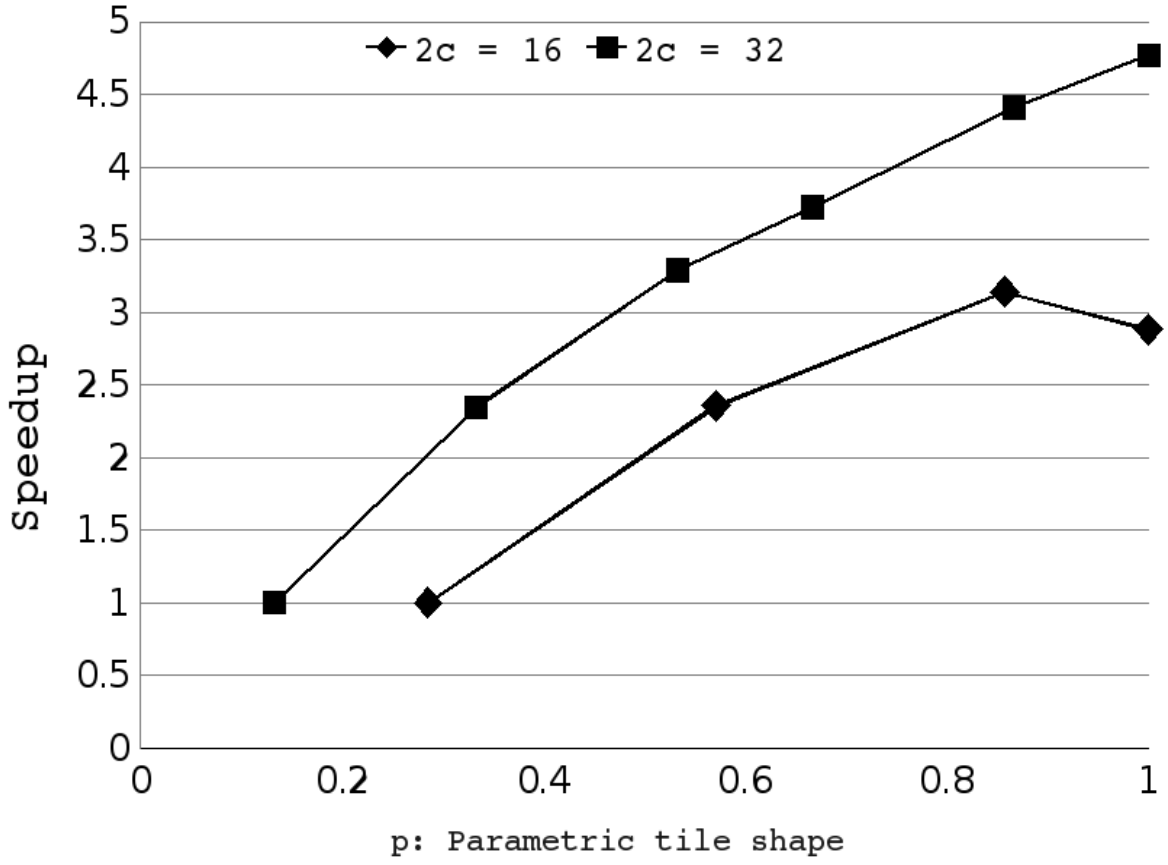

Figure 13: Performance speedup as a function of the tile shape
Performance speedup of FDTD for several tile shapes. Parameter p determines the shape of the tile, as shown in Figure 8. The performance reference is a tile with an absolute height of 2.

6 Related Work

Directly related applications of FDTD that share common properties with the work published here were described in detail in Section 2. We have extended their techniques by combining several optimization approaches and by proposing new tiling shapes.

There are a number of other FDTD implementations that use other approaches such as multiple chips or other kind of parallelization technique. In particular, MPI implementations of FDTD are common [17], [18] among the high performance computing community. Such works are outside of the scope of this paper since they use multiple chips and do not report tiling with sufficient details as to be compared fairly with our results.

7 Conclusions and Future Work

We have presented an efficient mapping of a common stencil application to Cyclops-64, a revolutionary many-core architecture. We have employed a number of loop transformations on the original problem to enable parallel tiling. By examining the data dependencies across iterations of the main computational loop we have proposed a tiling shape that optimizes the reuse of on-chip memory and lowers the bandwidth requirements of the application.

The main memory bandwidth required by the processor chip has been reduced by a factor $O(c)$ of the on-chip memory size. We have explored other tiling shapes and have concluded that on the limit, maximum reuse of on-chip memory yields the best effects in terms of performance. Our results apply to situations where each location in on-chip memory is fully manageable by the user.

Investigations conducted through our tests show that for architectures such as Cyclops-64, with simple control flow behavior, the added program complexity introduced by tiling shapes that are not square is negligible when compared to the performance gains brought by on-chip memory reuse.

Our results show that our tiling technique along with our loop transformations can be applied to existing implementations of FDTD and stencil computations to further increase their performance. In the future, we will continue our research by upgrading state of the art implementations of FDTD to demonstrate the usefulness of our contributions. We will apply other optimizations such as loop unrolling, instruction scheduling and register tiling in the style of [8] to get a fully optimized algorithm.

In the future, we will investigate the possibility to improve our tiling technique by reusing the on-chip memory across phases. Such an implementation is likely to require dataflow techniques that would have to be developed.

We are conducting ongoing research on a mathematical formulation that would be able to extend the results presented here using a one dimensional array to multidimensional arrays. Preliminary results up to three dimensions indicate that this line of work will provide good results in the near future.

Research on synchronization techniques for parallel tiling has to be further pursued. In the results presented here, synchronization was done using global barriers. It is known [21] that the use of global barriers is inefficient since all threads are forced to wait for termination of the slowest thread. The design of producer-consumer techniques for stencil applications will eliminate the wasted cycles where some threads wait for all the threads' dependencies to be ready, and would require several new contributions in the dataflow field, including a good understanding of how to automatically percolate the data [22] so that threads do not waste computing cycles waiting for memory loads to complete. The authors will continue working to solve those issues so that better alternatives to global barriers become available to application developers and computer architects.

Acknowledgment

This work was supported by NSF (CNS-0509332, CSR-0720531, CCF-0833166, CCF-0702244), and other government sponsors. We thank all the members of CAPSL group at University of Delaware. We thank Ge Gan, Robert Pavel and Aaron Landwehr for their comments and their valuable feedback.

References

[1] W. Zhu, "Synchronization state buffer: Supporting efficient fine-grain synchronization on many-core architectures," in *In The 34th International Symposium on Computer Architecture*, 2007.

[2] 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.

[3] A. Tavlove, *Computational Electrodynamics*, 1995.

[4] 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.

[5] G. Schiavone, I. Codreanu, R. Palaniappan, and P. Wahid, "Fdtd speedups obtained in distributed computing on a linux workstation cluster," *Antennas and Propagation Society International Symposium, 2000. IEEE*, vol. 3, pp. 1336–1339 vol.3, 2000.

[6] C. Guiffaut and K. Mahdjoubi, "A parallel fdtd algorithm using the mpi library," *Antennas and Propagation Magazine, IEEE*, vol. 43, no. 2, pp. 94–103, Apr 2001.

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

[8] I. Venetis and G. Gao, "Mapping the lu decomposition on a many-core architecture: Challenges and solutions," *Proceedings of the 2009 ACM International Conference on Computing Frontiers*, 2009.

[9] J. Proakis and D. Manolakis, *Digital Signal Processing*, 2006.

[10] K. R. Koch, R. S. Baker, and R. E. Alcouffe, "Solution of the first-order form of the 3d discrete ordinates equation on a massively parallel processor." *Transactions of the American Nuclear Society*, pp. 65:198–199, 1992.

[11] Y. Song and Z. Li, "New tiling techniques to improve cache temporal locality," *SIGPLAN Not.*, vol. 34, no. 5, pp. 215–228, 1999.

[12] 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*, pp. 171–180, 2000.

[13] 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.

[14] M. Frigo and V. Strumpen, "The memory behavior of cache oblivious stencil computations," *J. Supercomput.*, vol. 39, no. 2, pp. 93–112, 2007.

[15] S. Hagness, A. Taflove, and J. Bridges, "Two-dimensional fdtd analysis of a pulsed microwave confocal system for breast cancer detection: fixed-focus and antenna-array sensors," *Biomedical Engineering, IEEE Transactions on*, vol. 45, no. 12, pp. 1470–1479, Dec. 1998.

[16] J. Bourgeois and G. Smith, "A complete electromagnetic simulation of the separated-aperture sensor for detecting buried land mines," *Antennas and Propagation, IEEE Transactions on*, vol. 46, no. 10, pp. 1419–1426, Oct 1998.

[17] Z. Yu, D. Wei, and L. Changhong, "Analysis of parallel performance of mpi based parallel fdtd on pc clusters," *Microwave Conference Proceedings, 2005. APMC 2005. Asia-Pacific Conference Proceedings*, vol. 4, pp. 3 pp.–, Dec. 2005.

[18] W. Yu, X. Yang, Y. Liu, L. ching Ma, T. Sul, N.-T. Huang, R. Mittral, R. Maaskane, Y. Lu, Q. Che, R. Lu, and Z. Su, "A new direction in computational electromagnetics: Solving large problems using the parallel fdtd on the bluegene/l supercomputer providing teraflop-level performance," *Antennas and Propagation Magazine, IEEE*, vol. 50, no. 2, pp. 26–44, April 2008.

[19] J. Durbano and F. Ortiz, "Fpga-based acceleration of the 3d finite-difference time-domain method," *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pp. 156–163, April 2004.

[20] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* New York, NY, USA: ACM, 1989, pp. 25–35.

[21] F. Petrini, D. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q," *Supercomputing, 2003 ACM/IEEE Conference*, pp. 55–55, Nov. 2003.

[22] A. Jacquet, V. Janot, C. Leung, G. R. Gao, R. Govindarajan, and T. L. Sterling, "An executable analytical performance evaluation approach for early performance prediction," in *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing.* Washington, DC, USA: IEEE Computer Society, 2003, p. 268.1.