Locality Aware Concurrent Start for Stencil Applications

Sunil Shrestha
Guang R. Gao
University of Delaware
sunil@udel.edu, ggao@capsl.udel.edu

Joseph Manzano Andres Marquez
John Feo
Pacific Nothwest National Laboratory
{joseph.manzano,andres.marquez,john.feo}@pnnl.gov

Abstract

Stencil computations are at the heart of many physical simulations used in scientific codes. Thus, there exists a plethora of optimization efforts for this family of computations. Among these techniques, tiling techniques that allow concurrent start have proven to be very efficient in providing better performance for these critical kernels. Nevertheless, with many core designs being the norm, these optimization techniques might not be able to fully exploit locality (both spatial and temporal) on multiple levels of the memory hierarchy without compromising parallelism. It is no longer true that the machine can be seen as a homogeneous collection of nodes with caches, main memory and an interconnect network. New architectural designs exhibit complex grouping of nodes, cores, threads, caches and memory connected by an ever evolving network-on-chip design. These new designs may benefit greatly from carefully crafted schedules and groupings that encourage parallel actors (i.e. threads, cores or nodes) to be aware of the computational history of other actors in close proximity.

In this paper, we provide an efficient tiling technique that allows hierarchical concurrent start for memory hierarchy aware tile groups. Each execution schedule and tile shape exploit the available parallelism, load balance and locality present in the given applications. We demonstrate our technique on the Intel Xeon Phi architecture with selected and representative stencil kernels. We show improvement ranging from 5.58% to 31.17% over existing state-of-the-art techniques.

1. Introduction

As shared resources such as memory bandwidth and cache sizes increase at much slower pace than processing resources such as cores and threads, maximizing the utilization of these resources is very important to achieve higher performance. Compiler optimization techniques such as loop transformations have proven to be very effective. Such techniques improve reuse of data in multiple cache hierarchies and help to hide memory latency. In addition, loop transformations can also be performed to maximize parallelism and load balance. However, when there exist complicated loop carried dependencies, such transformations still operate under the paradigm that coarse parallelism is the norm. For example, in the case of hierarchically tiled codes, the inner tiles are usually executed sequentially without considering possible collaboration and reuse among other parallel actors.

Stencil computations are a computational intensive class of kernels that are used in many scientific and engineering codes. For some of these kernels, the computations are iterated over time until a solution is found. Since stencil computations involve calculating values using neighboring elements, there are plenty of opportunities for data reuse. However, when stencil applications are tiled, the dependencies flow across neighboring tiles. In lower dimensionality stencils, the amount of communication required is low. Thus, the interactions between the tiles do not greatly affect the reuse of the application, i.e. the stencil elements needed for calculation are a short stride away. However, in higher dimensionality stencils, the interactions between the elements involved are farther and farther away. This greatly affects reuse since their accesses would affect the memory hierarchy behavior.

Although there exist many stencils with different computational intensities, many of them feature an execution in which an entire dimension of the iteration space can be executed concurrently. Such stencils have been heavily studied to maximize parallelism and load balance such that processing resources are fully utilized. Techniques like diamond tiling [2, 20], split and overlapped tiling [17] were created to take advantage of such feature. However, so far such techniques have not been able to fully exploit locality at multiple levels of a memory hierarchy.

```
for (int t=0; t<T;t++){
   for (int i=1; i<=N;i++){
      A[t+1][i] = alpha*(A[t][i+1]
                - beta*A[t][i] + A[t][i-1]);
   }
}
```

Figure 1. Heat-1d Example

Figure 1 shows a **for** loop for one dimensional heat equation with the time dimension added in the array structure. We will use this simplified version of the heat-1d loop as a running example throughout the paper. Figure 2 shows

2015 IEEE/ACM International Symposium on Code Generation and Optimization
978-1-4799-8161-8/15/$31.00

a very efficient diamond tiling technique that maximizes parallelism. Although very efficient, such technique doesn't incorporate the concept of multiple levels or groupings of threads that can take advantage of each other's data movement. The reason for this is two fold. Firstly, when tiled hierarchically, such tiling technique doesn't exploit intra-tile parallelism. Secondly, when thread assignment is oblivious to the possibility of locality and reuse, threads are scheduled far enough in memory to miss out on locality opportunities across threads.

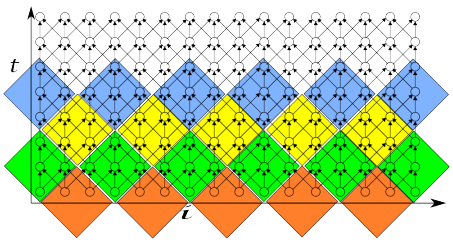


Figure 2. Diamond Tiling for Heat-1d

During parallel execution, multiple threads work in close proximity in time and space and hence provide opportunities for reuse not only for themselves but also for neighboring threads. In this paper, we attempt to explore this with our novel tiling technique that enables threads to work in a collaborative fashion. Such technique exploits locality of outer tiles without compromising inner tile parallelism. Due to their properties (i.e. easier to analyze, wide spread availability and use, etc), this paper mainly focuses on stencils with symmetric dependencies. The contributions of this paper includes:

- Highly parallel tiling technique that exploits concurrent start at multiple levels.
- Detail analysis of such technique at different levels of the memory hierarchy

The paper is divided as follows. Section 2 provides related work. Section 3 explains the background information and notations used in the paper. Section 4 explores tile shape requirements to exploit intra-tile parallelism and our novel tiling approach without compromising concurrent start. Section 5 and 6 provides experimental results and case studies/discussion. Finally, Section 7 concludes the paper.

2. Related Work

Many optimizations to improve data locality, parallelism and load balance aim to the improve overall performance [7, 14, 18, 24, 25] but they might not consider the impact of thread interactions. Iteration space tiling that aggregates iterations from different dimensions have been very useful to exploit reuse at multiple levels of a memory hierarchy. This approach is applicable to a wide range of applications and it has also been used extensively to improve locality and parallelism of scientific codes that use stencil computations. Compiler optimization techniques and tools using the polyhedral framework [11] have also gained significant attention with their ability to analyze and generate code automatically. As a result, a tool like PLUTO [6] has provided a convenient way to generate tiled code that is optimized for locality and communication.

Since stencils are at the heart of many scientific computation and are very compute intensive, there has been a considerable body of research on how to improve the performance of these codes [12, 15]. Some optimizations are focused mainly on stencil computation targeting higher order stencils where locality optimization is more important [8, 21]. Using a domain specific language that is translated to parallel Cilk code, Pochoir [23] provides efficient cache oblivious algorithm for stencils.

As many stencil codes have the property in which one face of the iteration space can run concurrently, optimizing stencils for load balance has gained popularity. Krishnamoothry [17] proposed split tiling and overlapped tiling for stencil computations that are able to provide concurrent start. Although these techniques are interesting, they do not look into the possibility of exploiting locality at multiple levels of the memory hierarchy, as we do.

Orozco [20] showed the effectiveness of diamond tiling on a FDTD application using a many core architecture. Similary, Bandishti [2] developed a diamond tiling technique using PLUTO as a base tool to optimize for parallelism and load balance. The technique is driven by data dependence and hence creates diamond shaped tiles which are very effective for applications with concurrent start. Besides diamond tiling, there also exists other tile size and shape optimizations [1, 9, 13]. Shrestha [22] introduced jagged tiling technique for pipeline parallel applications and showed its benefits on a Gauss-Seidel stencil. Our approach leverages on both diamond and jagged tiling to improve locality further with hierarchical jagged polygon tiling technique without compromising parallelism.

Optimizations focusing on data to improve locality and reuse have also been applied to improve performance. Kodukula [16] uses blocking of data based on its flow through the memory hierarchy. This approach shackles statements associated with a data block and execute them to improve locality. Similary, Bikshandi [5] introduced hierarchical tiled array that manipulates tiles using array operations. Bashkaram [3] used explicit data movement for improving locality on scratchpads. Meng [19] proposed symbiotic affinity scheduling for improving inter-thread cache sharing. However, such approaches have not been applied to stencil computations.

Although very efficient, most of the existing compiler work focus mainly on a single thread of execution. When threads work in close proximity in time and space, there exists an opportunity of a fine-grain collaborative view on code generation. In contrast, when threads interfere with

each other, performance can plummet rapidly. The approach presented in [10] exploits inter and intra (with threading) node parallelism using tiling. It takes advantage of vector register blocking and uses NUMA aware allocation showing benefits on stencils. Despite some similarities, our approach focuses on exploiting intra-tile parallelism and improving data reuse compared to thier intra-node parallelism. Stencil computations are, by design, computational intensive codes with tremendous opportunity of data reuse. When data movement and usage are carefully architected to improve locality among multiple threads, performance can be improved significantly.

3. Background

Our current framework relies heavily on polyhedral theory. Thus, a short overview of the main concepts in this framework is needed so that we can introduce our technique. In this section, we briefly explain basic polyhedral terminology and existent tiling techniques used to maximize parallelism and load balance.

3.1 Basics of Polyhedral Theory

Under polyhedral terminology, loops are characterized by their iteration spaces. In which each level of the loop would represent a dimension of a bounded polyhedra (i.e. polytope). Given an iteration space $\vec{v}$, a hyperplane $\phi(\vec{v})$ is a $n-1$ dimensional affine subspace in n dimensional space. For statements S, a hyperplane $\phi_s(\vec{v})$ with dimensionality m and normal $(c_1c_2...c_m)$ represents an affine transformation of the form,

$$\phi_s(\vec{v}) = (c_1c_2...c_m).\vec{v} + c_0 \quad (1)$$

For given 'k' statements, in order for statement-wise hyperplane $(\phi_{s_1}, \phi_{s_2}...\phi_{s_k})$ to be a legal tiling hyperplane, the distance between source 's' and target 't' along every dependence edge has to be strictly positive such that,

$$\phi_{S_i}(\vec{t}) - \phi_{S_j}(\vec{s}) \geq 0 \quad (2)$$

When a combination of 'm'[1] hyperplanes, represented by $\phi^1, \phi^2...\phi^m$, form tiles, they are self-contained i.e. dependencies for statements within tiles are either satisfied or can be satisfied within.

3.2 Diamond Tiling

Not all valid tiling hyperplanes lead to the best scheduling strategy. For example, in Figure 1, all elements along i can be executed concurrently at different time steps t in the original iteration space. However, in order to have concurrent start in the tiled domain, there must exist a face that can be executed concurrently such that its normal $\vec{f}$ carries all dependencies. Valid tiling hyperplanes for this example to create a diamond shaped tile are (1,-1) and (1,1) as shown in Figure 2.

[1] where m is less or equal to the number of dimensions of the iteration space

3.3 Condition for Concurrent Start

Given the set of vectors $\vec{x_1}, \vec{x_2}, \vec{x_3}...\vec{x_k}$, a conical combination is a vector of the form

$$\lambda_1 x_1 + \lambda_2 x_2 + ... + \lambda_k x_k \quad (3)$$

When λ_is are strictly positive, such combination becomes a strict conical combination. By using such a strict conical combination of all hyperplanes $\phi^1, \phi^2...\phi^k$, a hyperplane ϕ representing a face with normal $\vec{f}$ can be found such that,

$$\phi = \theta\vec{f} = \lambda_1\phi^1 + \lambda_2\phi^2 + ... + \lambda_k\phi^k \; where \; \theta, \lambda_i \in \mathbb{Z}^+ \quad (4)$$

Although, concurrent start can be used to exploit $n-1$ degrees of concurrency, such codes in practice are very complex and don't provide much performance. Exploiting just one degree of concurrent start using the first two hyperplanes provides partial concurrent start along one face of the iteration space. Thus, partial concurrent start can be exposed using a simplified version of Equation 4:

$$\phi = \lambda_1\phi^1 + \lambda_2\phi^2 \quad (5)$$

Interested readers are strongly recommended to read reference [2] to get more information about hyperplane selection for diamond tiling.

3.4 Jagged Tiling

Jagged tiling for inner concurrent start for pipeline parallel applications has been studied previously [22]. Instead of using the same tiling hyperplanes for tiling at different levels, outer tiles are created by finding at least one tiling hyperplane that carries all dependencies in the L1 tiled domain. This is done by finding a conic combination of the first two hyperplanes as shown in Equation 5. For example, Figure 3 shows a pictorial view of standard jagged tiling with tiling hyperplane (1,0) and (0,1) for L1 tiling and (1,1) and (0,1) for L2 tiling.

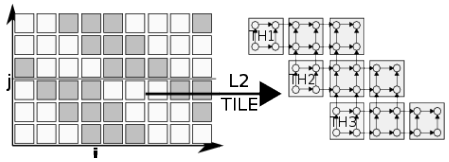


Figure 3. Jagged Tiling for Pipeline Start

In this paper, we take advantage of both diamond tiling and jagged tiling to exploit one degree of concurrent start for each level of tiling to provide outer and inner concurrent start.

4. Jagged Polygon Tiling

In this section, we introduce a novel jagged polygon tiling technique that exploits locality further for applications with concurrent start without compromising parallelism. Figure 5

provides a pictorial view of the jagged polygon tiling for the kernel shown in Figure 1. It uses a hierarchical approach of tiling in which outer tiles are designed to take advantage of the highest level[2] of the memory hierarchy i.e. (L2 cache). Threads working within L2 tiles thus have an opportunity to share data with each other and maximize data reuse as a group of threads working in close proximity. Such threads use a fine-grain execution strategy as shown in Section 4.3 to keep communication overhead to a minimal.

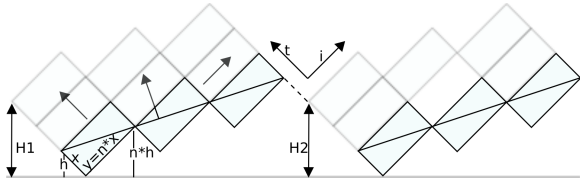


Figure 4. Tile Sizes for Jagged Polygon Tiling

4.1 Tile Shape and Sizes

Standard diamond tiling uses symmetric diamond shaped tiles that enable concurrent execution along at least one face of the iteration space. Although very effective for load balance and parallelism, such tiles are not able to provide intra-tile concurrent start when tiled hierarchically. Since jagged polygon tiling is designed to provide a locality aware concurrent start with intra-tile parallelism to further improve data locality, we provide a simple solution to solve this problem.

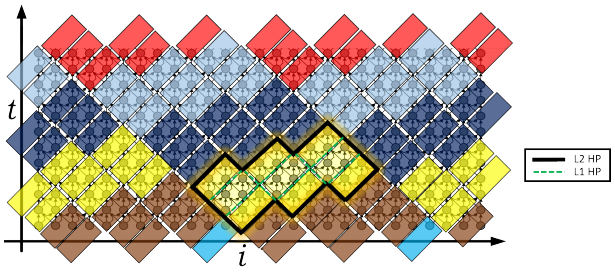


Figure 5. Jagged Polygon Tiling for Heat-1d. Tiling hyperplanes (1,-1) and (1,1) (for L1 Hyperplane) in original domain and (1,1) and (1,0) (for L2 Hyperplane) in the tiled domain

In Figure 4 (which shows two L2 tiles extracted from Figure 5), let the tiled iteration space be t and i and size of a L1 tile be x and y such that $y = nx$. This implies that it creates a rectangular polygon in which if one side has height h (towards x), the opposite side has height $n * h$. If we create jagged L2 tiles with j_t and j_i number of L1 tiles in t and i direction using Algorithm 1, two consecutive jagged polygon tiles must stand at the same level, or for this example at the same height, to be able to provide concurrent start. Let $H1$ and $H2$ be the level at which two consecutive tiles start. Then, $H1 = j_t * h$ and $H2 = j_i * (n * h - h) = j_i * h(n-1)$

For, concurrent start $H1$ has to be equal to $H2$, hence

$$j_t * h = j_i * h(n-1) \tag{6}$$

When n = 2, $j_t = j_i$. This means that when $y = 2 * x$, the number of L1 tiles required to form L2 tiles has to be equal in both direction (e.g, 3x3 L1 tiles in Figure 5). Although, other sizes can be extracted using Equation 6, we use L1 tiles in which one side is double the size of the other.

4.2 Generating Jagged Tiles

Jagged tiles are generated using the polyhedral optimization tool PLUTO [6] (PLUTO diamond is an enhancement of PLUTO to generate code for load balance e.g. diamond shaped tiles for concurrent start). PLUTO is designed to take a C code as an input and generate an OpenMP code optimized for communication across tile boundaries. It uses CLOOG [4] for code generation which scans the iteration space in a global lexicographic ordering. Such ordering is given by an affine transformation function, also known as a scattering function. The code generator is oblivious to any dependence information and hence uses the original iterator in absence of a scattering function.

We create L1 tiles using the approach used in PLUTO for diamond tiling. It uses an iterative scheme to find hyperplanes [2] that satisfy Equation 4. We leverage on their method while using a different tile shape, i.e. $y = 2 * x$. Once L1 tiles are created, given 'm' L1 hyperplanes, we use the condition given by Equation 5 to expose at least one hyperplane φ^1 with concurrent start such that,

$$\varphi^1_{S_i}(\vec{t}) - \varphi^1_{S_j}(\vec{s}) \geq 1 \tag{7}$$

For clarity, we represent all original hyperplanes by ϕ, L1 hyperplanes by φ and L2 hyperplanes by Φ. The algorithm to generate jagged polygon tiles is shown in Algorithm 1.

For example, let's assume that we want to create a jagged tile for Figure 1 with tile sizes 1024 x 2048 for L1 and 4x4 (L1 tiles) for L2. The tiling hyperplanes found by PLUTO are (1,1) and (1,-1). We use these hyperplanes to create inner L1 tiles with tile size 1024 and 2048 along iterators 't' and 'i'. These tiles can be viewed as supernodes for next level tiling and can be used to create outer L2 tiles using the supernode iterators 'T' and 'I'. The domain with these supernode iterators becomes the tiled domain, where using hyperplanes (0,1) and (1,0) as tiling hyperplanes results in the same hyperplanes as the ones used to create Level 1 tiles. Instead, we use hyperplanes (1,1) (which satisfies the partial concurrent start condition) and (1,0) with a tile size of 4. Once the tiles are created and scattering functions for L2 tiles ($c1_{L2}, c2_{L2}$), L1 tiles ($c1_{L1}, c2_{L1}$) and the original domain ($c1, c2$) are updated, the parallel code is generated using CLOOG. The resulting Domain and Scattering function for heat-1d is shown below.

[2] farthest from the core

Algorithm 1 Generating Jagged Polygon Tiles

Input: Given: (a) diamond tiling hyperplanes $\phi_s^i, \phi_s^{i+1}...\phi_s^{i+m-1}$ (b) Original Domain D_s (c) L1 tile sizes $tL1_i, tL1_{i+1}...tL1_{i+m-1}$ and L2 tile sizes $tL2_i, tL2_{i+1}...tL2_{i+m-1}$ (satisfying tile size condition from Equation 6 for $tL2_i, tL2_{i+1}$)

1: Tile for L1 using ϕ, D_s, TS_1 ▷ At this point, all $\varphi_{L1s(s)}$ are created

2: Update $Domain$ constraint to get hyperplane $\varphi^1_{L1s} \rightarrow \varphi^1_{L1s} + \varphi^2_{L1s}$ such that $\varphi^1_{L1s(t)} - \varphi^1_{L1s(s)} \geq 1$ leaving other hyperplanes as is

3: Tile for L2 using φ, D_s, TS_2 ▷ At this point, all $\Phi_{L2s(s)}$ are created

4: Perform Unimodular transformation on L1 scattering supernode: $\varphi T^1_{L1s} \rightarrow \varphi T^1_{L1s} + \varphi T^2_{L1s}$ to satisfy Equation 5.

5: Perform Unimodular transformation on L2 scattering supernodes: $\Phi T^1_{L2s} \rightarrow \Phi T^1_{L2s} + \Phi T^2_{L2s}$ to satisfy Equation 5.

Output: Updated domain and scattering function

$$
\begin{array}{rr}
\textit{Domain} & \textit{Scattering} \\
0 \leq t \leq T-1 & c1_{L2} = T_{L2} + I_{L2} \\
1 \leq i \leq N-1 & c2_{L2} = T_{L2} \\
1024T_{L1} \leq t - i \leq 1024T_{L1} + 1023 & c1_{L1} = T_{L1} + I_{L1} \\
2048I_{L1} \leq t + i \leq 2048I_{L1} + 2047 & c2_{L1} = T_{L1} \\
4T_{L2} \leq T_{L1} + I_{L1} \leq 4T_{L2} + 3 & c1 = t \\
4I_{L2} \leq I_{L1} \leq 4I_{L2} + 3 & c2 = t + i
\end{array}
$$

4.3 Fine-Grain Execution Framework

In this section, we provide a brief description of a fine-grain execution framework we use in order to reduce communication overhead across threads. With the goal of taking advantage of locality across threads we combine threads into different groups. Each thread group work together within L2 tile taking advantage of intratile parallelism, L2 locality and sharing data brought into caches by threads within the group.

Stencil computations have very regular dependencies and are repeatable across all L2 tiles. In other words, for any given L2 tile ready to execute, the same face (represented by L1 tiles) in its iteration space are ready to execute. To simplify the execution, we represent a tiled domain dependencies by a set of bits which are collectively updated by a group of threads working together within the L2 tile. Each thread performs atomic bit-wise operations to create a task mask. Every nonzero bit in the task mask represents a L1 task ready to execute. Such execution happens in a highly parallel fashion and all required updates are done using atomic operations to minimize synchronization overhead. Figure 6 shows a pictorial view of such execution.

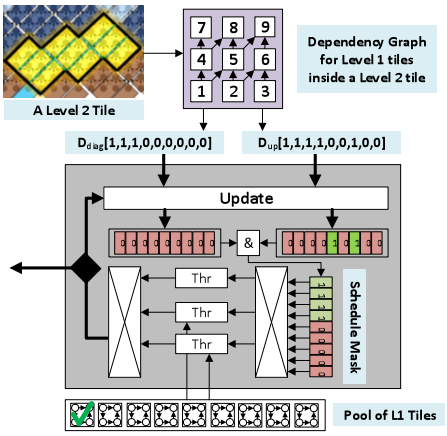


Figure 6. Fine Grain Execution Example

5. Experiments

We used a Intel Xeon Phi 7110P coprocessor as our experimental platform. Each coprocessor has 61 cores arranged in a ring running at 1.1 GHz. Each core has a 32KB L1 cache and a 512KB L2 cache shared by 4 hyper threads. Besides their local L2 cache, threads have access to the L2 caches of every core within the ring. A latency to access data within the ring varies and can be as high as accessing data from the memory. However, only when there is a cache miss in both local and shared L2s, the request is served by the memory. For our experiments, we use 60, 120, 180 and 240 threads. These sizes represent a quarter, half, three quarter and the full machine minus the core that is reserved for the operating system. In this way, we try to minimize the interference from the operating system.

On the software side, we picked five stencil kernels that exhibit concurrent start. In addition to our running example (Heat-1d), we also use Heat-2d, Heat-3d, Jacobi-2d, 7 point-3d stencils to show the efficiency of our techniques experimentally. All these kernels have a face along the iteration space where they can be executed concurrently. We selected a range of tile sizes based on full tile sizes, number of available cores, caches sizes, memory capacity, etc. From this range, we selected the best results as our experimental sizes. Table 1 shows the kernels and the sizes we used for our experiments.

We compare our fine-grain (FG) jagged polygon tiling approach with the diamond tiling generated by PLUTO (PLT). To test the effect of thread scheduling on the given techniques, we use different thread mappings supported by the

Application	Size(N)	Size(T)
Heat-1d	10000000	10K
Heat-2d	11504x11504	2K
Heat-3d	480x480x480	100
Jacobi-2d	11504x11504	2K
7point-3d	480x480x480	100

Table 1. Applications and its sizes

architecture. For PLUTO generated OpenMP code we used 'BALANCED' and 'COMPACT' mode. In 'BALANCED' mode, threads are distributed equally among cores to reduce load imbalance. This approach has the advantage of using physical cores first in which resources are not contended, in contrast to using hyperthreads. In 'COMPACT' mode, all hyperthreads within a core are assigned before moving to the next one. For our fine-grain framework, we use mappings equivalent to 'SCATTER' and 'COMPACT' (with pthread_setaffinity_np) mode. In our framework's scatter mode, the scheduler assigns threads to different cores and hence a group is formed by threads from different cores. On the other hand in compact mode, hyperthreads within the core form a group. Such approach has the advantage of data reuse among threads when they work in close proximity in time and space.

Table 2 shows the performance for each of the selected kernels in GFLOPS for both PLT and FG along with the selected tile sizes. For FG, a second set of tile sizes represent number of Level 1 (a.k.a L1) tiles in different dimensions. For example, tile size 16x32x256 / 4x4x2 represents a tile with Level 1 tile size 16x32x256 and Level 2 (a.k.a L2) tile size 16*4x32*4x256*2. As shown in this table, the best performance overall occurs when using the jagged tiling framework in compact mode, except for the case of heat 1d. In addition, the best performing cases were seen when 240 threads were used, except for heat-3d (PLT Balanced mode). The improvement over the PLUTO generated code ranges from 5.58% (in the case of Jacobi-2d) to 31.17% (in the case of the 7 point stencil kernel) for the highest performing codes. The last column of Table 2 shows the performance for two level tiling for the heat based kernels. The reduced performance is because of the additional control overhead and idle resources created due to lack of inner parallelism. However, further investigation shows improved performance for Heat-2D when using smaller tiles for both PLUTO and our framework, 127.24 GFLOPS and 140.01 GFLOPS respectively.

The three heat kernels are used as a case study to show how our framework performs with respect to the architecture's memory hierarchy. The kernel's access patterns illustrate the advantage and disadvantages of our approaches. The memory behavior for the three kernels are reported in Figures 7, 8 and 9. These figures showcase the most expensive operations (i.e. misses and remote hits) in each level of the memory hierarchy. An interesting feature of the study on the heat 3d kernel is that the best cases from our initial runs were selected for two tile sizes, named PLT_2 (2x2x4x480) and PLT_3 (3x3x2x480) in the graphs, to compare and contrast the effect of the size of the diamond tile for PLUTO generated code. The PLT_2 cases have better load balance whereas the PLT_3 cases take advantage of locality better, however cannot utilize all available threads. As the stride increases (in higher dimensionality stencils), the jagged tiling framework effectively reduces the accesses to main memory. The presented case studies use the compact schedule as their default. Section 6 discusses the graphs in more detail.

6. The Heat Kernels Case Study

The heat 1d kernel presents the lowest improvement of all the kernels. This behavior can be connected to an increase in L1 cache misses (reflected in the collected counters shown below) presented in our framework. The worst degradation of our framework versus the PLUTO generated code is 10% when using 60 threads and compact scheduling. The highest improvement occurs when we use the balanced and scatter schedules which is around 9%. The difference between the balance and compact schedules can be attributed to the access pattern exhibited by the 1D case. The Intel Phi hardware can take advantage of the contiguous access pattern of this kernel. Our framework behavior might interfere with the prefetcher hardware, especially to the highest memory hierarchy levels. A short discussion of the memory hierarchy counters is presented below.

Figure 7(d) shows the number of cache misses that were served by the memory subsystem. This chart shows that there is a considerable reduction of memory served cache misses (around 93% on average of total accesses are reduced) over all the presented cases. In case of using the entire machine, we have a 96% reduction of memory accesses.

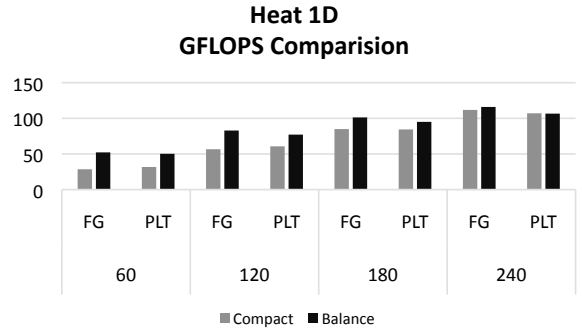
Since the remote cache hits cycle can be as high as access to the memory, a higher number of remote cache hits may result in performance degradation. Figure 7(c) shows also a great reduction of these types of hits when comparing our framework and the PLUTO generated code. This reduction averages around 50% in favor of our framework and a 66% when using the full machine.

However, when examining Figure 7(b), we see that the Level 1 misses are disproportional high (around 2 times the PLUTO number). This rapidly decreases the performance advantages FG has over PLT from the other accesses. However, the reduction on the other level still provides the performance improvement shown in Figure 7(a).

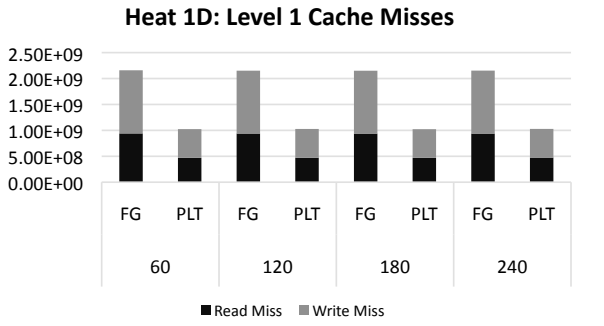
The reason for this behavior is because 1D stencils do not have a locality problem to begin with. The accesses are fairly contiguous and tiles are big. Due to this, the hardware prefetcher can take advantage by bringing data to the level 1 cache. Our framework's thread group might interfere with this behavior resulting in the higher miss rates for level 1.

	PLT			FG				PLT
	GFLOPS		Tile Size	GFLOPS		Tile Size		GFLOPS
Kernel	Balanced	Compact	Best Case	Scatter	Compact	Best Case	Speedup	Hierarchical
Heat-1d	106.59	107.05	4Kx4K	115.95	111.77	1Kx2K / 4x4	8.31%	99.65
Heat-2d	122.42	122.78	16x16x256	131.47	134.95	16x32x256 / 4x4x2	9.91%	109.78
Heat-3d	**59.375**	57.22	3x3x2x480	61.73	74.168	1x2x4x480/4x4x2x1	24.91%	27.91
Jacobi-2d	68.26	68.40	16x16x256	67.94	72.22	16x32x256 / 4x4x2	5.58%	–
7point-3d	31.48	31.82	2x2x4x480	33.46	41.74	1x2x4x480/4x4x2x1	31.17%	–

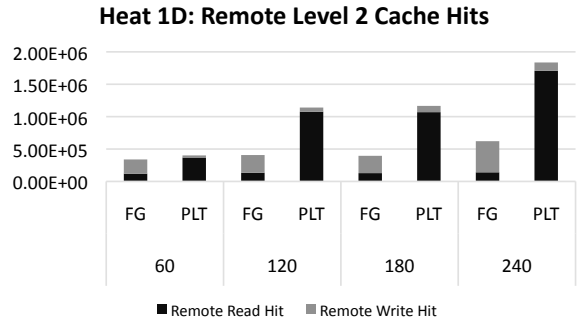
Table 2. Performance for PLT and FG execution with 240 threads. Note: Heat 3d peaks at 180 threads.



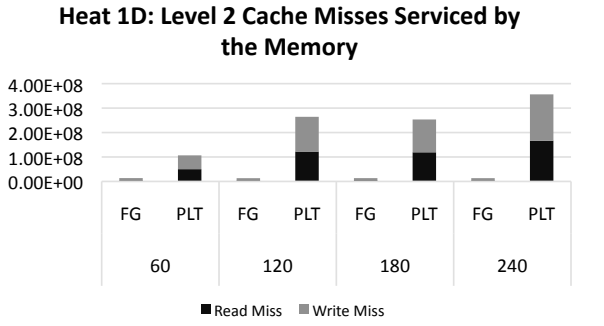
(a) Heat 1D GFLOPS comparison



(b) Heat 1D Level 1 Cache Miss Behavior for PLUTO (PLT) and Jagged Polygon Tiling (FG)



(c) Heat 1D Remote Cache Hit (Local Cache Miss) for PLUTO (PLT) and Jagged Polygon Tiling (FG)



(d) Heat 1D Cache Misses served by the Memory for PLUTO (PLT) and Jagged Polygon Tiling (FG)

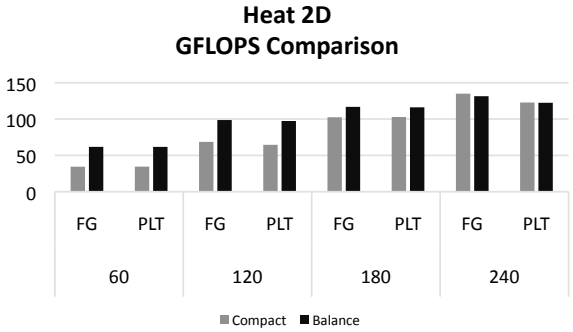
Figure 7. Heat 1D Memory behavior for PLUTO generated code (PLT) and our framework (FG)

Figure 7(a) shows the evolution of performance for different number of threads. FG does better than PLT only when 240 threads are used. Also, it is interesting to note that for heat-1d, balanced mode does better than compact mode. This is because when locality is not much of an issue, having threads from different cores have the advantage as resources are not contended. This behavior however changes for higher order stencils.

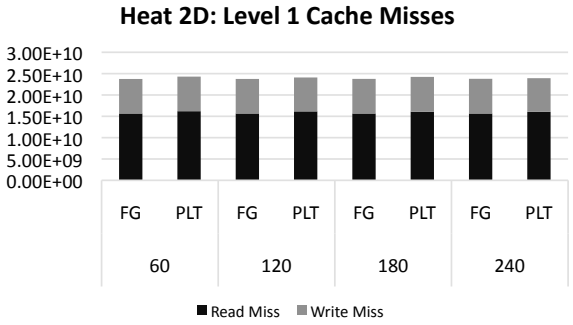
The Level 1 miss rates for the other heat kernels (Figures 8(b) and 9(b)) show a slightly different story with some improvements (around 7% for heat 3d) on the Level 1 misses. As the dimensionality of stencil increases, locality optimization becomes more important. Because of the strided access, PLT is not able to take full advantage of L1 locality as the effectiveness of hardware prefetcher reduces. However with FG, we start to see advantage of grouping threads and accessing Level 2 tiles to improve locality.

In the case of heat 2d, the running times for both our framework and the PLUTO codes are very similar when not using the entire machine. However when using the entire machine, the gains are more visible (around 10% in compact scheduling) as shown in Figure 8(a). These gains are represented in the total reduction of memory accesses showcased by the counters presented below.

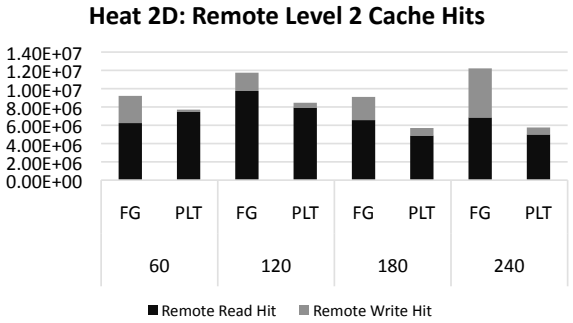
Figure 8(d) shows the cache misses that were served by the main memory. The chart shows a general trend in reduc-



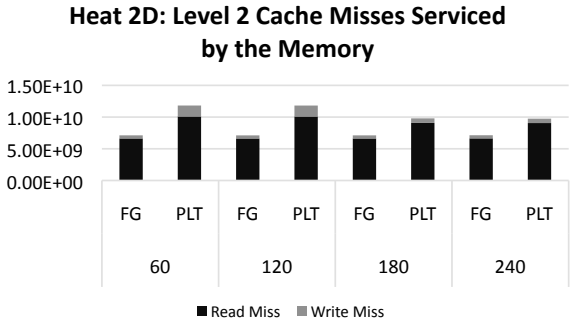
(a) Heat 2D GFLOPS comparison



(b) Heat 2D Level 1 Cache Miss Behavior for PLUTO (PLT) and Jagged Polygon Tiling (FG)



(c) Heat 2D Remote Cache Hit (Local Cache Miss) for PLUTO (PLT) and Jagged Polygon Tiling (FG)



(d) Heat 2D Cache Misses served by the Memory for PLUTO (PLT) and Jagged Polygon Tiling (FG)

Figure 8. Heat 2D Memory behavior for PLUTO generated code (PLT) and our framework (FG)

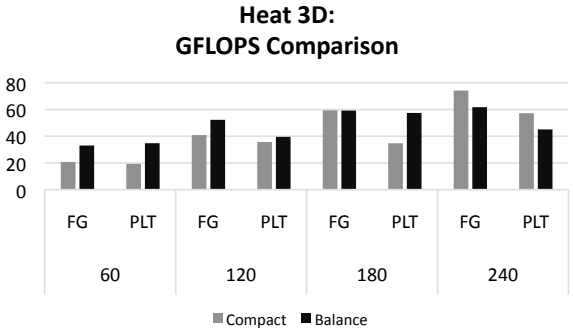
tion of cache misses for all the thread sizes. On average, the reduction is around 33% over the PLUTO generated code with a maximum of 40% when using 60 threads. Although when the number of threads is increased, the gap reduces to up to 27% reduction. This showcases a small interference between the running threads that might be attributed to our framework. Nevertheless, this general trend shows us that more data is kept inside the internal caches of the machine (as was the case with heat 1d).

In Figure 8(c), we showcase the remote level 2 cache hits. These numbers show an increase (instead of a reduction) of the number of remote hits. As stated before, this might be a performance issue due to their latencies. However, since the other two set of misses (L1 misses, L2 misses served by memory) are orders of magnitude higher than these hits, their effect are ameliorated. The increase, in average, is 50% and as high as twice when using the entire machine.

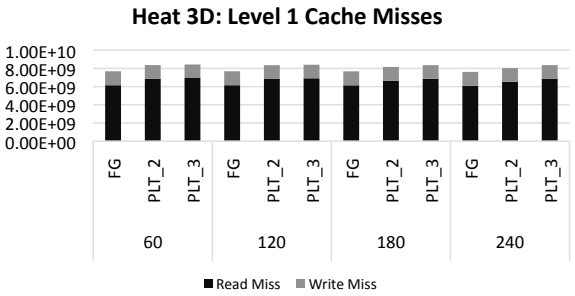
This set of charts shows us that the evolution of the application over the different thread sizes is similar for both level 1 misses and memory served misses. However, the Level 2 remote hits slightly increase with size. This is due to write miss increase resulting from our framework overhead. We are working to reduce this overhead. Figure 8(a) shows the performance evolution for different number of threads. FG, in this case, can take advantage of locality and reuse among thread group and hence shows better performance for all cases.

Figure 9(a) shows the gains in GFLOPS of heat 3d over one of the PLUTO generated cases. Over all cases, our framework has improvements over the PLUTO generated cases with around 30% using the full 240 threads and it peaks at 70% improvement when using 180 threads. The memory characterization is discussed below and shows a great reduction on the memory accesses, remote hits and a slight reduction of level 1 cache misses.

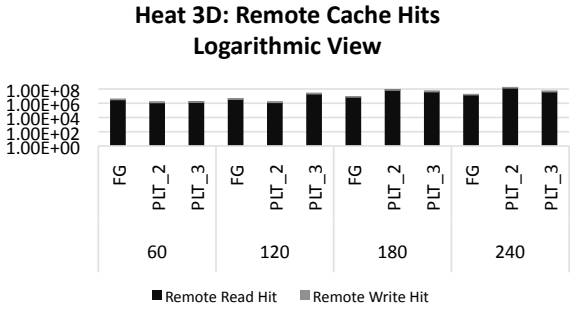
For heat 3d, we have two special cases. The reason is, for PLT, using smaller tile size (2x2x4x480) creates perfect load balance, however, cannot take advantage of locality much. On the other hand tile sizes (3x3x2x480) takes advantage of locality better but is not able to use all processing resources. In the first case, we have a smaller diamond tile (shown as PLT_2). When increasing the number of threads, Figure 9(c) shows that the number of remote cache hits reduces as more threads are added to the mix. Our framework starts (running on 60 threads) with an increase of 60% over the PLUTO generated code. However, as more threads are added, the remote hits reduce to up to 90% over the PLUTO generated



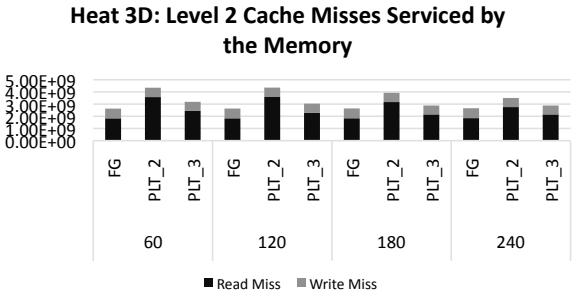
(a) Heat 3D GFLOPS comparison



(b) Heat 3D Level 1 Cache Miss Behavior for PLUTO (PLT) and Jagged Polygon Tiling (FG)



(c) Heat 3D Remote Cache Hit (Local Cache Miss) for PLUTO (PLT) and Jagged Polygon Tiling (FG)



(d) Heat 3D Cache Misses served by the Memory for PLUTO (PLT) and Jagged Polygon Tiling (FG)

Figure 9. Heat 3D Memory behavior for PLUTO generated code (PLT) and our framework (FG)

code. The reduction of memory served cache misses is not as impressive but still considerable (around 34% on average).

For the second tile size (named PLT_3) showcases a larger diamond tile. The story about the memory hierarchy behavior is very similar to the smaller diamond tile. However, the gaps are narrowed with up to 62% reduction for remote level 2 hits and 12% reduction in memory served cache misses. Here the bigger diamond tile takes advantage of the available locality. However, as mentioned earlier, doing so creates load imbalance resulting on idle resources. Because of this, the performance for PLT_3 peaks at 180 (in balanced mode, so all cores can be used) as shown in figure 9(a). Our framework would take advantage of the locality with greater tile sizes as well, but our test platform does not have enough memory to run this experiment.

The charts support the overall premise that there is a reduction of memory accesses over PLUTO generated diamond tiling. Thus, this supports the idea that our framework reuses data better by a hierarchical tiling strategy that manages data movement collectively across threads. This notion is reflected by the performance and supported by the counters. The behavior of heat 2d and heat 3d are representative of the memory behavior of the Jacobi kernel and the 7 Point Stencil, respectively. They present similar behaviors in all the characterizations. For example, for the 7 Point Stencil, there are similar reductions across the board for all the memory related counters (4% reduction in Level 1 caches, 90% reduction in remote hits and 18% in memory accesses when using 240 threads) and an improvement of around 30% in GFLOPS (as shown in Table 2).

7. Conclusion and Future Work

In this paper, we showed a novel hierarchical tiling technique that improves locality and reuse for stencil applications with concurrent start. In addition, with grouping of threads such that they work within dedicated L2 tiles and perform fine-grain execution of L1 tiles, we are able to improve locality by sharing data among closely associated threads. However, such notion of locality improvement is not limited to stencils. We plan to extend our approach to other applications with reuse and architectures that emphasize spatial multithreading over temporal multithreading. In addition, we plan to look into prefetcher interactions with our tiling strategy.

When limited memory resources are contended in multicore systems, performance degradation can be very rapid. In such a case, threads need to be aware of its vicinity and be able to collectively orchestrate memory fetches to improve performance. In this paper, with our tiling technique we ex-

ploit multi-level concurrent start and showed that with a locality aware tiling strategy, threads working in close proximity in time and space can share data and improve data reuse. Such technique can be beneficial for wide range of compute intensive applications in an environment with abundant processing resources.

Acknowledgments

This research was supported in part by DOE ASCR XStack program under Awards DE-SC0008716, DE-SC0008717

References

[1] R. Andonov, S. Balev, S. Rajopadhye, and N. Yanev. Optimal semi-oblique tiling. *Parallel and Distributed Systems, IEEE Transactions on*, 14(9):944–960, 2003.

[2] V. Bandishti, I. Pananilath, and U. Bondhugula. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 40:1–40:11, Los Alamitos, CA, USA, 2012. ISBN 978-1-4673-0804-5.

[3] M. M. Baskaran. *Enhancing locality and parallelism on multicore and many-cire systems*. PhD thesis, The Ohio State University, 2009.

[4] C. Bastoul. Generating loops for scanning polyhedra: Cloog users guide. *Polyhedron*, 2:10, 2004.

[5] G. Bikshandi et al. Design and use of htalib: a library for hierarchically tiled arrays. In *Proceedings of the 19th international conference on Languages and compilers for parallel computing*, LCPC'06, pages 17–32, Berlin, Heidelberg, 2007.

[6] U. Bondhugula and J. Ramanujam. Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer. 2007.

[7] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *ACM SIGPLAN Notices*, volume 30, pages 205–217. ACM, 1995.

[8] K. Datta and K. A. Yelick. *Auto-tuning stencil codes for cache-based multicore platforms*. PhD thesis, University of California, Berkeley, 2009.

[9] J. Dongarra and R. Schreiber. Automatic blocking of nested loops. 1990.

[10] H. Dursun et al. Hierarchical parallelization and optimization of high-order stencil computations on multicore clusters. *The Journal of Supercomputing*, 62(2):946–966, 2012.

[11] P. Feautrier. Some efficient solutions to the affine scheduling problem. i. one-dimensional time. *International journal of parallel programming*, 21(5):313–347, 1992.

[12] M. Frigo and V. Strumpen. The memory behavior of cache oblivious stencil computations. *The Journal of Supercomputing*, 39(2):93–112, 2007.

[13] K. Högstedt, L. Carter, and J. Ferrante. Selecting tile shape for minimal execution time. In *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, pages 201–211. ACM, 1999.

[14] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 319–329, New York, NY, USA, 1988.

[15] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *Proceedings of the 2005 workshop on Memory system performance*, pages 36–43. ACM, 2005.

[16] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multilevel blocking. pages 346–357, 1997.

[17] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *ACM Sigplan Notices*, volume 42, pages 235–244. ACM, 2007.

[18] K. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):424–453, 1996.

[19] J. Meng, J. W. Sheaffer, and K. Skadron. Exploiting interthread temporal locality for chip multithreading. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[20] D. Orozco, E. Garcia, and G. Gao. Locality optimization of stencil applications using data dependency graphs. In *Languages and Compilers for Parallel Computing*, pages 77–91. Springer, 2011.

[21] G. Rivera and C.-W. Tseng. Tiling optimizations for 3d scientific computations. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 32–32, 2000.

[22] S. Shrestha, J. Manzano, A. Marquez, J. Feo, and G. R. Gao. Jagged tiling for intra-tile parallelism and fine-grain multithreading. In *Proceedings of the 27th International Workshop on Languages and Compilers for Parallel Computing, Hillsboro, OR, USA*, 2014.

[23] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 117–128, 2011.

[24] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 30–44, New York, NY, USA, 1991.

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