# Jagged Tiling for Intra-tile Parallelism and Fine-Grain Multithreading

Sunil Shrestha[1], Joseph Manzano[2], Andres Marquez[2], John Feo[2], and Guang R. Gao[1]

[1] CAPSL, University of Delaware, Newark, DE 19716
sunil@udel.edu,ggao@capsl.udel.edu
[2] Pacific Northwest National Laboratory, Richland, WA 99354
joseph.manzano@pnnl.gov,andres.marquez@pnnl.gov,john.feo@pnnl.gov

**Abstract.** In this paper, we have developed a novel methodology that takes into consideration multithreaded many-core designs to better utilize memory/processing resources and improve memory residence on tileable applications. It takes advantage of polyhedral analysis and transformation in the form of PLUTO[6], combined with a highly optimized fine grain tile runtime to exploit parallelism at all levels. The main contributions of this paper include the introduction of multi-hierarchical tiling techniques that increases intra tile parallelism; and a data-flow inspired runtime library that allows the expression of parallel tiles with an efficient synchronization registry. Our current implementation shows performance improvements on an Intel Xeon Phi board up to 32.25% against instances produced by state-of-the-art compiler frameworks for selected stencil applications.

## 1 Introduction

With the increasing number of cores in current computing systems and the massive computational power they offer, one of the bottlenecks in achieving higher performance has been the access to the memory. Along with the computation power, memory speed has increased as well, however at a much slower pace. Memory access latency is determined by many factors such as bandwidth, interconnect delay, memory bank contention, memory paging overhead and unbalanced task distribution. Taking advantage of the locality principle, multiple efforts [13], have been made so far to minimize the access time to memory in an integrated framework, such as by storing recently used data in cache, efficient reuse of cached data through tiling, data percolation using communication-avoiding algorithms [9], code transformations and prefetching. Although, these approaches work very well in pushing the "memory wall" farther, data movements end up being performed in most cases for the benefit of a single thread or computational unit.

An important class of optimization approaches tackling the "memory wall" maximizes memory reuse inside the most computational expensive parts of an applications – commonly nested loops. One of the most successful techniques to

date involves the concept of a "tile". A tile is a sub-partition of a loop nests' iteration space into blocks with the purpose of increasing the data locality and reducing communication overhead across them. This approach is very effective as we can see in classical tiling approaches where tile sizes are based on cache sizes. In these techniques, a single thread effectively maximizes reuse from caches before heading back to the main memory. As effective as this approach is, there are few things that need consideration: Firstly, when processing resources are in abundance and optimization is highly coarse grained, this may lead to idle resources. One of the main reasons that this pathology has persisted is of the limited memory bandwidth assumption. However, this assumption ignores the possibility of memory reuse across processing units, especially when they share different levels of the memory hierarchy. Secondly, when each thread performs data movements for itself without a priori knowledge of concurrent thread execution and data movement pattern, they can interfere with each other to their performance's detriment. This could take the form of increased caches misses and a higher strain on the entire memory subsystem.

In addition, the trend of increasing the number of processing elements in a chip seems to be the norm for many years to come. In order to take advantage of additional processing resources, current and new software stacks need to provide concurrent units the ability to (re)act based on other threads execution and the application data movement pattern. In other words, these stacks need to update their machine models to reflect modern platforms. With the axiom that the base optimization technique used would be tiling, memory reuse needs to be considered in both the inter, as well as, the intra tile so that the strain or underuse of resources can be alleviated. For the simple for-loop in Figure 1, a classical approach produces tiling as shown in Figure 2. This is a very efficient way of tiling as it is designed for minimal communication and coarse-grained parallelism. However, given the enormous amount of processing power, can we do better and utilize available resources while keeping communication overhead relatively low? In this paper, we attempted to answer this question with an approach that maximizes parallelism and improve locality for threads working together in a highly synchronous fashion.

```
for (int i=1; i<=n; i++){
    for (int j=1; j<=n; j++){
        A[i][j] = A[i-1][j] + A[i][j-1];
    }
}
```

**Fig. 1.** Stencil Example

Firstly, in order to maximize available processing resources while keeping memory bandwidth strain low, we have developed a technique to exploit intra-tile parallelism without compromising inter-tile parallelism. Using the existing

polyhedral framework PLUTO [6] and the code generator CLOOG [4], we create a multi-hierarchical highly parallel tiles that reside at the lowest level cache and communicate with minimal overhead. Sets of these inner tiles form outer level tiles that can also run in parallel. Secondly, in order to execute these highly parallel tiles, we have developed a data-flow inspired fine grained execution mechanism in which threads sync using atomic operations. Using locality provided by higher level caches (e.g L2 cache) and parallelism provided by our tiling technique, we are able to improve locality and reuse, thus improving the overall system performance.

The main contributions of this paper include multi-hierarchical tiling technique that increase intra tile parallelism; and a data-flow inspired runtime library that allows the expression of intra tile parallelism. The rest of the paper is organized as follows. Section 2 provides related work. Section 3 explains our framework in detail. Section 4 and 5 showcases experiments, results and discussion. Section 6 presents our future work. Finally, section 7 does the conclusion.

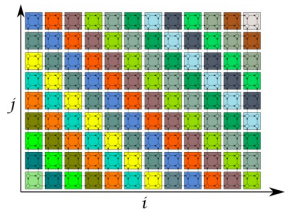



**Fig. 2.** Classical Tiling

## 2 Related Work

The technique of Iteration Space Tiling proposed by Michael Wolfe [27][26], has been widely used to aggregate different dimensions of loop iterations to improve locality and reuse [25]. It improves performance by exploiting reuse beyond innermost loops. Such approach works very well for perfectly nested loops, although it can be extended beyond. With the advancement made by the linear algebra community, hierarchical tiling code can be generated automatically using tools like PLUTO [6]. Using the polyhedral framework proposed by Feautrier [11][12], PLUTO attempts to generate communication minimal tiling code. The main idea here is to have efficient space and time mapping [15] while extracting multiple degree of parallelism that can be tiled with cost efficient hyperplanes.

One of the difficulty of tiling is to find right shapes and sizes that can map well to the iteration space and is also cost efficient [17]. Apart from classical rectangular or cubical shaped tiles, other shaped tiling shapes (e.g. diamond, hexagonal) has also been studied. These tiling shapes are selected based on the access patterns of the loop nest such that communication becomes minimal and/or allows parallel start for certain types of applications [16][2]. These techniques are very effective, however are normally limited to tiling for only one level of memory hierarchy.

Data centric approach used by Kodukula [20], uses blocking of data based on its flow through memory hierarchy . His approach selects sequence of blocks that

is touched by a processor and executes statements associated with those blocks. However, depending on dependence such transformation can be very complex.

Mitigating long latency cache miss penalties by overlapping data movement with computation is another avenue for memory latency optimization. In order to do so, memory access pattern needs to be known a priori to the actual use of data. However, prefetching has to be timed such that data stay in cache and do not get evicted before the use happens. Techniques such as loop splitting / loop peeling [22] to divide the loop in multiple phases where one can be used to prefetch data has been attempted before. However, loop peeling is not obvious for all nested loops. Prefetch when done unnecessarily, becomes just an extra overhead. To alleviate these shortcomings, systems like APACS [21] provide an integrated solution for cache and prefetching with adaptive partitioning, prefetch pipelining and prefetch buffer management techniques. Although such solutions are promising, they require substantial hardware support. Moreover, techniques like Intel helper threads [19] and Gan's data percolation [14] allows "special" threads to improve locality.

Bikshandi [5] introduced hierarchical tiled array (HTAs) that manipulates tiles using array operations such that parallel computation takes array form in distributed tiles. Baskaran [3] used explicit data movement and index transformation to improve locality on scratchpad. These techniques benefit block of threads when contiguous memory is used but they are less efficient when dealing with strided or irregular accesses.

Current fine grain runtime concentrate on the interaction of many threads and had dealt with memory bandwidth and contention problems in many interesting ways. The Efficient Architecture for Running THreads (EARTH) [23] uses the un-interruptable fine grain threads, called fibers, which shares their activation frames. The SWARM framework developed by ET International [18] has similar concepts as EARTH, but added support for direct access shared memory structures, placement information and different "codelet" pattern operations that allow more efficient use of data. Finally, the most recent development is Rice's Open Community Runtime (OCR) [7] system. It is designed as a fertile ground to experiment with new techniques and systems.

Most of the original compiler work presented here has a single thread focus or starkly favors coarse grain parallelism. Improving on this, a fine-grain collaborative view on code generation for multiple threads can yield significant improvements. This thread collaboration can extend beyond scheduling techniques and encompass data restructuring as well; a topic that was not considered in the fine-grain multithreaded work presented above. The approach presented in [10] aims to solve these issues with a hierarchical tiling framework for multicore clusters. They exploit inter and intra (with threading) node parallelism using tiling. Moreover, they exploit NUMA aware allocation and take advantage of vector register blocking to further increase performance. Stencil codes are used to showcase benefits. Although their approach is similar to our framework, key differences in our favor are the creation of more intra-tile parallelism (similar to their intra-node parallelism) with our jagged tiling technique.

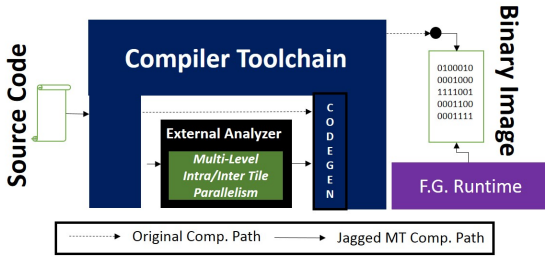



**Fig. 3.** Framework

## 3 Framework

The overview of the framework is shown in Figure 3. The framework is designed to maximize the use of parallel processing resources while keeping runtime overhead low. It uses PLUTO [6] and CLOOG [4] as a compiler toolchain for highly parallel tiled code generation as shown in Section 3.1. It is followed by data-flow like execution where threads working in close proximity in time and space form a thread group $ThGrp$ and collaborate within with minimal overhead. Such grouping is designed to take advantage of locality on L2 tiles while executing in highly parallel and synchronous fashion.

### 3.1 Overview of Polyhedral Code Generation

In this subsection, we provide a brief overview of PLUTO, CLOOG and some of the polyhedral terminologies used in this paper.

PLUTO takes a C code as an input and transforms it into a coarse-grain parallel OpenMP code that is optimized for data locality. Using its affine transformation framework, statement wise transformations are done to minimize communication across boundaries. A cost function is used to reduce the communication distance and volume between tiling hyperplanes.

Under polyhedral terminology, a hyperplane $\phi(\boldsymbol{v})$ is a $n-1$ dimensional affine subspace in $n$ dimensional space. For statements $S$, a hyperplane $\phi_s(\boldsymbol{v})$ with dimensionality $m$ and normal $(c_1 c_2 ... c_m)$ represents an affine transformation in the form,

$$\phi_s(\boldsymbol{v}) = (c_1 c_2 ... c_m).\boldsymbol{v} + c_0 \tag{1}$$

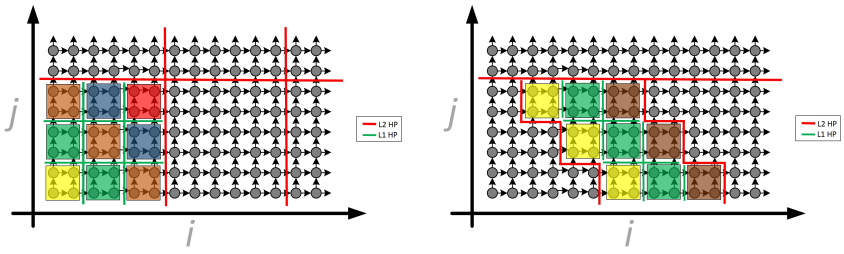
For given '$k$' statements, in order for statement-wise hyperplane $(\phi_{s_1}, \phi_{s_2} ... \phi_{s_k})$ to be a legal tiling hyperplane, the following has to be satisfied between source '$s$' and target '$t$' along all dependence edge.

$$\phi_{S_i}(\boldsymbol{t}) - \phi_{S_j}(\boldsymbol{s}) \geq 0 \tag{2}$$

When a combination of '$m$'[3] 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.

PLUTO finds communication minimal tiling hyperplanes and uses them for multiple level tiling targeting different levels of the memory hierarchy. Such tiling creates supernodes for each different tiling levels. It uses CLOOG, which is a code generation tool that scans the polyhedra in a global lexicographic ordering. Such scanning is performed as specified by the scattering function, which is an affine transformation function. The code generator is oblivious to any information about the dependencies and, in absence of a scattering function, scans the polyhedra in the lexicographic order as specified by the original iterator. It uses PolyLib [24] for its polyhedral operations. Interested readers are strongly encouraged to read references [6] and [4] for more information about PLUTO and CLOOG respectively.

Our framework leverages the existing transformation framework, parallelization and locality optimization that PLUTO uses for code generation. In order to generate parallel inner tiles, our framework uses an external analyzer that calls PLUTO using a modified *Domain* (iteration space polytope) and a updated *Scattering function* (scanning order of polyhedra). We explain our algorithm in the next section.




(a) Two level hierarchical Tiling with (1,0) and (0,1) Tiling Hyperplanes

(b) Two level hierarchical Tiling with (1,1) and (0,1) Tiling Hyperplanes

**Fig. 4.** Two level hierarchical Tiling: Classical and Jagged

### 3.2 Jagged Tiling for Intra-tile Parallelism

Our framework uses the tiling hyperplanes generated by PLUTO for the lowest level memory hierarchy i.e. (L1 cache). These tiling hyperplanes are designed to be communication minimal using a cost function that reduces the communication distance and volume. Under the PLUTO framework, these hyperplanes

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

are used for tiling multiple level of memory hierarchy. However, using the same tiling hyperplanes for both levels come at the cost of sequential or pipeline parallel execution of inner L1 tiles as shown in figure 4(a). In our framework, we solve this by constructing outer tiles in which at least one face of the polytope has concurrent start. Given '$m$' hyperplanes, we create at least one parallel hyperplane and use it together with the other hyperplanes to create the L2 tiles. For clarity, we represent all original hyperplanes by $\phi$, L1 hyperplanes by $\varphi$ and L2 hyperplanes by $\Phi$. The condition for such tiling is shown in equations below,

$$\varphi^1_{S_i}(\boldsymbol{t}) - \varphi^1_{S_j}(\boldsymbol{s}) \geq 1 \quad for\ one\ hyperplane \tag{3}$$

$$\varphi^l_{S_i}(\boldsymbol{t}) - \varphi^l_{S_j}(\boldsymbol{s}) \geq 0 \quad for\ l\ hyperplanes\ where\ 1 < l \leq m \tag{4}$$

The algorithm to generate such tiles is shown in algorithm 1. Once the level 1 tiles are created, we use the $L1$ supernode hyperplanes $\varphi i_{L1s}, \varphi^{i+1}_{L1s} ... \varphi^{i+m-1}_{L1s}$ to create the outer L2 tiles. The iteration space matrix representing the *Domain* is updated to reflect the new L2 tile domain and its scattering function is updated to mark outer and inner tiles parallel. Figure 4(b) shows the pictorial view of jagged tiling.

For our stencil example, the tiling hyperplanes are (1,0) and (0,1). Using these hyperplanes, we create L1 tiles and iterators '$i$' and '$j$' with tile size of 32. Once the supernode iterator '$I$' and '$J$' are created, hyperplanes (1,0) and (0,1) are added and thus hyperplanes (1,1)[4] and (0,1) are used to create L2 tiles using the tile size of 8 (which gives L2 size 8*32). The resulting scattering functions are used to create parallel code. The *Domain* and *Scattering functions* produced by this process are shown below[5] for the example stencil in Figure 1.

| *Domain* | *Scattering* |
|---:|---:|
| $1 \leq i \leq n-1$ | $c1_{L2} = I_{L2} + J_{L2}$ |
| $1 \leq j \leq n-1$ | $c2_{L2} = J_{L2}$ |
| $32I_{L1} \leq i \leq 32I_{L1} + 31$ | $c1_{L1} = I_{L1} + J_{L1}$ |
| $32J_{L1} \leq j \leq 32J_{L1} + 31$ | $c2_{L1} = J_{L1}$ |
| $8I_{L2} \leq I_{L1} + J_{L1} \leq 8I_{L2} + 7$ | $c1 = i$ |
| $8J_{L2} \leq J_{L1} \leq 8J_{L2} + 7$ | $c2 = j$ |

### 3.3 Fine-Grain Execution

Codes that are designed to run at a very fine grain level suffer from communication overhead, reflected in its performance. With jagged tiling we have created a

[4] the parallel hyperplane

[5] where $n$ is the size of a dimension in the iteration space. For our example, both dimensions are the same

**Algorithm 1** Generating Jagged Tiles

**Input:** Given tiling hyperplanes $\phi_s^i, \phi_s^{i+1}...\phi_s^{i+m-1}$, Domain $D_s$, L1 tile sizes $tL1_i, tL1_{i+1}...tL1_{i+m-1}$ and L2 tile sizes $tL2_i, tL2_{i+1}...tL2_{i+m-1}$

1: $TS_1 = \{tL1_i, tL1_{i+1}...tL1_{i+m-1}\}$ ▷ A set of all L1 Tile sizes
2: $TS_2 = \{tL2_i, tL2_{i+1}...tL2_{i+m-1}\}$ ▷ A set of all L2 Tile sizes
3: $\phi = \{\phi_s^i, \phi_s^{i+1}...\phi_s^{i+m-1}\}$ ▷ A set of the original tiling hyperplanes
4: PLUTO_TILING_ALGORITHM($\phi$, $D_s$, $TS_1$) ▷ At this point, all $\varphi_{L1s(s)}$ are created
5: Update *Domain* constraint to get parallel hyperplane $\varphi_{L1s}^1 \rightarrow \varphi_{L1s}^1 + \varphi_{L1s}^2$ such that $\varphi_{L1s(t)}^1 - \varphi_{L1s(s)}^1 \geq 1$ leaving other hyperplanes as is
6: $\varphi = \{\varphi_{L1s}^i, \varphi_{L1s}^{i+1}...\varphi_{L1s}^{i+m-1}\}$
7: PLUTO_TILING_ALGORITHM($\varphi$, $D_s$, $TS_2$) ▷ At this point, all $\Phi_{L2s(s)}$ are created
8: Perform Unimodular transformation on L1 scattering supernode: $\varphi T_{L1s}^1 \rightarrow \varphi T_{L1s}^1 + \varphi T_{L1s}^2$ to extract inner parallelism.
9: Perform Unimodular transformation on L2 scattering supernodes: $\Phi T_{L2s}^1 \rightarrow \Phi T_{L2s}^1 + \Phi T_{L2s}^2$ to extract outer parallelism.

**Output:** Updated domain and scattering function

highly parallel code that is capable of running multiple level of tiles in parallel. In order to exploit available parallelism, we use a data-flow inspired low overhead dependency and task update scheme, based on bit masking, that enables multiple threads to work synchronously within a tile. Figure 5 shows the overview of our fine grain execution approach. These mechanisms allows high performance, low overhead communication within thread groups during the program execution.

Each group of threads grab a L2 task and initial dependency mask associated with it. Such masks are repeatable across different L2 tiles for regular applications. Dependencies among the lowest level tiles are represented by different sets of bits which are collectively updated by a group of threads working together inside the highest level tile. Each thread perform atomic bit-wise operations on the dependency masks and creates a task mask that represents all the tasks ready to execute. If the task mask is non-zero, thread finds the task, execute, update dependencies and update the task. This happens in a highly parallel fashion such that every thread is aware of the status of the tasks within the assigned tile. The implementation of this approach of synchronization between threads is done solely using atomic operations to keep the overhead low. This process is shown in Figure 5, where the initial bit-mask is updated during the execution until all tasks are completed.

Our goal is to exploit the parallelism for a given architecture. With such an approach, threads can work in a collaborative fashion while reducing contention and hence improving overall performance.

## 4 Experimental Setup

The experiments were done on Intel Xeon Phi 7110P coprocessor. Each coprocessor is equipped with 61 cores running at 1.1 GHz connected with FDR infiniband interconnect. Each core can support up to 4 hyper-threads, totaling up to 244

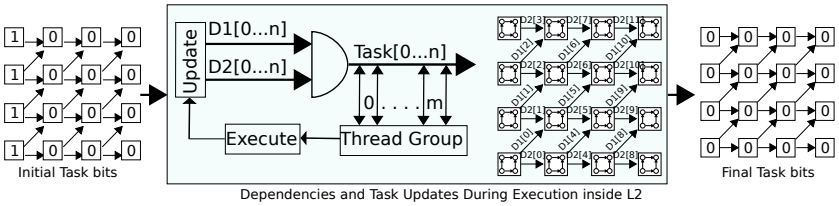
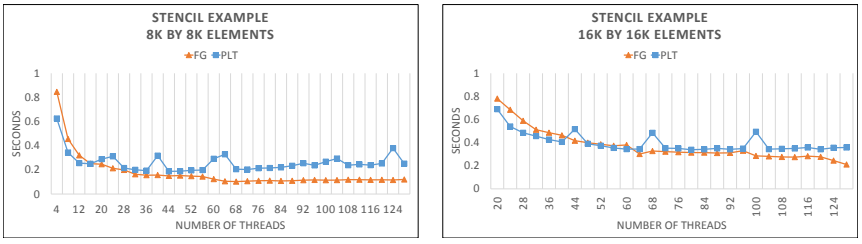



**Fig. 5.** Fine Grain Execution Example

threads. Each core has 32KB L1 cache per thread and 512KB L2 cache shared by 4 threads. In addition to the private L2, cores in this system also have access to L2s of all other cores via a ring topology. Only when there is a private L2 cache miss as well as ring L2 caches miss (shared L2s), the request is served by the memory.

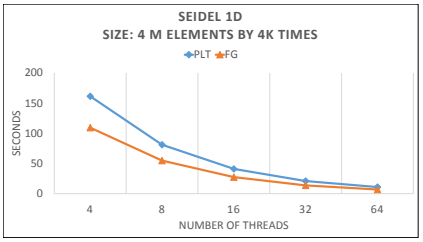



(a) Stencil 8k by 8k Execution time (b) Stencil 16k by 16k Execution time

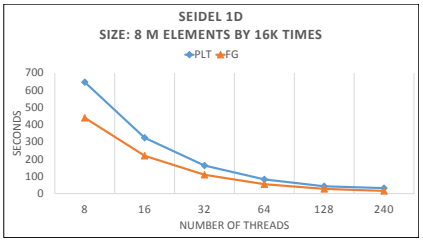
**Fig. 6.** Stencil Example Execution times for PLUTO generated code (PLT) and our framework (FG)

On the software side, we divided our experiments into three different sections. In the first section, we selected an example stencil to explore the behavior of parallelism at fine granularity. We ran the example stencil with sweeps from 4 to 128 threads with 4 threads per group with two workloads: 8k by 8k and 16k by 16k elements, as shown in Figure 6.

In addition, we implemented two versions (1D and 2D) of the Seidel solver loop, a well known scientific algorithm that computes the solution of a set of linear equations. We selected these examples to showcase our framework against PLUTO generated code using OpenMP as their parallel target. We selected arrays of 4 million elements and 8 millions elements running 4k and 16k times respectively for the one dimensional Seidel. The execution time for these runs are presented in Figure 7. For our 2D Seidel, our selected workloads are a 4k by 4k array of elements ran over 2K times and a 10k by 10K element array ran over 6k times. The results of these runs are shown in Figure 8

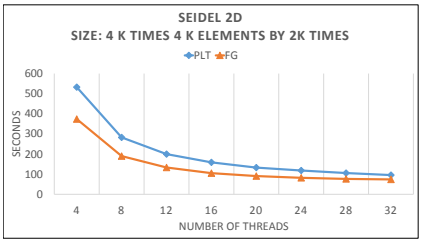



(a) Seidel 1D 4 Million elements Execution time

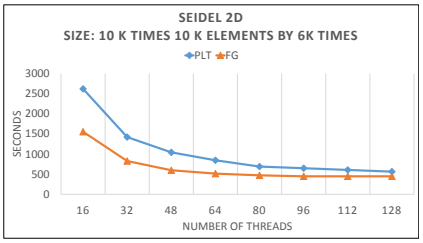



(b) Seidel 1D 8 Million elements Execution time

**Fig. 7.** Seidel 1D Execution times for PLUTO generated code (PLT) and our framework (FG)

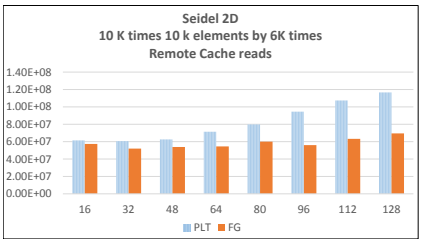



(a) Seidel 2D 4 thousand by 4 thousand elements Execution time

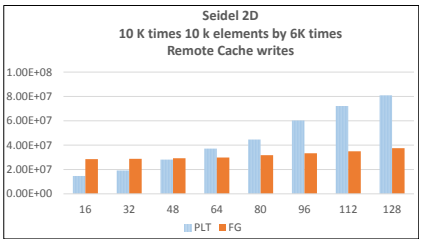



(b) Seidel 2D 10 thousand by 10 thousand elements Execution time

**Fig. 8.** Seidel 2D Execution times for PLUTO generated code (PLT) and our framework (FG)



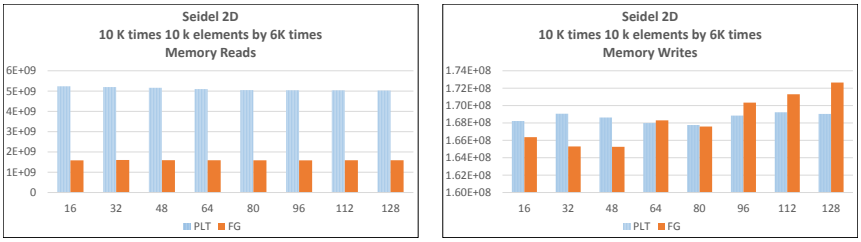

(a) Seidel 2D Remote Cache misses for Reads




(b) Seidel 2D Remote Cache misses for Writes

**Fig. 9.** Seidel 2D Remote Cache misses for PLUTO generated code (PLT) and our framework (FG)

In the final section, we chose one Seidel example, the biggest 2D case, to characterize the memory and remote cache misses. These results are shown in Figure 9 for caches and in Figure 10 for memory.




(a) Seidel 2D Memory-served misses for Reads

(b) Seidel 2D Memory-served misses for Writes

**Fig. 10.** Seidel 2D Memory-served misses for PLUTO generated code (PLT) and our framework (FG)

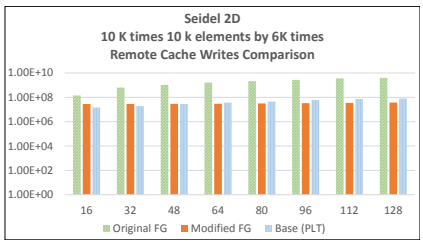



**Fig. 11.** Seidel 2D write serviced by memory for PLUTO generated code (PLT) our framework plus overhead and our framework without overhead. Note that the Y axis is in logarithmic scale.

All our experiments were designed to utilize all 4 hyperthreads provided by our target architecute. In order to do so, we did 'compact pinning' such that hyperthreads form a thread group in our fine-grain execution. Similary, we set 'KMP_AFFINITY' to compact for OpenMP code. Our experiments show better runtime for fine grain execution compared to PLUTO generated parallel code using OpenMP as a baseline – denoted as 'PLT' in figures. In addition, our results shows that when threads collaborate, we gain in performance. All our codes were compiled using Intel's icc version 13.1.1 and use the Linux Perf tool [1] to collect the memory and cache related performance counter information. The performance counters collected from our experiments are presented in Table 1.

## 5 Discussion

The benefit of execution with locality consideration comes from the amount of reuse an application offers.The stencil in example in Figure 1 does not have much reuse. Thus, when smaller number of threads are used, the generated PLUTO code (represented as PLT in the charts) can outperform fine-grain execution with jagged tiling (represented as FG in the charts) since only limited amount

| Performance Counter | Description |
|---|---|
| L2_DATA_READ_MISS_CACHE_FILL | Level 2 Cache misses for reads serviced by a remote cache |
| L2_DATA_WRITE_MISS_CACHE_FILL | Level 2 Cache misses for writes serviced by a remote cache |
| L2_DATA_READ_MISS_MEM_FILL | Level 2 Cache misses for reads serviced by the memory |
| L2_DATA_WRITE_MISS_MEM_FILL | Level 2 Cache misses for writes serviced by the memory |

**Table 1.** Performance Counters collected for the Largest Seidel Example

of data is reused in few cores L2 caches and it is not enough to overcome the slow start introduced by coarser outer L2 tiles. However, as number of threads are incremented, FG show better execution time. This trend is visible in Figure 6.

Figures 7 and 8, on the other hand, have much better reuse since the Seidel loops (for both 1D and 2D) have time dimensions in which the entire arrays are reused. When L2 locality is not considered as is the case with the PLUTO generated code, eviction rates increases and this is reflected in the performance of this approach. However with FG, threads working as a group execute different tiles within L2 in a synchronous fashion exploiting both parallelism and reuse offered within outer tiles. In all these approaches, the performance gains are clearly visible up to a number of threads and afterward the performance stay constant. This plateau is reached when the available parallelism is exhausted and the rest of the threads do not have any useful work. In Figures 7(a) and 8(a), the plateau is reached at 64 and 32 threads respectively. When increasing the workload sizes (as in Figures 7(b) and 8(b)), the plateau is pushed further (to 240 threads for the first case and 128 for the second).

In Intel Xeon Phi, when a thread misses a read or a write access to local L2 caches, it first checks if data is available in neighboring L2 caches and goes to the memory only if there is both local and remote L2 misses. These events include both demand fills as well as prefetches and is hence a close approximation for demands hits and misses in local L2 caches. In the Intel Xeon Phi, the remote cache accesses might be as expensive as an access to memory; thus, having a large number of remote cache access might greatly affect performance [8]. Figures 9(a) and 9(b) show amount of reads and write shared among caches within the L2 ring for the largest Seidel loop example (c.f. Figure 8(b)). Similarly, Figures 10(a) and 10(b) show amount of data brought from memory for the same test case.

The information extracted from Figures 9(a) and 10(a) show that the PLUTO generated code has a higher number of remote cache accesses and memory serviced reads compared with the Fine grained approach. This tells us that for the same workload, fine grained data is reused more often than its PLUTO coun-

terpart (i.e. low memory misses and low remote cache misses for the same data set).

In the case of writes, Figures 9(b), 11 and 10(b), show interesting results. The remote cache misses for the writes shows an inflection point around 64 threads. After 64 threads, the FG approach shows improvements over the PLUTO generated code. However, before the inflection point, our approach incurs in more share writes. The reason is that when using more cores more core caches can participate into the computation.

Figure 11 show the original overhead for memory writes for our framework. The Y axis in this figure is in log-scale to better show the differences between the approaches. The original framework suffers from a large number of writes to memory. This effect is due to a synchronization variable inside the framework. However, the number of writes in this application are an order magnitude lower than reads. Thus, their effect in performance is small. However, to obtain the application's memory-serviced writes, we modified the framework to bypass the synchronization variable. This data is presented in Figure 10(b). In this case, we have an increase on writes, but as expressed before, the sheer number of more reads than writes, means that reads have a larger influence in the performance of the application for this application. However, we are investigating how to reduce this write pressure on the main memory.

These charts show that for application with plenty of reuse, threads can take advantage of accesses by threads within group and hence collaborate to maximize memory residence in nearby memory with minimal interference.

The balance between granularity of tasks and the amount of parallelism is very crucial to maximize performance. Coarse grain execution can lead to underutilization of resources whereas fine grain can lead to more conflicts and contentions at different levels of memory hierarchy. Most of current architectures offer vast amount of computing resources, some with hyperthreads sharing caches and some with non-uniform memory access where accessing some address ranges are cheaper than others. In such cases, taking advantage of shared resources such as caches or address range in close proximity by threads working together as a unit can have significant effect on performance.

## 6 Future Work

With our jagged tiling technique and fine grain execution, we improve both locality and parallelism of an application. However, current implementation of our jagged tile creation is limited to pipeline parallel applications. We plan to extend our apporoach to include highly parallel algorithms that has parallel start such that both inner and outer tile parallelism can be exploited. Also, the concept of grouping of thread to improve locality with thread collaboration can benefit many other applications besides stencils. We plan to look into different applications with reuse and architectures that can take advantage of thread and memory mapping. In addition, we plan to reduce synchronization overhead that our framework currently incurs to make our framework more efficient.

## 7 Conclusion

Today's systems poses tremendous computational power, however memory subsystems haven't been able to keep up with the accelerating pace of computing resources. Many optimization techniques over the years have provided significant boosts in performance by reducing memory access latency. These techniques, although very effective, often stay blindfolded towards collaboration chances that processing resources in massively parallel systems present. This can lead to resource underutilization and missed opportunities to maximize reuse among threads working in close proximity in time and space. With our novel tiling approach that exploits multi-level parallelism along with our fine grain execution framework, we showed that when parallel threads collaborate, it leads to higher cache reuse and better resource utilization.

## References


1. perf: Linux profiling with performance counters.
2. Vinayaka Bandishti, Irshad Pananilath, and Uday 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.
3. Muthu Manikandan Baskaran et al. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 1–10. ACM, 2008.
4. Cédric Bastoul. Generating loops for scanning polyhedra: Cloog users guide. *Polyhedron*, 2:10, 2004.
5. Ganesh Bikshandi et al. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 48–57, New York, NY, USA, 2006. ACM.
6. Uday Bondhugula and J Ramanujam. Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer. 2007.
7. Intel Open Source Technology Center. Open community runtime, 2012.
8. Shannon Cepeda. Optimization and performance tuning for intel xeon phi coprocessors, part 2: Understanding and using hardware events, 2012.
9. Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors. *Siam Review*, December 2008.
10. Hikmet 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. Paul Feautrier. Some efficient solutions to the affine scheduling problem. i. one-dimensional time. *International journal of parallel programming*, 21(5):313–347, 1992.
12. Paul Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

13. Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–, Washington, DC, USA, 1999. IEEE Computer Society.
14. G. Gan, X. Wang, J. Manzano, and G. Gao. Tile percolation: an openmp tile aware parallelization technique for the cyclops-64 multicore processor. *Euro-Par 2009 Parallel Processing*, pages 839–850, 2009.
15. Martin Griebl, Christian Lengauer, and Sabine Wetzel. Code generation in the polytope model. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 106–111. IEEE, 1998.
16. Tobias Grosser, Sven Verdoolaege, Albert Cohen, and P Sadayappan. The relation between diamond tiling and hexagonal tiling. *HiStencils 2014*, page 65, 2014.
17. Karin Högstedt, Larry Carter, and Jeanne 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.
18. ET International. Swarm (swift adaptive runtime machine), 2012.
19. Dongkeun Kim et al. Physical experimentation with prefetching helper threads on intel's hyper-threaded processors. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 27–, Washington, DC, USA, 2004. IEEE Computer Society.
20. Induprakas Kodukula, Nawaaz Ahmed, and d Keshav Pingali. Data-centric multilevel blocking. pages 346–357, 1997.
21. J. Lewis et al. An automatic prefetching and caching system. In *Performance Computing and Communications Conference (IPCCC), 2010 IEEE 29th International*, pages 180–187, Dec 2010.
22. Massachusetts Institute of Technology. Laboratory for Computer Science and D.O.J. Tanguay. *Compile-time Loop Splitting for Distributed Memory Multiprocessors*. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1993.
23. K.B. Theobald. *Earth: An Efficient Architecture for Running Threads*. McGill University, 1999.
24. Doran K. Wilde. A library for doing polyhedral operations. Technical report, 1997.
25. Michael E. Wolf and Monica 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. ACM.
26. 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. ACM.
27. Michael Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics.