Load Balancing with Polygonal Partitions

Aniket Shivam
Department of Computer Science
University of California, Irvine, USA
aniketsh@uci.edu

Priyanka Ravi
Department of Computer Science
University of California, Irvine, USA
priyanr1@uci.edu

Alexander V. Veidenbaum
Department of Computer Science
University of California, Irvine, USA
alexv@ics.uci.edu

Alexandru Nicolau
Department of Computer Science
University of California, Irvine, USA
nicolau@ics.uci.edu

Rosario Cammarota
Qualcomm Research, San Diego, USA
ro.c@qti.qualcomm.com

Abstract

In this work we propose a new dynamic scheduling technique for DOALL loops with non-uniform reuse patterns.

Dynamic scheduling of a DOALL loop with non-uniform reuse patterns partitions the loop into chunks of iterations that exhibit poor locality at runtime - dynamic scheduling algorithms do not factor in locality. This results in suboptimal completion time even if the load imbalance is bounded by the chosen scheduling algorithm, e.g., guided self-scheduling. Partitioning a DOALL loop for maximizing locality produces chunks of sizes that depend on the reuse patterns in the loop and do not account for load imbalance at runtime. Hence, even though locality in each chunk is maximized, the performance achieved during parallel execution is suboptimal.

The proposed technique starts with partitioning a DOALL loop with a polyhedral framework to create partitions that maximize locality, and then *re-tiles* these partitions to achieve load balance at runtime. The proposed re-tiling and scheduling technique shows a performance speedup up to 2x against PLuTo.

Keywords Polygonal Partitions, Shape and Size Independent Tiling, Workload Balancing

1 Introduction

Dynamic scheduling techniques for DOALL loops aim to allocate chunks of iterations to parallel computing cores in ways to minimize load imbalance[9, 12]. Dynamic scheduling techniques can be classified as follows: (a) *forward looking*; (b) *profile-based*; (c) *adaptive*. In *forward looking* techniques [6, 9, 12], given the number of iterations and the number of computing cores the sequence of chunks can be allocated ahead of the loop execution. For example, given a number of iterations $N = 1000$ and a number of parallel cores $P = 4$, then the guided self-scheduling[12] will generate the following set of 21 chunks: $C = \{c_1 = 250, c_2 = 188, c_3 = 141, c_4 = 106, c_5 = 79, c_6 = 59, c_7 = 45, c_8 = 33, c_9 = 25, c_{10} = 19, c_{11} = 14, c_{12} = 11, c_{13} = 8, c_{14} = 6, c_{15} = 4, c_{16} = 3, c_{17} = 3, c_{18} = 2, c_{19} = 1, c_{19} = 1, c_{20} = 1, c_{21} = 1\}$, in which c_1 includes iteration 0 to 249, c_2 includes iteration from 250 to 437 etc. in the lexicographic order. Techniques such as guided-self scheduling guarantee tight bounds on load imbalance, e.g., within one minimum chunk of iterations (1 in the example above), when the standard deviation on the average iteration execution time is constant. *Profile-based* techniques, e.g., [8] perform a step of profiling to partition the loop in a way to cope with variability in the execution time due to caching effects and irregular shape of the iteration space, e.g., triangular iteration spaces in matrix transposition and lower-upper decomposition - even in this case, iterations within a chunk execute in the lexicographic order. Finally, *adaptive* techniques profile chunks completion time at run-time to establish the size of the next chunk to form and schedule [1, 10], or it finds a more suitable set of static chunks across DOALL loop executions [3, 4].

Dynamic scheduling techniques do not factor in information about locality. In DOALL loops with uniform reuse patterns, loop tiling finds the size of tiles with uniform shape - e.g., rectangles or parallelograms, to improve locality. Then chunks of tiles can scheduled according to a dynamic scheduling algorithm to bound load imbalance at run-time and maximize locality. However, in DOALL loops with non-uniform reuse distance, traditional loop tiling does not help with improving locality, and other techniques that partition the loop by following the reuse patterns in the loop need to be applied to ensure that locality is maximized in each tile [11, 14].

In this work we propose a new dynamic scheduling technique for DOALL loops with non-uniform reuse patterns. Techniques to partitions DOALL loops with non-uniform reuse patterns use the polyhedral framework [11, 14] to reorder the iterations and arrange them in chunks with maximum locality. However, the shape of such chunks is non-uniform and unique to the DOALL loop - as the partitioning algorithm follows the reuse pattern in the loop. Hence, the sizes and shape of the polygons do not conform with the

IMPACT 2018

January 23, 2018, Manchester, United Kingdom
In conjunction with HiPEAC 2018.
http://impact.gforge.inria.fr/impact2018

schedule of known scheduling algorithms, e.g., guided self-scheduling. Hence, dynamic scheduling of such polygons produces load imbalance during parallel execution.

Shape and size of the the polygonal partitions are created for maximizing data locality by capturing data reuse patterns, which vary across the entire iteration space, and with the size of the iteration space. For example, there exist partitions with iterations with no data reuse, other partitions with iterations with data reuse with one or more than one iterations. The application of a dynamic scheduling algorithm, which is agnostic of the properties of the polygonal partitions, ultimately results in suboptimal parallel execution time as the size of the partitions (chunks, in the terminology of dynamic scheduling), as locality is compromised. Furthermore, since the partition sizes do not adhere to the schedule of any of the dynamic scheduling algorithms, e.g., [6, 9, 12], executing each partition on specific thread creates massive workload imbalance.

In this work, we propose *re-tiling*, a technique for scheduling non-uniform partitions such that parallel execution load imbalance is bounded and each chunk of execution preserves a certain amount of locality. The proposed technique provides scalability for processors with many cores. The heuristics for the scheduling of these re-tiled partitions is based on the careful analysis of the sizes and scaling factors of each type of the generated polygonal tiles. The proposed technique is implemented using the integration of source-to-source optimizer PLuTo[2] with Polylib library[1].

The remainder of the paper is organized as follows. Section 2 describes the approach to define the optimal scheduling strategy. Section 3 discusses the experiments and their results. Section 4 presents and comments on prior and related work. Finally, Section 5 summarizes the benefits of the technique.

2 Orchestrating Polygonal Partitions

In this section, we first give an overview of the loop transformations to generate polygonal partitions. Then, we justify the need for workload balancing when executing these polygonal tiles across multiple threads. Lastly, we explain the technique and the scheduling strategy to balance workload, which also allows the parallel execution to scale with the workload and the number of cores.

2.1 Polygonal Partitioning Technique

The technique for generating polygonal partitioning of iteration space[14] can be summarized as follows:

Each instance of a statement enclosed in a loop-nest may be defined by an iteration vector $\boldsymbol{I}$ for the multi-dimensional iteration space. If the enclosed statement accesses the data in a multi-dimensional array $\mathbf{A}$, the exact location of the data ($A(\boldsymbol{I})$) can be calculated as: $A(\boldsymbol{I}) = \mathbf{R} \times \boldsymbol{I} + \mathbf{r}$. R (*reference matrix*) is based on the coefficient of the iteration variables in the subscript representing the data access in A. Whereas, r (*offset vector*) represents the constant from the subscript. For a D-dimensional array A, with N being the depth of the loop-nest, R will be a $D \times N$ matrix and r will be a D-dimensional vector identifying an offset in each dimension. For example, consider a loop-nest of depth 2 with a single statement $X[i, j] = Y[i, i + j + 3] + Y[i + j, j]$. For reference Y[i,i+j+3], R will be a 2×2 matrix $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$. Each row of R represents the projection of the reference along each dimension of the array, i.e., the value of subscript in each dimension (i and i+j+3). The column represents the coefficient associated with each iteration variable (i and j) of the loop-nest. The offset vector r is a column vector, $\begin{pmatrix} 0 \\ 3 \end{pmatrix}$, representing the offset for reference along every dimension, i.e., the constants in the subscript. An iteration $\boldsymbol{I}$ can be substituted using a column vector (i j). Therefore, each reference to the array is an unique combination of (R, r). The pair can be represented as Γ to compute the *image* of a polyhedron and locate the accessed data point by an iteration using $\Gamma = R \times \boldsymbol{I} + r$. Using two different functions Γ_α for $Y[i, i + j + 3]$ and Γ_β for $Y[i + j, j]$, a *temporal reuse* relation $\mathcal{T}$ can be formulated such that substituting a particular iteration (I_α) in the relation yields another iteration (I_β) that reuses the same data. In the above statement, iteration (2,1) using Γ_α have reuse with iteration (-4,6) using Γ_β.

For a pair of references represented by Γ_α and Γ_β to an array in a statement in the loop-nest, the primary step is to partition the iteration space ($\mathcal{D}$) into four sets denoted by $\mathcal{L}$, $\mathcal{D}_1$, $\mathcal{D}_2$ and C.

- $\mathcal{D}_1$ iterations reference the data using Γ_α that is only accessed by Γ_β of another iteration.
- $\mathcal{D}_2$ iterations reference the data using Γ_β that is only accessed by Γ_α of another iteration.
- C contains iterations that reference data using Γ_α and Γ_β which are referenced in other iterations too.
- The rest of the iterations in $\mathcal{D}$ i.e. the iterations that reference the data which is not referenced by another iteration are denoted by L. Hence, $\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2 \cup C \cup \mathcal{L}$.

Partitions ($\mathcal{D}C_k$ and C_k) of iterations are generated to maximize the temporal locality based on the reuse pattern, as shown in figure 1. The partitioning algorithm generates the partitions at the kth step as follows:

- $\mathcal{D}C_k$ partitions: $\mathcal{D}_1$ iterations that link to the chain of $k - 1$ C iterations and at the end link to a $\mathcal{D}_2$ iteration by relation $\mathcal{T}_k$.
- C_k partitions: The remaining C iterations that are linked to themselves by $\mathcal{T}_k$ forming a cyclic-link of kC iterations.

[1]Polylib - A library of polyhedral functions, http://www.irisa.fr/polylib/

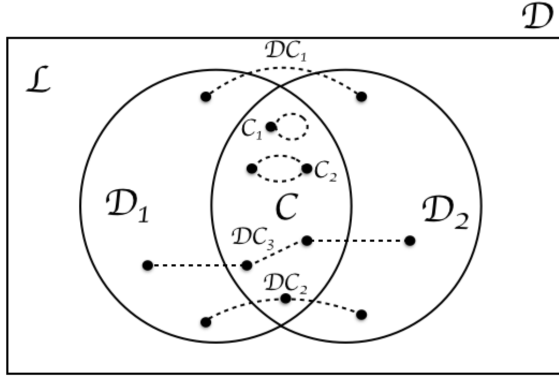


Figure 1. Set Representation and Classification of iterations for $\mathcal{D}C_1$, C_1, $\mathcal{D}C_2$, C_2 and $\mathcal{D}C_3$.

However, the algorithm needs to be halted at an optimal point, which has been described as follows in prior art [14]:

- If after the kth repetition, the entire iteration space ($\mathcal{D}$) is completely partitioned.
- If the value of k_{max} is too high, then it is critical to find an optimal value of k to protect gained speedup from increasing control statement overhead for managing the partitions. From the experimental results, it was determined that the algorithm must be halted if the number of iterations in generated partitions is below 625.

The application of the algorithm is extended to the Multi-Reference statements using following rules:

1. Eliminate pair of references that link iterations to themselves by either $\mathcal{T}$ or $\mathcal{T}^2$.
2. Eliminate pair of references that create a single partition for the entire iteration space.
3. Select the best-pair based on the amount of reuse in the partitions using the reuse count function:

$$Reuse(\Gamma_\alpha,\ \Gamma_\beta) = \sum_{i=1}^{k} i \times |\mathcal{D}C_i^0| + \sum_{i=1}^{k} i \times |C_i^0| \qquad (1)$$

Therefore, to exploit reuse while distributing workload across multiple threads, iterations belonging to a certain type of partition (either a $\mathcal{D}C_k$ type or a C_k type) and accessing the same data need to be executed on the same thread. $\mathcal{L}$ type of partition consists of independent iterations with no such reuse of data, hence its iterations can be either distributed equally across dispensable threads or the iterations can be executed together, considering the $\mathcal{L}$ as a partition.

2.2 Determining the Size and Scaling factor of a Partition

In case study 1 - two dimensional irregular reuse pattern (Listing 1), later analyzed in section 3.1: On scaling the dataset,

```
for (i = -N; i <= N; i++) {
  for (j = -N; j <= N; j++) {
    X[i][j] = Y[i][i+j+3] + Y[i+j][j];
  }
}
```

Listing 1. Loop-Nest with 2-D Non-Uniform Reuse

the number of iterations in partition C_6 increases at the same rate as the dataset, whereas $\mathcal{D}C_1$ and $\mathcal{D}C_4$ grow at half the rate of the dataset (Table 1). For example, on increasing N from 1024 to 2048, the iteration space/dataset is increased 4 times, since the loop bounds are -N to N for both inner and outer loop. C_6 will grow 4 times (same rate as the dataset) and $\mathcal{D}C_4$ will grow 2 times (half the rate of the dataset). The size of $\mathcal{D}C_3$ remains constant for any dataset. The size of L partitions does not affect the process the workload balancing since these iterations have no data reuse and can be computed independently of any other iteration in the space. Table 2 shows the scale of irregularity in the iteration count per partition, which eventually causes workload imbalance. The loop bounds for the partitions that scale are dependent on the size of the iteration space and for the ones that do not scale are based on a constant value. This is the reason for this extent of irregular scaling of the partitions. The scaling factors too, vary from partition to partition depending on the number of *faces* of the polyhedron, representing a partition, that are dependent on the iteration space size.

2.3 Parallel Execution of the Partitions

2.3.1 Re-tiling

Inconsistent geometries of the polygonal tiles for each individual partition can be noticed in figure 2. This property makes it hard to find a generalized solution for splitting every polyhedral geometry into chunks with equal workload.

Partition	Approx. Scaling Factor w.r.t. Dataset
C_6	1x
$\mathcal{D}C_1$, $\mathcal{D}C_4$	0.5x
$\mathcal{D}C_3$, C_1	0x

Table 1. Irregular scaling of partitions based on type

Size	$\lvert\mathcal{D}C_4\rvert$	$\lvert C_6\rvert$	Ratio ($\lvert C_6\rvert/\lvert\mathcal{D}C_4\rvert$)
128	1860	47250	26
256	3780	192786	52
512	7620	778770	103
1024	15300	3130386	205
2048	30660	12552210	410
4096	61380	50270226	820

Table 2. Iteration counts in C_6 and $\mathcal{D}C_4$ partitions

```
lbp=ceild(-N-31,32);
ubp=-1;
#pragma omp parallel for schedule(dynamic) private(lbv,ubv,t2,t3,t4)
for (t1=lbp;t1<=ubp;t1++) {
  for (t2=0;t2<=min(floord(N-4,32),-t1-1);t2++) {
    for (t3=max(-N,32*t1);t3<=min(32*t1+31,-32*t2-4);t3++) {
      lbv=32*t2;
      ubv=min(32*t2+31,-t3-4);
      for (t4=lbv;t4<=ubv;t4++) {
        x[t3][t4]             = y[t3][t3+t4+3]         + y[t3+t4][t4];
        x[-t4-3][t3+t4+3]     = y[-t4-3][t3+3]         + y[t3][t3+t4+3];
        x[-t3-t4-6][t3+3]     = y[-t3-t4-6][-t4]       + y[-t4-3][t3+3];
        x[-t3-6][-t4]         = y[-t3-6][-t3-t4-3]     + y[-t3-t4-6][-t4];
        x[t4-3][-t3-t4-3]     = y[t4-3][-t3-3]         + y[-t3-6][-t3-t4-3];
        x[t3+t4][-t3-3]       = y[t3+t4][t4]           + y[t4-3][-t3-3];
      }
    }
  }
}
```

Listing 2. Re-Tiled code for C_6 partition

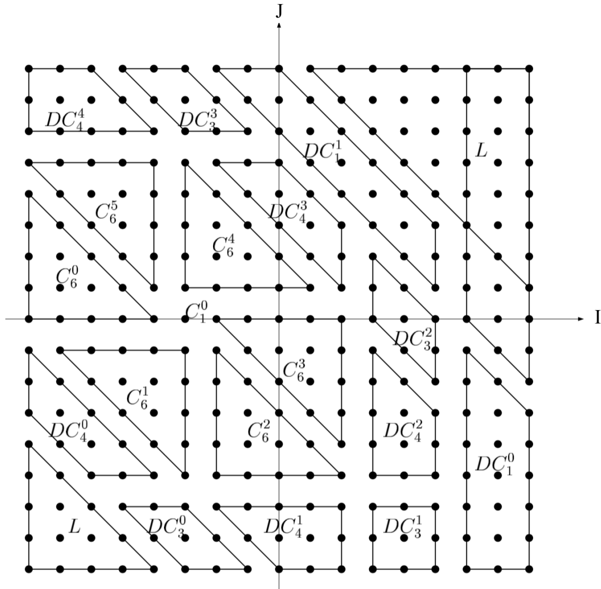


Figure 2. Polygonal Partitions for Case Study 1

Since it is not feasible to find equal splitting for every kind of partition, we go back to basic rectangular (re)tiling of the polygonal tiles. For re-tiling a type of partition, we consider each partition as an independent iteration space and since we can derive index for iterations in the same type of partitions as shown in the loop body of listing 2, re-tiling is required only for one partition. From here forward, partitions will refer to the polygonal partitions and tiles will refer to the tiles generated after re-tiling a polygonal partition. The benefit of generating rectangular tile is that during execution only a block of data is cached that could fit in the low level caches like L1-cache or L2-cache. If the outermost dimension of a partition is parallelized without tiling, data required for computation for an iteration overflows the capacity of the cache and hence locality is lost even with code optimizations.

An important benefit of re-tiling is that these tiles allow scalability of distributing partitions while keeping the locality optimizations intact. But rectangular tiling over the polygonally tiled code (with enabled optimizations like wavefront execution) presents various challenges such as loop-nests with MIN() and MAX() calls in both dimensions, example $\mathcal{DC}_4$ partitions in figure 2.

2.3.2 Splitting Partitions across the Outermost Loop

The concepts of splitting a partition from [14] and split domains from Razanajato et. al.[13] contribute towards reducing the control statement overhead. The latter work contributes to this by finding chambers that reduce the overhead and getting rid of MIN()/MAX() functions in the loop bounds. Getting rid of these functions also make loop bounds for some partitions turn into Static Control Parts or *SCoPs* on which tiling can be applied using PLuTo. In cases where it is not possible to transform loop bounds into SCoPs, we parallelize the outer-most loop using OpenMP directives.

Listing 2 presents an illustration of re-tiled polygonally tiled code for C_6 type of partition. We add the *schedule* directive for OpenMP since along the outermost dimension of the partition, the number of tiles can vary as we move along. To resolve this issue and balance workload inside the partitions too, we dynamically schedule these tiles such that the threads more or less execute similar number of re-tiles.

2.3.3 Tile Alignment along the Wavefronts

We also preserve the concept of wavefront execution from [14] since it helps reduce, both, the control statement overhead and the size of the generated code. Since the partitions in a wavefront follow certain patterns, as seen in figure 5, partitions can be coalesced under an outer loop that iterates from one wavefront to the other.

2.3.4 Load Balancing across Partitions

Scheduling the re-tiled code requires attention to an important factor i.e. each tile in the re-tiled code performs different number of operations/computations based on the type of polygonal partition. This issue is handled by computing one type of partition across all threads at a time. The illustration of this scheduling of partitions can be seen in Listing 3. This keeps the workload across threads balanced. As discussed earlier, the workload balance inside partitions is handled by dynamic scheduling.

3 Experiments and Analysis

Experiments were performed on Intel Xeon Phi Knights Landing CPU 7210 @ 1.30GHz (64 cores, 1MB L1-cache, 32MB L2-cache). The re-tiling on the polygonally tiled code was performed using PLuTo[2] with following flags: `--tile --parallel --noprevector`. For the performance analysis of the kernels described later in the section, the polygonally tiled code with re-tiling was compared against PLuTo tiled code (Tile-Size = 32x32). Both codes were compiled with Intel ICC v18.0.0 with following flags: `-O3 -xHost -fopenmp` and executed with following affinity settings: `OMP_PROC_BIND = spread` and `OMP_PLACES = threads`. The comparison is not performed against [14] since the strategy proposed in that work doesn't scale to the extent which is evaluated in this work. The two case studies represent two different classes of loop-nests exhibiting non-uniform reuse pattern. The kernels are tested with three different dataset sizes (N), each with different timing loop (T) configuration: Small Dataset (N=1024, T=1000), Medium Dataset (N=2048, T=400) and Large Dataset (N=4096, T=100).

3.1 Case Study 1: Two dimensional irregular reuse pattern

In this kernel (Listing 1), the reuse pattern varies across both dimensions of the iteration space. In figure 3, partitions belonging to the same type are represented with the same

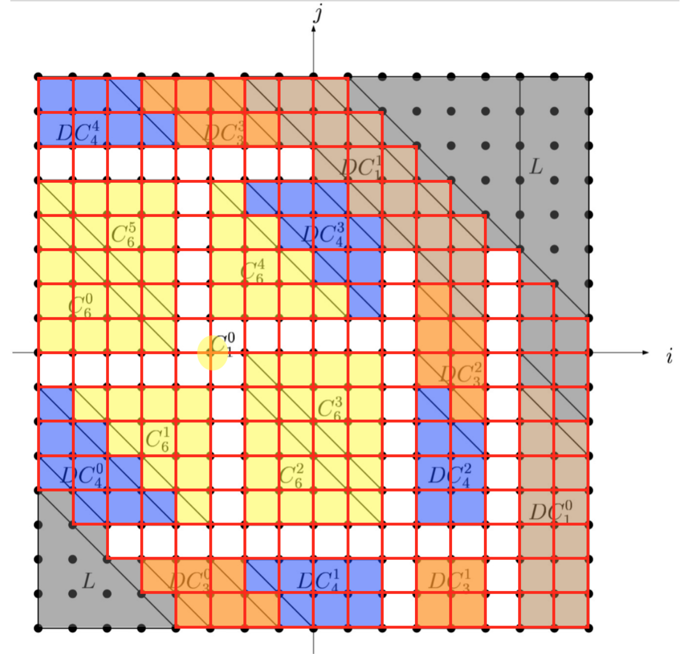


Figure 3. Re-Tiling of Polygonal Partitions for Case Study 1

color. Each tile in the figure represents the re-tiling of the partition. White space represents void space with no iterations. L partitions are not tiled since they have no reuse with any other iteration and hence each such iteration can be executed irrespective of any other iteration in the space. The performance improvement can be seen in figure 6a. The speedup reaches up to 2x over PLuTo tiled code.

Since the iterations in the partitions exhibiting same reuse pattern are linked by the temporal reuse relation $\mathcal{T}$, we can generate code to compute all linked iteration in the loop body of a partition. As shown in Listing 2, for computing all iterations in C_6 partitions, only the loop-nest computing

```
#pragma omp parallel for schedule(dynamic)
Loop-Nest: Re-tiled C6 partitions
#pragma omp parallel for schedule(dynamic)
Loop-Nest: Re-tiled DC4 partitions
#pragma omp parallel for schedule(dynamic)
Loop-Nest: Re-tiled DC3 partitions
#pragma omp parallel for schedule(dynamic)
Loop-Nest: Re-tiled DC1 partitions
#pragma omp parallel for schedule(dynamic)
Loop-Nest: Re-tiled C1 partitions
#pragma omp parallel for
Loop-Nest: L partitions
```

Listing 3. Scheduling pattern using OpenMP directives for Case Study 1

C_6^0 is re-tiled and thereafter the index for iterations having data reuse in $C_6^1, C_6^2, C_6^3, C_6^4$ and C_6^5 is derived (in the same order in the loop body). This process of index derivation for iterations having data reuse can be performed for all partitions in this kernel without increasing size of code too much, since no more than 6 iterations can be linked by the temporal reuse relation.

The result of improving data locality optimization i.e. polygonal partitions can be seen in figure 4. L2 Hit Rate percentage, gathered using Intel Vtune Amplifier 2018, shows the improvement in data reuse at L2-cache level.

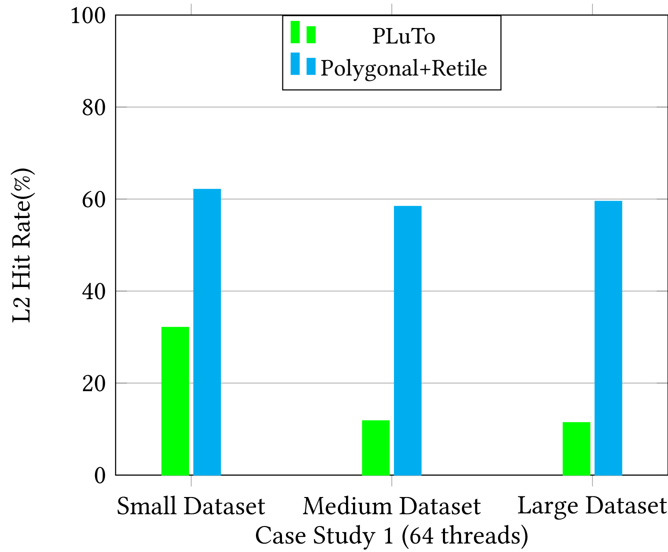


Figure 4. Improvement in Cache Locality

3.2 Case Study 2: One dimensional irregular reuse pattern

In the following kernel (Listing 4), the reuse pattern only varies across one-dimension, hence contributing towards regularity or similarity between the generated partitions.

```
for (i = -N; i <= N; i++) {
  for (j = -N; j<= N; j++) {
    X[i][j] = Y[i][j] + Y[i][i+j+N];
  }
}
```

Listing 4. Loop-Nest with 1-D Non-Uniform Reuse

The illustration of the effects of re-tiling the polygonal partitions while preserving the wavefront execution can be seen in figure 5. In this kernel too up to 2x speedup can be seen over PLuTo tiled code (figure 6b).

The code generated for this kernel is executed in wavefronts so that to reduce control statement overhead and reduce the size of code. Since in this case the partitions being created keep on reducing in size as we moved along

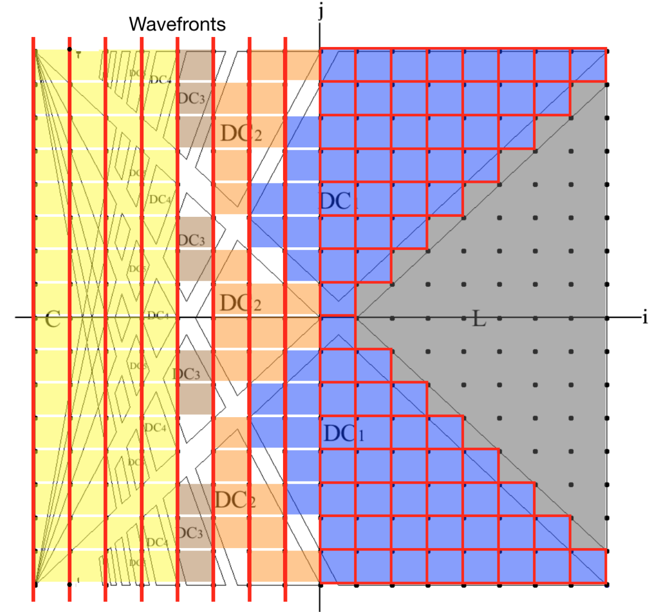


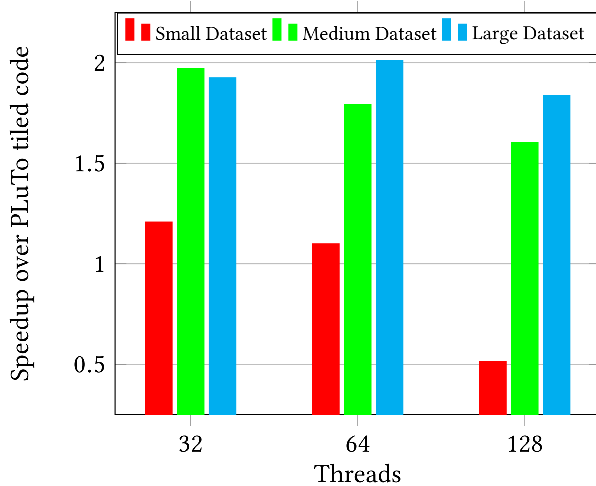
Figure 5. Re-Tiling of Polygonal Partitions for Case Study 2

the $-I$ direction, we stop generating partitions as soon as the iteration count in the partitions is reduced below 625, same as proposed in [14]. The rest of iteration are executed as part of the C partition. As we move along $-I$ direction the distance between iterations having data reuse keep on reducing resulting in closer, smaller and increasing number of partitions of similar type.

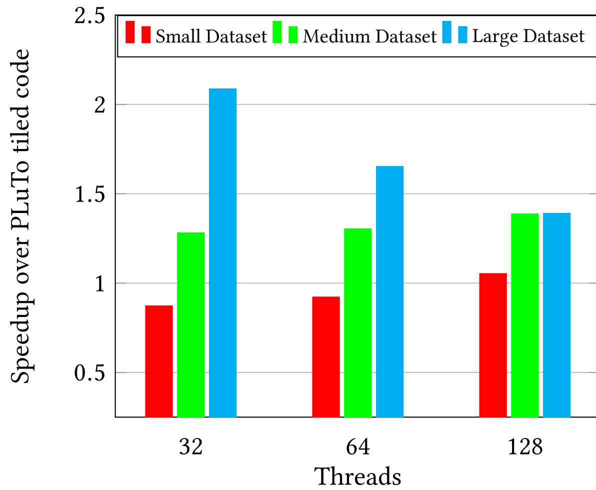
4 Prior Art

Distributing equal workload across a many core processor is challenge, especially when the amount of computation across the iteration space is not homogeneous. For a loop executing a non-uniform iteration space, load balancing requires careful partitioning of the loop. Static loop scheduling is not successful since only the outermost dimension of the loop is evenly partitioned and generated chunks are non-uniform. To solve this problem, scheduling approaches were proposed in [5, 7] that are based on profiling but require (re)analysis if the workload or dataset is changed.

Dynamic scheduling strategies provide a solution here since they schedule either iterations or chunk of iterations during runtime to the an idle processor. Dynamically scheduling large chunks of iteration create load imbalance whereas scheduling small chunks require more synchronization overhead. Different dynamic scheduling strategies manage chunk sizes to trade-off load balancing and scheduling overhead. Guided Self Scheduling (GSS)[12] which uses techniques like implicit coalescing of the loops followed by the scheduling where the size of chunks assigned to the processors are decreasing with time.



(a) Case Study 1: Performance



(b) Case Study 2: Performance

Figure 6. Speedup of Polygonal Tiles after Re-Tiling over PLuTo tiled code

Although the strategies mentioned above are very effective in many applications, they do not fit the requirement of polygonal tiles. Reason being the complex geometries of the partitions and also the complex loop bound calculations that do not allow precise static analysis.

5 Conclusions

This work presents an extension for scaling the execution of irregular polygonal tiles across multiple threads without compromising the data locality focused optimizations. This approach balances the workload across threads for the polygonal tiles by distributing the re-tiled partitions evenly across all available threads. The re-tiling code faces certain challenges due to the complex loop bounds which are addressed in the paper. Then, strategic scheduling of these partitions is performed to reduce the synchronization overheads incurred by OpenMP implementations. Experimental results show that re-tiling allows scaling and workload distribution of the polygonally tiled code with significant performance improvements.

References

[1] H. Bast. 2000. *Provably optimal scheduling of similar tasks.* Ph.D. Dissertation. Saarland University, Saarbrücken, Germany.

[2] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Program Optimization System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[3] J Mark Bull. 1998. Feedback guided dynamic loop scheduling: Algorithms and experiments. In *European Conference on Parallel Processing.* Springer, 377–382.

[4] Rosario Cammarota, Alexandru Nicolau, and Alexander V Veidenbaum. 2012. Just in time load balancing. In *International Workshop on Languages and Compilers for Parallel Computing.* Springer, 1–16.

[5] Ricolindo L. Cariño and Ioana Banicescu. 2008. Dynamic load balancing with adaptive factoring methods in scientific applications. *The Journal of Supercomputing* 44, 1 (01 Apr 2008), 41–63. https://doi.org/10.1007/s11227-007-0148-y

[6] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. 1992. Factoring: A Method for Scheduling Parallel Loops. *Commun. ACM* 35, 8 (Aug. 1992), 90–101. https://doi.org/10.1145/135226.135232

[7] A. Kejariwal and A. Nicolau. 2005. An Efficient Load Balancing Scheme for Grid-based High Performance Scientific Computing. In *The 4th International Symposium on Parallel and Distributed Computing (ISPDC'05).* 217–225. https://doi.org/10.1109/ISPDC.2005.14

[8] Arun Kejariwal, Alexandru Nicolau, Utpal Banerjee, Alexander V. Veidenbaum, and Constantine D. Polychronopoulos. 2009. Cache-aware Partitioning of Multi-dimensional Iteration Spaces. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference (SYSTOR '09).* ACM, New York, NY, USA, Article 15, 12 pages. https://doi.org/10.1145/1534530.1534551

[9] C. P. Kruskal and A. Weiss. 1985. Allocating Independent Subtasks on Parallel Processors. *IEEE Transactions on Software Engineering* SE-11, 10 (Oct 1985), 1001–1016. https://doi.org/10.1109/TSE.1985.231547

[10] Steven Lucco. 1992. A Dynamic Scheduling Method for Irregular Parallel Programs. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92).* ACM, New York, NY, USA, 200–211. https://doi.org/10.1145/143095.143134

[11] Benoît Meister, Vincent Loechner, and Philippe Clauss. 2000. *The polytope model for optimizing cache locality.* Technical Report. Technical Report RR 00-03, ICPS-LSIIT.

[12] C. D. Polychronopoulos and D. J. Kuck. 1987. Guided Self-scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Trans. Comput.* 36, 12 (Dec. 1987), 1425–1439. https://doi.org/10.1109/TC.1987.5009495

[13] Harenome Razanajato, Vincent Loechner, and Cédric Bastoul. 2017. Splitting Polyhedra to Generate More Efficient Code. In *IMPACT 2017, 7th International Workshop on Polyhedral Compilation Techniques.*

[14] Aniket Shivam, Alexandru Nicolau, Alexander V. Veidenbaum, Mario Mango Furnari, and Rosario Cammarota. 2017. *Polygonal Iteration Space Partitioning.* Springer International Publishing, Cham, 121–136. https://doi.org/10.1007/978-3-319-52709-3_11

Acknowledgments

This work was supported in part by NSF award XPS 1533926.