UNIVERSITY OF CALIFORNIA,
IRVINE

Polygonal Iteration Space Partitioning using the Polyhedral Model

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

by

Aniket Shivam

Dissertation Committee:
Professor Alexander V. Veidenbaum, Chair
Professor Alex Nicolau
Professor Ardalan Amiri Sani

2016

TABLE OF CONTENTS

LIST OF FIGURES

List of Algorithms

ACKNOWLEDGMENTS

I would like to thank Professor Alex Veidenbaum and Professor Alex Nicolau for their guidance and support during the course of this thesis. Special thanks to Dr. Ro Cammarota for his technical advice, enthusiasm and teaching abilities that ignited my interest in compiler research and also lead to the completion of this thesis.

I would also like to acknowledge Professor Ardalan Amiri Sani for serving as committee member and for teaching an interesting and research oriented course on operating systems.

Also, I would like to thank to Juan Besa Vial for proof reading the entire thesis and giving pointers on making it easier to comprehend.

Last but not least, I would like to thank my parents for always supporting and understanding my academic endeavors.

ABSTRACT OF THE DISSERTATION

Polygonal Iteration Space Partitioning using the Polyhedral Model

By

Aniket Shivam

Master of Science in Computer Science

University of California, Irvine, 2016

Professor Alexander V. Veidenbaum, Chair

Loop-nests in most scientific applications perform repetitive operations on array(s) and account for most of the program execution time. Traditional loop transformations, such as tiling, leverage data locality and maximize program performance on modern micro-architectures. These transformations, however, effectively maximize performance of programs when loop-nests exhibit uniform reuse patterns.

In this thesis, a new loop transformation is presented to target loop-nests with non-uniform reuse patterns. The proposed loop transformation uses the norms of the Polyhedral Model to represent the loop-nests and then leverages such a representation to partition the iteration space into polygonally shaped partitions. These partitions optimize locality resulting in an improvement in performance for both serial and parallel execution. Improving locality in parallel execution requires selective mapping of partitions on threads based on the type of reuse these partitions exhibit.

The experiments on certain loop-nests show that a significant portion of the achievable performance is missed when applying the traditional loop transformations. Compared to state-of-the-art Polyhedral Model frameworks, the transformation shows a consistent performance speedup in serial (up to 1.2x over Polly) and parallel (up to 3.17x over PLuTo) executions for certain loop-nests with non-uniform reuse patterns.

Chapter 1

Introduction

Loop-nests in most scientific applications perform repetitive operations on array(s) and accounts for the majority of the total execution time of the program. Popular benchmarks like `SPEC CPU` are evident of the impact of loop-nest on overall performance. Their performance depend heavily on the locality of the data accessed by the iterations and also on the amount of parallelism that can be exposed. Data locality depends heavily on the memory hierarchy of the underlying architecture, especially cache levels, size of the caches and their bandwidth. Whereas parallelism depends on the set of iterations that can run concurrently on separate cores.

There are variety of loop transformations for improving the performance of the loop-nests like tiling, loop-interchange, strip-mining, loop fusion, loop skewing and other optimizations that have been proposed in the past[36, 5]. These techniques target efficient utilization of the cache hierarchy. *Tiling* or *iteration space partitioning* has been explored as a major optimization for improving data locality. Better locality schemes not only reduce the latency in fetching data from lower level caches but also reduce the energy dissipated by the on-chip/off-chip components in fetching data.

In last three decades, various tiling techniques[25, 1, 3] have been proposed. Tiling partitions the iteration space into smaller blocks of iterations, known as *tiles*, such that the data accessed by the enclosed iterations effectively utilize either a single level of cache or multiple levels. This loop restructuring transformation also extract tiles so as to maximize reuse across loop iterations and among statements. Thereafter, an schedule is generated for these tiles such that they cover the complete iteration space at runtime. The shape and size of the tiles is determined at compile time based on several factors like dependency vectors, cache sizes, synchronization overhead, etc.

The introduction of Polyhedral Model for powering such optimizations may provide more insight about the structure and the dependences in the loops. This has an advantage in scheduling iterations and generating better parallel execution model for the loop-nest[11]. These optimizations are included in the modern compilers, such as `LLVM`[28], and in combination with other program transformations speedup the program execution.

Although tiling can successfully speedup the execution of loop-nests which exhibit uniform reuse pattern, tiling for loop nests with non-uniform reuse patterns has rarely been explored. The loop-nest in Fig. 1.1a exhibit a uniform reuse pattern since same data is used among consecutive iterations ($I_{i,j}$ and $I_{i+1,j+1}$). Whereas, the loop-nest in Fig. 1.1b reuse pattern changes for every iteration of the outer-most loop, hence the term non-uniform reuse pattern. These non-uniform reuse pattern arise due to the presence of two or more iteration variables in the references to an array. For the statement in Fig. 1.1b, the presence of iteration variables i and j in the reference $a[i+1][j+i]$ leads to the constant variation in reuse pattern as the iteration space expands along i.

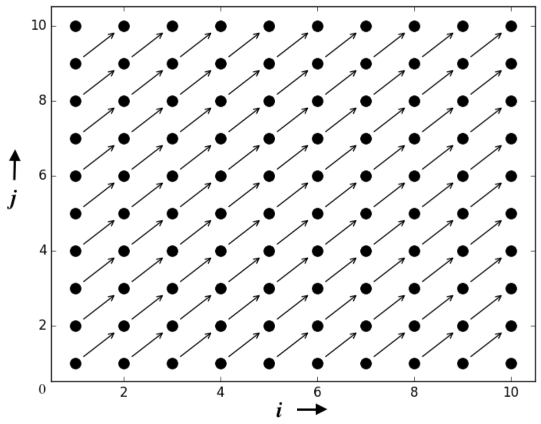
Tiling loop-nests with non-uniform reuse pattern is challenging because partitioning the entire iteration space into tiles of regular shape and size is not an optimal strategy. The distance between iterations that have reuse outgrows these tiles for a large part of the iteration space. Symmetric tiles spread over the iteration space forms a recurring pattern which facilitates easier code generation and less control statement overhead. A tile is an affine hyperplane which can precisely be defined using set of linear inequalities. These linear inequalities are transformed to control statements during code-generation. The complexity and irregularity in the tile shape directly escalates the additional instructions (Max, Min, Ceil and Floor) required in the control statement to define the bounds for the tile in the control statements. The technique proposed by Meister et. al.[35] for partitioning the iteration space is not constrained to either shape or size of the tiles. Their technique uses a set of mathematical computations to produce sets of iterations that reuse the same data. The price of this technique, however, is that for certain reuse patterns the number of partitions generated may be too high. The instruction overhead in managing these irregular tiles might overshadow the speedup achieved from the improvement in locality.

```
for (i = 0; i < N; i++) {
   for (j = 0; j < N; j++) {
      b[i][j]=a[i][j]+a[i+1][j+1];
   }
}
```

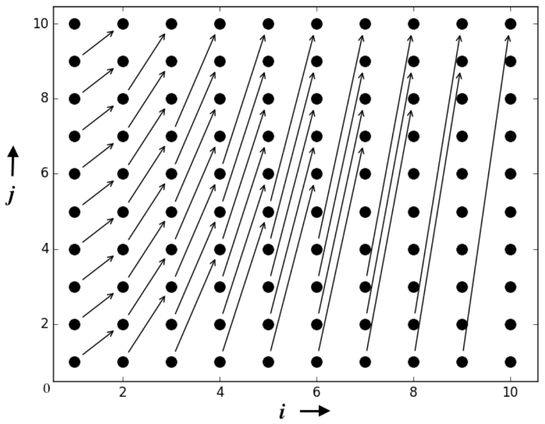
(a) Uniform reuse pattern

```
for (i = 0; i < N; i++) {
   for (j = 0; j < N; j++) {
      b[i][j]=a[i][j]+a[i+1][j+i];
   }
}
```

(b) Non-Uniform reuse pattern



(c) Reuse pattern for loop in (a)



(d) Reuse pattern for loop in (b)

Figure 1.1: Reuse patterns in loop-nests

State-of-the-art Polyhedral Model based code optimizers like `PLuTo`[12, 37] and `Polly`[22, 38] can efficiently tile loop-nests with uniform reuse pattern. But, they ignore RAR (Read-After-Read) dependence while finding the optimal tile shape. RAR dependence does not present any challenge in rescheduling the iterations unlike loop-carried dependencies, but impact the locality of data accesses. This work tries to exploit the presence of RAR dependencies to create partitions that increase data reuse in caches and hence reducing latency introduced by fetching from lower level of caches and memory.

In this work, a set of loop restructuring transformations are proposed to improve locality for the loop-nests with non-uniform reuse pattern. This work is an extension to the technique proposed by Meister et. al.[35]. The contributions from this work are as follows:

1. Variable shaped and sized Polygonal tiling is proposed for the perfectly nested loop-nests with non-uniform reuse. These tiles are divided into sets based on the reuse among iterations. Hence, each set of tiles improves locality on multiple levels of cache.

2. Determining the optimal halt condition for the algorithm. This condition assures that the benefit of such reuse-based partitioning is not outweighed by the instruction overhead necessary to execute the partitions at run-time.

3. Extending the algorithm to work for programs with multiple references after careful selection of the references having maximum reuse using a reuse calculation formula.

4. Strategic parallelism generations in such loops by mapping set of partitions, which possess reuse among themselves, on same core.

5. Extending application of the technique to some well-known scientific stencil benchmarks that exhibit uniform reuse pattern.

A comparison of the performance between the proposed technique and the state-of-the-art tiling techniques implemented in Polyhedral Model based code optimizers like Polly and Pluto is presented. Also, the shortcomings and mathematical limitations of the proposed technique are mentioned.

Chapter 2

Background

This chapter provides the mathematical background[1] required to understand the application of the polyhedral model to transform loop-nests into a set of integer points. Also, a detailed background is provided on various concepts and techniques used for optimizing loop-nests using the Polyhedral Model.

2.1 Polyhedron

Definition 1. ***Affine Hyperplane*** *The set of all vectors $\vec{v} \in \mathbb{Z}^n$ such that $\vec{h}.\vec{v} = k$, where $\vec{h}$ is a horizontal vector and $n \in \mathbb{Z}$, defines an affine hyperplane.*

Conceptually, a hyperplane is an $n-1$ dimensional affine (sub)space in n dimensional space. For a three dimensional space, a hyperplane can be visualized as a two dimensional slice. The value of k is different for a set of parallel hyperplanes, having h as their normal. Vectors lying on the same hyperplane will produce the same value of k i.e. $\vec{h}.\vec{v}_1 = \vec{h}.\vec{v}_2$. Hence, the characterizing feature for representing a set of hyperplanes facing a certain direction is the normal to these hyperplanes. Each hyperplane partitions the space into two parts: a positive *half-space* and a negitive *half-space*. The function to denote a half-space is represented as:

$$\phi(\vec{v}) = \vec{h}.\vec{v} + c \ (constant) \tag{2.1}$$

[1]General representation: $\mathbb{Z}$ - set of integers, $\mathbb{Q}$ - set of rational numbers and $\mathbb{R}$ - set of real number.

Definition 2. ***Convex Polyhedron*** *The set of all vectors $\vec{v} \in \mathbb{R}^n$ such that $A\vec{v} + \vec{b} \geq 0$, where A is an integer matrix, represents an convex polyhedron.*

A bounded polyhedron, also known as *polytope*, is a space confined by a set of affine inequalities, where each inequality represents a half-space. Therefore, a convex polyhedron is a space bounded by a set of hyperplanes. In geometric terms, a convex polyhedron is a space in which a line segment joining any two points lying inside the polyhedron does not intersect any face (hyperplane), edge or corner of the polyhedron. For a polyhedron in an n-dimensional space with m inequalities, A will be a matrix of m x n dimensions, vectors $\vec{v}$ and $\vec{b}$ will be represented by n dimensional and m dimensional vectors respectively. Since, in the polyhedral model we are concerned with integer loop bounds and loop iterators, it can be represented as:

$$\{\vec{v} \in \mathbb{Z}^n | A\vec{v} + \vec{b} \geq 0\} \tag{2.2}$$

2.1.1 Representation of the polyhedron

The domain of the polyhedron is represented by the set of inequalities derived from the bounds of each loop present in the loop-nest. A set of inequalities must be written for each loop to define the bounds for that particular loop iterator. In modern Polyhedral Model implementations like `PLuTo`[12, 37], `Polly`[22, 38] and `PolyLib` [39], each statement, S, enclosed in the loop-nest (perfectly nested[2] or imperfectly nested[3]) is represented using the *iteration vector* ($\vec{i_S}$). Each iteration vector is represented as a matrix where each row define the lower and upper bounds for the nested loops in terms of inequalities, starting from the outermost to the innermost loop enclosing the statement. Hence, each statement is determined by a polytope representing its *domain* (D_S).

Polyhedral optimizations work on the *Static Control Parts (SCoPs)* of the program. SCoPs are parts of the programs for which the control flow and memory access pattern can be defined at compile time. These parts consist of for-loops and if-conditions. The expressions defining the loop bounds must be known at time of compilation to generate the correct polytope. Therefore, SCoP eligible loop bounds are generally the combination of outer loop iterators and/or a known constant. The same conditions apply to the subscripts of the arrays

[2]All statements are nested inside the innermost loop.

[3]Statements are not necessarily nested inside the innermost loop.

```
for (i = 0; i < N; i++)
  for (j = 0; j <  N; j++)
    for (k = 1; k <=  i; k++)
      S(i,j,k);
```

(a) Valid SCoP

$$D_S = \begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \\ \end{array}\left(\begin{array}{ccccc} i & j & k & N & 1 \\ 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & -1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & -1 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{array}\right) \begin{array}{c} \\ i \geq 0 \\ i \leq N-1 \\ j \geq 0 \\ j \leq N-1 \\ k \geq 1 \\ k \leq i \\ \\ \\ \end{array}$$

(b) Domain for the statement in the SCoP

Figure 2.1: Representation of a valid SCoP

that are accessed in the loop-nest. The optimizations are performed on all statements that are present in the SCoP. An example of a valid SCoP and the polyhedral representation of its domain is shown in Fig. 2.1.

An important aspect of the polyhedral model are constituted by *Lattice* and *$\mathcal{Z}$-polyhedron.*

Definition 3. ***Lattice*** *A subset of $\mathbb{Q}^n$ generated as a combination of integral number of vectors $\vec{v}_i$, where $\vec{v}_i \in \mathbb{Q}^n$, represents a Lattice (L).*

A lattice is an integral lattice if each vector representing it has integral coordinates.

Definition 4. ***$\mathcal{Z}$-polyhedron*** *An intersection of a polyhedron P and a affine full dimensional integral lattice L i.e. $\mathcal{Z} = P \cap L$.*

The $\mathcal{Z}$-polyhedron is used for representing the accessed data points in a data space D_S of a variable generally an array. The bounds for the data space polyhedron D_S can be generated using the variable's subscripts and bounds of the loop variables.

Definition 5. ***Image*** *The image of a polyhedron $P \in \mathbb{Z}^n$ by an affine function f: $\mathbb{Z}^n \to \mathbb{Z}^m$ is a $\mathcal{Z}$-polyhedron i.e. $\mathcal{Z} = \{f(\vec{x}) \in \mathbb{Z}^m | \vec{x} \in P\}$*

An affine function mapping each iteration to a data point in the accessed data space, defines an Image. This affine function can be considered as a function based on a Lattice for mapping a integer point in the polyhedron (iteration) to the data point (memory unit) it accesses. This is shown in Fig. 2.2.

```
for (i = 0; i <= N; i++)
   for (j = 0; j <=  N; j++)
      S: A[3i][j+2] = i + j;
```

(a) A sample code

$$\begin{aligned}
P =& \{i, j | 0 \leq i \leq N, 0 \leq j \leq N\} \\
L =& \{3i, j+2 | i, j \in \mathbb{Z}\} \\
\mathcal{Z} =& P \cap L \\
=& \{3i, j+2 | 0 \leq 3i \leq N, -2 \leq j \leq N-2\}
\end{aligned}$$

(b) Derivation of a $\mathcal{Z}$-polyhedron

Figure 2.2: Representation of a $\mathcal{Z}$-Polyhedron for computing accessed memory

Definition 6. ***Pre-Image*** *The pre-image of a polyhedron $P \in \mathbb{Z}^n$ by an affine function f: $\mathbb{Z}^n \to \mathbb{Z}^m$ is a $\mathcal{Z}$-polyhedron i.e. $\mathcal{Z} = \{f(\vec{x}) \in \mathbb{Z}^n | f(\vec{x}) \in P\}$*

Therefore, $Image(f^{-1}, P) = Pre\text{-}Image(f, P)$ iff f is invertible.
The $\mathcal{Z}$-polyhedron is an extension to the polyhedra model. It provides precise analysis of periodic domains i.e. domains containing "holes". Such cases arise in loop-nests with non-unit strides and non-unimodular transformations, where the application of general polyhedral model becomes a challenge. The application and importance of $\mathcal{Z}$-polyhedron for the optimization and parallelization of loop-nests can be found in [43, 34, 33, 23].

2.1.2 Dependence and Reuse Relation

Dependency relation between iterations of a loop-nest can be characterized based on the order and type (read or write) of usage of the same memory location. Lets assume a sequential execution of a loop-nest with a single statement S and any two iterations i and j from the iteration space I. First, if a memory location is read by I_i after it is written by I_j during the execution, then I_i is *flow dependent* (or Read-After-Write or RAW) on I_j. Second, if a memory location is read by I_i before it is written by I_j, then I_i is *anti-dependent* (or Write-After-Read or WAR) on I_j. Third, if I_i writes a memory location after it is written by I_j, then I_i is *output dependent* (or Write-After-Write or WAW) on I_j. Fourth, similarly if I_i reads a memory location after it is read by I_j, then I_i is *input dependent* (or Read-After-Read or RAR) on I_j. Input dependence is better characterized as *Reuse* of a memory location rather than a dependency. Also, dependencies can also be characterized based on the existence of dependency for a particular loop. For a nested loops like the following:

```
for ( i = 0; i < N; i++)
      for ( j = 1; j <=  N; j++)
```

```
S: A[i][j] = A[i][j-1] + A[i][j] + A[i][j+1];
```

There exists an *loop-carried* dependence in the innermost loop, since there are flow dependence and anti-dependence between successive iterations. But, the outermost loop inhabits the *loop-independent* part, since the same flow dependence will exist with or without this loop. Presence of loop-carried dependence restricts the possibility of parallelization across the loops, but parallelization may still be exploited for other outer or inner loops that carry no such dependence.

Definition 7. ***Dependence Graph*** *The Dependence Graph is a directed multigraph $G = (V, E)$, where each vertex represents a statement i.e. $V = S$ and every directed edge e ($e \in E$) represents a dependence from the source statement to the target statement.*

Another important property of the dependence relation is its transitivity as mentioned in [8]. The *transitivity closure* of the dependence relation establishes that a dynamic instance of S, say S_i is *indirect dependent* on another instance, S_j if there exists a directed path from S_i to S_j in the dependence graph. This property is important for the application of the optimization technique mentioned in Chapter 4.

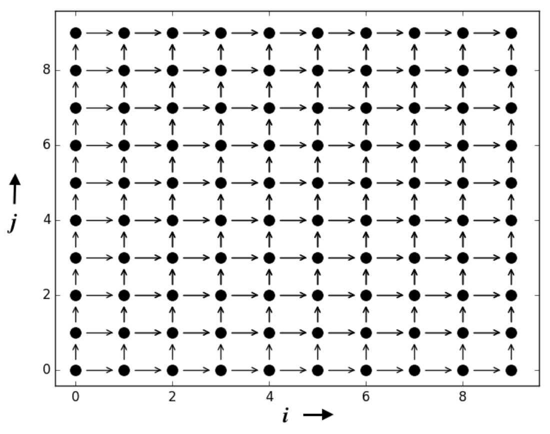
An important factor for defining the nature of dependency is the *distance* of the dependence. This *distance* represents the instances of S that must be executed between the execution of the source and the target statement. Based on the distance of dependence, the dependence relation can be classified into three kinds. First, *uniform dependence* where the distance between two dependent statements is constant throughout the iteration space. Second, *non-uniform dependence* where the distance between the source and target statements varies throughout the iteration space. This can also be looked as varying dependence pattern across the iteration space. We mainly try to address the issues in optimizing such programs and propose a solution in the coming chapters. Third, *parametric dependence* where the distance is guided by an external parameter which may or may not remain constant.

2.2 Dependence Analysis in the Polyhedral Model

The main step in performing dependence analysis on a polyhedral representation starts with the definition of the *dependence polyhedron* (P). This analysis is based on techniques first presented in [25, 17, 41]. The dependence polyhedron helps in accurately applying the

```
for(i = 0; i < N ; i++) {
  for(j = 0; j < N ; j++) {
    A[i][j] += A[i][j+1]+A[i+1][j]
  }
}
```

(a) Loop-nest exhibiting two WAR dependency



(b) Dependences in the iteration space

$$\begin{array}{c} \begin{array}{cccccc} i & j & i' & j' & N & 1 \end{array} & \\ \left(\begin{array}{cccccc} \mathbf{0} & \mathbf{-1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{-1} \\ \mathbf{-1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 1 & -2 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & -3 \end{array}\right) & \begin{array}{c} \mathbf{-j+j'-1=0} \\ \mathbf{-i+i'=0} \\ i \geq 0 \\ i \leq N-1 \\ N-2 \geq 0 \\ j \geq 0 \\ j \leq N-3 \end{array} \end{array}$$

(c) Dependence Polyhedron for WAR dependency (A[i][j] → A[i][j+1])

$$\begin{array}{c} \begin{array}{cccccc} i & j & i' & j' & N & 1 \end{array} & \\ \left(\begin{array}{cccccc} \mathbf{0} & \mathbf{-1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{-1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{-1} \\ 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & -2 \\ 0 & 0 & 0 & 0 & 1 & -2 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & -2 \end{array}\right) & \begin{array}{c} \mathbf{-j+j'=0} \\ \mathbf{-i+i'-1=0} \\ i \geq 0 \\ i \leq N-2 \\ N-2 \geq 0 \\ j \geq 0 \\ j \leq N-2 \end{array} \end{array}$$

(d) Dependence Polyhedron for WAR dependency (A[i][j] → A[i+1][j])

Figure 2.3: Dependence Polyhedron for a loop-nest

mathematical models like Farkas lemma that is used for finding the legal schedules for the hyperplanes. The dependence polyhedron is a collections of linear equalities and inequalities that define the scope of accessed data. This assumes an affine relation between the iterations of the loop-nest and the data accessed within these iterations. The inequalities in dependence polyhedron are based on the number of loop iterators and parameters involved in a particular dependency and factors like intra-statement or inter-statement dependency. Whereas, the equalities represent the relation between the dependent iterations. A explanatory example is shown in Fig. 2.3, where first two rows in the dependence polyhedron shows the dependence relation. State-of-the-art mathematical library `isl`[49] is used by several polyhedral optimizers to compute dependences from the polyhedral representation of the loop-nests.

2.3 Transformations

Polyhedral transformations are a combination of several one-dimensional affine transformations. An *affine transformation* can be explained as a transformation of a convex polyhedron into another convex polyhedron. These transformations maintain the polyhedral representation, allow further transformation and most importantly fit the requirement of widely used code generation tools like `CLooG`[10][16]. Transformations can be divided into two main categories: unimodular [7, 50] and non-unimodular[29, 44, 45, 51]. Unimodular transformations generate regular integer polyhedron i.e. loop with unit stride, whereas non-unimodular transformations may generate sparse integer polyhedron which present difficulties in code generation since it contain loops with non-unit stride.

A multi-dimensional transformation for a loop-nest is a combination of one-dimensional transformations, corresponding to each level of the loop-nest, for each of the enclosed statements. For a loop-nest of depth d, the affine hyperplane corresponding to a transformation can be represented as $\phi^k_{S_i}$, where $k \geq d$ and S_i represent one of the statements inside the loop-nest. The reason for k being greater or equal to d is that extra rows are added to the transformation matrix to accommodate the *scalar dimensions* as mentioned in [13]. Scalar dimensions are a constant function with no hyperplane specification. They specify the level at which the following hyperplanes/transformation for different statements can be fused together. The generic mathematical representation of a transformed loop-nest with hyperplanes (T) and scalar dimensions ($\vec{s}$) is shown in Fig. 2.4c. Scalar dimensions allow efficient nesting of loop-nest during code generation. The example in Fig. 3.1 shows the transformation matrices for a transformation on a code involving matrix multiplication.

Each transformation affecting a statement is supposed to satisfy the dependences involving the statement mentioned in the dependence polyhedron P. This is accomplished by using heavy machinery like **Farkas Lemma** which is a non-negative linearized combination of boundary-hyperplanes of the dependence polyhedron. This ensures the legality of transformation too, since each transformation must satisfy the Farkas Lemma representation. A detailed application and methodology for finding affine transformation using Farkas Lemma can be seen in [13]. The same condition applies for finding parallelism in the loop-nest. If all the dependences that were not satisfied until level $k-1$ are satisfied at level k of the loop-nest, then the loop at level k is said to be communication-free and hence can be parallelized. An automatic model for maximizing locality and exploiting parallelism using Polyhedral Model is described in [12, 11].

```
for(i = 0; i < M; i++) {
   for(j = 0; j < N; j++) {
      for(k = 0; k < K; k++) {
         S1:C[i][j]+=A[i][k]*B[k][j];
      }
   }
}
for(i = 0; i < M; i++) {
   for(j = 0; j < N; j++) {
      for(k = 0; k < K; k++) {
         S2: D[j][k]+=C[i][j];
      }
   }
}
```

(a) Original Code

```
for(j = 0; j < N; j++) {
   for(i = 0; i < M; i++) {
      for(k = 0; k < K; k++) {
         S1: C[i][j]+=A[i][k]*B[k][j];
      }
      for(k = 0; k < K; k++) {
         S2: D[j][k]+=C[i][j];
      }
   }
}
```

(b) Optimized Code

$$\mathcal{T}_{S_i} = \begin{pmatrix} \phi^1_{S_i} \\ \phi^2_{S_i} \\ \phi^3_{S_i} \\ . \\ . \\ . \\ \phi^k_{S_i} \end{pmatrix} = T_{k\times d}.\vec{i}_S + \vec{s}\,, \; where \; \vec{s} = \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ . \\ . \\ . \\ s_k \end{pmatrix}$$

(c) Generic transformation matrix for a statement S_i

$$\mathcal{T}_{S_1}(\vec{i}_{S_1}) = \begin{array}{c} \begin{array}{ccc} \text{i} & \text{j} & \text{k} \end{array} \\ \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

(d) Transformation matrix for S_1

$$\mathcal{T}_{S_2}(\vec{i}_{S_2}) = \begin{array}{c} \begin{array}{ccc} \text{i} & \text{j} & \text{k} \end{array} \\ \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

(e) Transformation matrix for S_2

Figure 2.4: Dependence Polyhedron for a loop-nest

2.3.1 Scheduling and Partitioning based parallelization techniques

Parallelization techniques using the Polyhedral Model can be categorized into two major categories: Partitioning-based and Scheduling-based. Partitioning-based approaches [31, 30, 32] scan the iteration space to identify affine partitions that optimize for parallelism and minimize communication, whereas Scheduling-based approaches[18, 19] schedule iteration so as to maximize number of parallel sections of the iteration space with minimum synchronization. The technique presented in Chapter 4 fits as the partition-based technique, since we determine affine partitions that can be executed in parallel.

2.4 Code generation for the Polyhedral Model

The code generation for Polyhedral Model was first presented in [4] based on the Fourier-Motzkin Elimination [48]. However, this method was restricted to a single polyhedron code generation and unimodular transformation. The major challenge in code generation for the Polyhedral Model is the size and redundancy of the generated code. This problem gets worse when there are multiple polyhedron to generate code for. This problem was addressed in [20] by merging the loop-nest of different polyhedron but it generates redundant control statements. This work also proposed a method to generate code for non-unimodular transformations. The work from [42] improved on the problem of redundant control statements by computing the union of all the polyhedrons and thereafter, generating code for subsets of disjoint polyhedrons. This work was later improved by [9] in terms of performance by reducing redundant polyhedral computations by applying pattern matching between polyhedrons. This is implemented in the widely used polyhedral code generation tool, `CLooG` [16].

Chapter 3

Related Work

Multi-level caches and shared off-chip memory between processors requires a smart use of memory to reduce communication overhead and thus achieving better performance. It is critical to: (1) maximize the number of cache hits and (2) minimize the number of cache access from lower levels of memory. An effective approach from the compiler's point-of-view is to reorder instructions so that instructions accessing the same data (or data on same cache line) are executed in proximity. In terms of loop-nests, this implies to grouping iterations so as to exploit data locality. Doing so reduces the time and energy spent in transferring data from off-chip memory or the Last Level of Cache (LLC) to L1 and/or L2 caches or even registers. A loop transformation for generating such groups of iterations is called Tiling.

3.1 Tiling

Tiling divides an iteration space into uniformly shaped partitions with their size being either pre-determined based on the underlying architecture or dynamically varied (using parameterized loop bounds) during execution. In addition to improving locality, tiling also controls the granularity for parallelism to reduce the communication overhead across tile boundaries. Tiling can be a recursive transformation for exploiting locality on different levels of memory hierarchy. Tiling as mentioned in [50] may sometimes be referred as *blocking*[26], *strip-mine and interchange* and *iteration space partitioning*. The legality conditions for tiling are presented in [25, 51] make sure that the tiling hyperplane does obey all the dependences in the loop-nest.

Generation of tiles generally doubles the number of loops in a loop-nest. The inner half of the loop-nest executes the iterations within a tile and the outer half of the loop-nest determines the execution order of the tiles. This increases the control overhead of the program compared to the original program. This overhead is a drawback of using Tiling. An example loop-nest shown in Fig. 3.1a has loop-carried dependencies. The code is then tiled with the code-optimizer Pluto[37] for locality and parallelism. Since, it carries dependences along direction (0,1) and (2,1), tile execution (either serial or parallel) must not violate any of these dependences across the tile boundaries when executed in parallel.

3.2 Multi-Level Tiling

Tiling was extended to skewed tiling by [25] for perfectly nested loops to improve locality for two-levels of cache. The work in [1, 2] provided a theoretical framework for deriving parallelepiped tiling for minimal communication among multi-processors with shared memory. They also present a mathematical methodology for computing the memory/data footprint for an iteration space and for estimating the communication required among processors. Another such method for computing the integer points (data points) accessed in a loop using the Polyhedral Model is shown in [15] by computing the Ehrhart polynomials. Tiling for imperfectly nested loops was first proposed in [3] by applying concept of forming a *product space* i.e. the cartesian product of each statement's iteration space to apply transformations. Modern optimizers like Pluto[12] can handle and optimize imperfectly nested loops. Multi-level tiling for a Matrix-Multiplication kernel is shown in Fig. 3.2. The matrix multiplication kernel is optimized for two level tiling using Pluto is shown in Fig. 3.2c, where outer-most three loops (t1, t2 and t3) execute tiles which can be seen as sets of sub-tiles. These sub-tiles that are parsed by the subsequent three loops (t4, t5 and t6). Hence, accomplishing multi-level tiling for two levels in the memory hierarchy.

Tiling presents a challenge for multi-level cache since finding the optimal tile size should be based on parameters such as the capacity of each level of cache and the memory bandwidth. This issue has been addressed by [46, 14] for determining the optimal tile size at compile time.

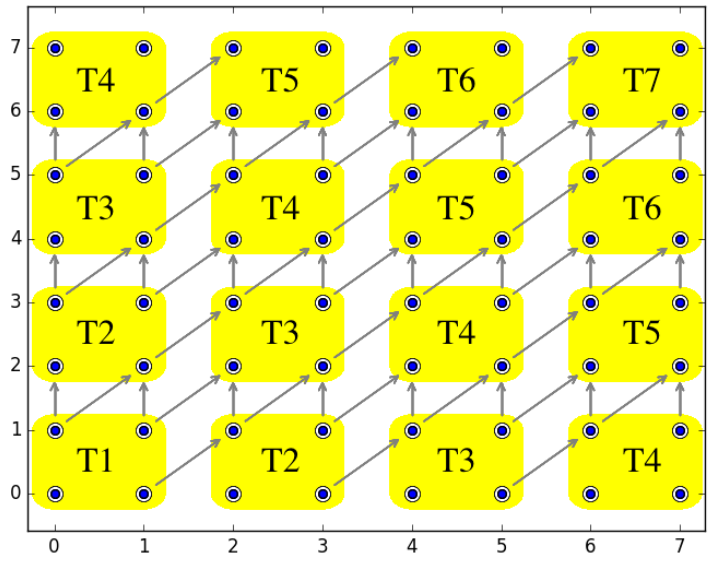
```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        a[i][j] = a[i+1][j]+a[i+1][j+1];
    }
}
```

(a) Original Code

```
#define ceild(n,d)  ceil(((double)(n))/((double)(d)))
#define floord(n,d) floor(((double)(n))/((double)(d)))

/* Inter-tile loops t1 and t2 */
for (t1 = 0; t1 < N; t1++) {
#pragma omp parallel for private(t3,t4)
  for (t2 = max(0,ceild(2*t1-N+1,2)); t2 < min(floord(N-1,2),t1); t2++) {
/* Intra-tile loops t3 and t4 */
    for (t3 = 2*t1-2*t2; t3 <= min(N-1,2*t1-2*t2+1); t3++) {
      for (t4 = 2*t2; t4 <= min(N-1,2*t2+1); t4++) {
        a[t3][t4] = a[t3][t4+1] + a[t3+1][t4+1];
      }
    }
  }
}
```

(b) Pluto optimized code: Tiling (Tile size = 2 × 2) with OpenMP directives for parallelism



(c) Tiling pattern (Tile size = 2 × 2, N = 8): Tiles are executed as numbered (T1, T2, T3,... and so on). Parallelism is achieved by executing same numbered tiles in parallel. Arrows represent dependence.

Figure 3.1: Tiling for locality and parallelism

```
for(i = 0; i < M; i++)
  for(j = 0; j < N; j++)
    for(k = 0; k < K; k++)
      C[i][j] = beta*C[i][j] + alpha*A[i][k] * B[k][j];
```

(a) Original Code

```
/* Inter-tile loops t1, t2 and t3 */
for (t1=0;t1<=floord(M-1,8);t1++)
  for (t2=0;t2<=floord(N-1,128);t2++)
    for (t3=0;t3<=floord(K-1,8);t3++)
/* Intra-tile loops t4, t5 and t6 */
      for (t4=8*t1;t4<=min(M-1,8*t1+7);t4++)
        for (t5=8*t3;t5<=min(K-1,8*t3+7);t5++)
          for (t6=128*t2;t6<=min(N-1,128*t2+127); t6++)
            C[t4][t6] = beta*C[t4][t6] + alpha*A[t4][t5] * B[t5][t6];
```

(b) Single Level of Tiling for L1 cache (Tile size = 8 × 128 × 8)

```
/* Level 1 Tiling: Inter-tile loops t1, t2 and t3 */
for (t1=0;t1<=floord(M-1,128);t1++)
  for (t2=0;t2<=floord(N-1,256);t2++)
    for (t3=0;t3<=floord(K-1,128);t3++)
/* Level 2 Tiling: Inter-tile loops t4, t5 and t6 */
      for (t4=16*t1;t4<=min(floord(M-1,8),16*t1+15);t4++)
        for (t5=2*t2;t5<=min(floord(N-1,128),2*t2+1);t5++)
          for (t6=16*t3;t6<=min(floord(K-1,8),16*t3+15);t6++)
/* Intra-tile loops t7, t8 and t9 */
            for (t7=8*t4;t7<=min(M-1,8*t4+7);t7++)
              for (t8=8*t6;t8<=min(K-1,8*t6+7);t8++)
                for (t9=128*t5;t9<=min(N-1,128*t5+127);t9++)
                  C[t7][t9]=beta*C[t7][t9]+alpha*A[t7][t8]*B[t8][t9];
```

(c) Multi-Level of Tiling for L1 cache (8 × 128 × 8) and L2 cache (128 × 256 × 128)

Figure 3.2: Rectangular Tiling and Multi-Level Tiling

3.3 Optimal Tile Size and Parametrized Tiling

Determining the tile size at compile-time usually produce suboptimal solution since the cache sizes for the target architecture are not known in many situations. Therefore, using symbolic parameters - left to be determined at runtime - as loop parameters presents an opportunity for dynamic optimization. **Parameterized tiling** techniques[47, 27] based on the Polyhedral Model has shown that it is possible to get comparable, if not better, performance and parallelism[24] from the parameterized tiled loop-nests as compared to statically compile-time generated tiled loop-nests. Parametric tiling presents challenges in scheduling tiles for both single core or multi-core processing in the presence of dependencies because tile shapes and size may vary during runtime optimization of the code. An attempt to handle this problem was made in [24] by adding code to the program at compile time which at runtime generate sets of tiles that can be executed in parallel and also generate the schedule for these sets. The technique presented in Chapter 4 is independent of the tile size constraint. The size of the tiles is solely determined by the reuse pattern of the loop-nest.

3.4 Modern Tiling Geometries

In addition to the variable sized tiles, some recent work on the exploration of newer tiling geometries have shown some promise, especially for stencil computations. The work in [6] shows that diamond-shaped tiles - when executed in parallel - can achieve concurrent start for the tiles which might not have been possible with regular rectangular/parallelogram tiles. Tiling in the shape of variable-sized Hexagons[21] provides better locality and concurrent execution of tiles for parallel architectures like GPUs. But, varying tile shapes for better locality has not received similar attention. The Polygonal tiling technique presented in the next chapter is not bound to a specific tile shape. Instead, tile shapes are determined based on the iteration space's reuse pattern.

Chapter 4

Polygonal Tiling

In this chapter, the technique for generating the polygonal partitions of a loop-nest is presented. This loop transformation exposes locality in the loop-nests exhibiting non-uniform reuse pattern and therefore improving the performance. This transformation is an improvement over the partitioning technique described in [35] in the following ways:

- A halt condition is proposed for the original technique such that the overhead due to control statements for managing partitions does not overshadow the performance gained from the improvement in locality.
- Improvised code generation for efficient execution of polygonal partitions.
- A technique for the mapping and scheduling partitions on threads during parallel execution of loops with no data dependence is suggested.
- An application to multi-reference statements and references across statements is also described.

4.1 Determining Reuse using the Polyhedral Model

Based on the mathematical representation of loop-nests in the Polyhedral Model as described in Chapter 2, mathematical equations can be derived for identifying the data accessed by the references in a statement. Each instance of a statement, S, enclosed in a loop-nest may also be defined by an iteration vector $\boldsymbol{I}$ for the multi-dimensional iteration space. For instance, in an iteration S accesses the data from an array $\mathbf{A}$, then using Equation 4.1 the exact location of the data can be calculated. In the equation, $A(I)$ represents a data point accessed in A. The *reference matrix*, R, is based on the coefficient of the iteration variables in the subscript representing the data access in A. Whereas, the *offset vector*, r, 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.

$$A(\boldsymbol{I}) = \mathbf{R} \times \boldsymbol{I} + \mathbf{r}, \text{where I represents an iteration.} \tag{4.1}$$

To provide a explanatory example, consider the following loop-nest:

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

The reference Y[i,i+j+3] in the above example references a two dimensional array Y enclosed in a two dimensional loop-nest. Therefore, R will be a 2×2 matrix, $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$. Each row represent the projection of the reference along each dimension of the array i.e. value of subscript in each dimension (i and i+j+3). Whereas, 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 can be substituted using a column vector I, $\begin{pmatrix} i \\ j \end{pmatrix}$.

Each reference to the array is an unique combination of (R, r). The pair is represented as Γ to locate the accessed data point by an iteration. Γ is a function which computes the *image* of the polyhedron. In the above loop-nest, the two references to the array Y are written as follows:

$$\Gamma_{i,i+j+3} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \mathbf{I} + \begin{pmatrix} 0 \\ 3 \end{pmatrix} \qquad and \qquad \Gamma_{i+j,j} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \mathbf{I} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Let's say there is reuse of a data by two different references Γ_α and Γ_β in iterations I_α and I_β respectively. The dependence between two iterations can be formally described by Equation

4.2.

$$\Gamma_\alpha = \Gamma_\beta \Leftrightarrow R_\alpha I_\alpha + r_\alpha = R_\beta I_\beta + r_\beta \tag{4.2}$$

Therefore, as suggested in [35], the temporal reuse relation or dependence relation $\mathcal{T}$ between I_α and I_β can be formally represented by Equation 4.3, deriving from Equation 4.2.

$$R_\beta^{-1} R_\alpha I_\alpha + R_\beta^{-1}(r_\alpha - r_\beta) = I_\beta \Leftrightarrow T_{\alpha\beta} I_\alpha + t_{\alpha\beta} = I_\beta, \quad \textsf{iff } R \textsf{ is invertible.} \tag{4.3}$$

The reuse relation $\mathcal{T}$ is a combination of $(T_{\alpha\beta}, t_{\alpha\beta})$, where $T_{\alpha\beta} = R_\beta^{-1} R_\alpha$ and $t_{\alpha\beta} = R_\beta^{-1}(r_\alpha - r_\beta)$. Substituting a particular iteration in place of I_α yields another iteration (I_β) that reuses the same data.

An important condition to be noted here is: if and only if R is invertible. It is a necessary condition for the invertibility of T and computing the reuse relation $\mathcal{T}$. Therefore, the *reference matrix* R needs to be an square matrix. This implies that it is critical for the application of this technique that the dimensions of the involved array is same as the depth of the loop-nest. This in turn makes possible to determine that the data accessed by I_β using Γ_β is also accessed by I_α using Γ_α, $I_\alpha = T_{\alpha\beta}^{-1} I_\beta$ - $T_{\alpha\beta}^{-1} t_{\alpha\beta}$.

Therefore, the temporal reuse relation $\mathcal{T} = (T, t)$ for the loop-nest in the example is:

$$T = \begin{pmatrix} 0 & -1 \\ 1 & 1 \end{pmatrix} \, and \, t = \begin{pmatrix} -3 \\ 3 \end{pmatrix} \, using\ Equation.(4.3)$$

In the example, to check if the data accessed by an iteration, say i = 3 and j = 2, using reference $\Gamma_{i,i+j+3}$, is also accessed by another iteration using the reference $\Gamma_{i+j,j}$. Substituting the iteration vector by (3,2) in Equation 4.3, $\begin{pmatrix} 0 & -1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 3 \\ 2 \end{pmatrix} + \begin{pmatrix} -3 \\ 3 \end{pmatrix}$, yields vector (-5,8). Therefore, it can be concluded that iterations (3,2) and (-5,8) have reuse.

4.2 Partitioning Technique

The partitioning technique proposed in this work requires the representation of the loop in the polyhedral framework. The goal of the partitioning technique is to identify and execute non-adjacent portions or partitions of the iteration space in an order which ensures that the data is reused cache before being flushed out. Whereas, in an unoptimized version of the program, this data would have been flushed out of the cache before its reuse. These partitions are thereafter grouped based on the locality of the data their iterations access. Hence, all the partitions accessing the same set of data are agglomerated.

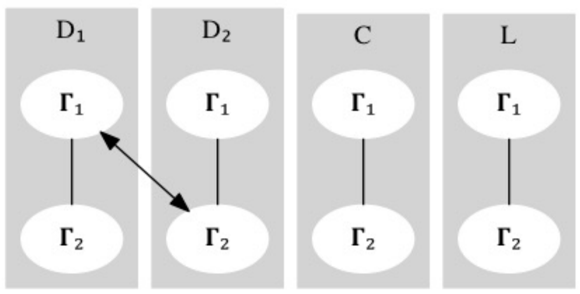
For the mathematical formulation of the technique, assume there are two references Γ_α and Γ_β to an array in a single statement in the loop nest. The primary step is to partition the iteration space ($\mathcal{D}$) in three sets denoted by L, $\mathcal{P}_1$ and $\mathcal{P}_2$.

- $\mathcal{P}_1$ contain iterations that reference the data using Γ_α that another iteration in $\mathcal{D}$ accesses by Γ_β i.e. these iterations have an *image* in $\mathcal{D}$ using relation $\mathcal{T}$.
- Iterations referencing the data using Γ_β that is also referenced by another iteration in $\mathcal{D}$ using Γ_α form set $\mathcal{P}_2$. These iterations are the *images* of the iterations in $\mathcal{P}_1$. In other words, they have a *Pre-Image* in $\mathcal{D}$ (*Image*($\mathcal{T}^{-1}$,$\mathcal{D}$)).
- 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 $\mathcal{L}$. These iterations neither project nor they are projected in $\mathcal{D}$ using $\mathcal{T}$. Hence, $\mathcal{D}$ = $\mathcal{P}_1$+$\mathcal{P}_2$+$\mathcal{L}$.

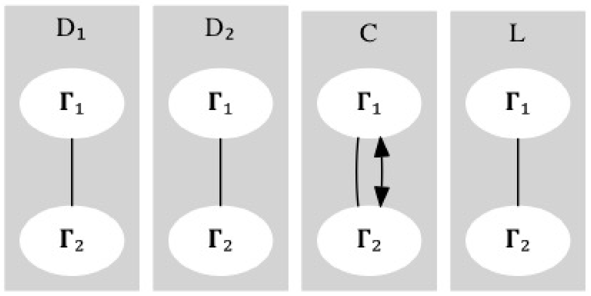
The sets $\mathcal{P}_1$ and $\mathcal{P}_2$ can be further categorized into three subsets $\mathcal{C}$, $\mathcal{D}_1$ and $\mathcal{D}_2$, in addition to $\mathcal{L}$.

- $\mathcal{C}$: These iterations belong to both $\mathcal{P}_1$ and $\mathcal{P}_2$, i.e., $\mathcal{C}$=$\mathcal{P}_1 \cap \mathcal{P}_2$. Data accessed by these iterations using both the references (Γ_α and Γ_β) is also accessed by other iterations.
- $\mathcal{D}_1$: These iterations belong to $\mathcal{P}_1$ only, i.e., $\mathcal{D}_1$=$\mathcal{P}_1$-$\mathcal{C}$ or $\mathcal{D}_1$=$\mathcal{P}_1$-$\mathcal{P}_2$. The data accessed by Γ_α of these iterations is accessed by other iterations. Data accessed by Γ_β is not reused.
- $\mathcal{D}_2$: These iterations belong to $\mathcal{P}_2$ only, i.e., $\mathcal{D}_2$=$\mathcal{P}_2$-$\mathcal{C}$ or $\mathcal{D}_2$=$\mathcal{P}_2$-$\mathcal{P}_1$. Similarly, the data accessed by these iterations using Γ_β is reused, whereas data accessed by Γ_α remains unused.

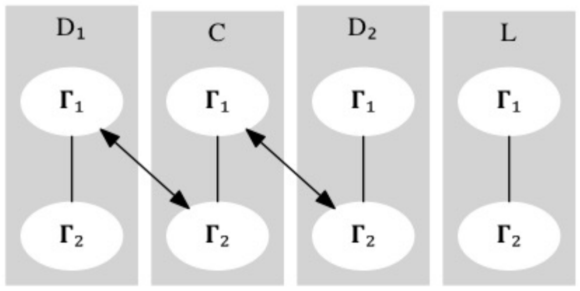
After categorizing the iterations based on the reuse of their accessed data, a further sub-categorization is performed such that each subset when executed in an order maximize the temporal locality. That is, iterations having reuse among them and forming smaller partitions($\mathcal{DC}_k$ and $\mathcal{C}_k$) are linked together. Fig. 4.1 shows an graphical illustration of categorizing iterations.



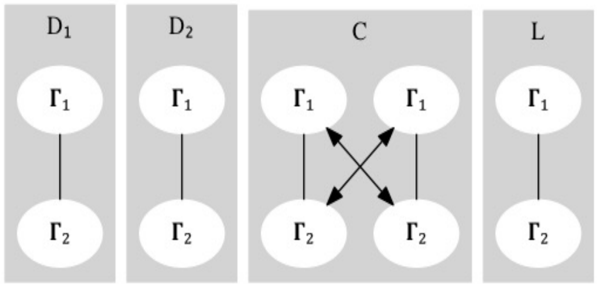
(a) $\mathcal{DC}_1$



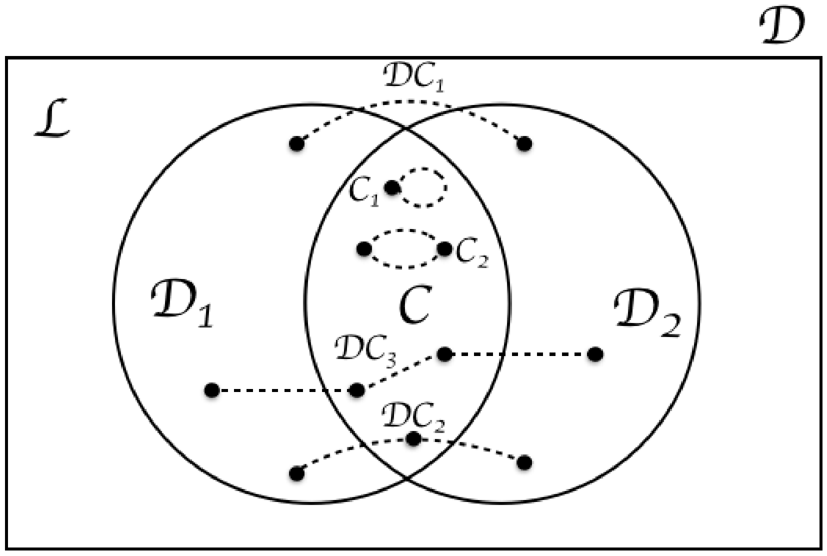
(b) $\mathcal{C}_1$



(c) $\mathcal{DC}_2$



(d) $\mathcal{C}_2$



(e) Set Representation

Figure 4.1: Classification of iterations - formation of the sets $\mathcal{DC}_1$, $\mathcal{C}_1$, $\mathcal{DC}_2$, $\mathcal{C}_2$.

- $\mathcal{DC}_1$: $\mathcal{D}_1$ iterations that link to $\mathcal{D}_2$ iterations by $\mathcal{T}$, i.e., $\mathcal{DC}_1 = \mathcal{D}_1 \cap \mathcal{T}^{-1}(\mathcal{D}_2)$.

- $\mathcal{C}_1$: $\mathcal{C}$ iterations that are linked to themselves by $\mathcal{T}$, i.e., $\mathcal{T}(\mathcal{C}_1) = \mathcal{T}^{-1}(\mathcal{C}_1)$.

- $\mathcal{DC}_2$: $\mathcal{D}_1$ iterations that link to $\mathcal{C}$ iterations that link to $\mathcal{D}_2$ iteration, i.e., $\mathcal{D}_1$ iterations that link to $\mathcal{D}_2$ iterations by $\mathcal{T}^2$, $\mathcal{DC}_2 = \mathcal{D}_1 \cap \mathcal{T}^{-1}(\mathcal{C}) \cap \mathcal{T}^{-2}(\mathcal{D}_2)$.

- $\mathcal{C}_2$: The remaining $\mathcal{C}$ iterations that form cyclic-link with one other iteration in $\mathcal{C}$ i.e., $\mathcal{C}$ iterations that are linked to themselves by $\mathcal{T}^2$, $\mathcal{C}_2 = \mathcal{C} \cap \mathcal{T}^{-1}(\mathcal{C}) \cap \{I \in \mathcal{C} | T^2 I + Tt + t = I\} - \mathcal{C}_1$.

After k repetitions of the previous steps:

- $\mathcal{DC}_k$: $\mathcal{D}_1$ iterations that link to chain of $k-1$ $\mathcal{C}$ iterations and at the end link to a $\mathcal{D}_2$ iteration by $\mathcal{T}^k$, i.e., $\mathcal{DC}_k = \{I \in \mathcal{D}_1 | Tt + t \in \mathcal{C}, T^2 I + Tt + t \in \mathcal{C}, ..., T^k I + T^{k-1} t + ... + Tt + t \in \mathcal{DC}_2\}$.

- $\mathcal{C}_k$: The remaining $\mathcal{C}$ iterations that are linked to themselves by $\mathcal{T}^k$ forming a cyclic-link of k $\mathcal{C}$ iterations, i.e., $\mathcal{C}_k = \{I \in \mathcal{C} | Tt + t \in \mathcal{C}, T^2 I + Tt + t \in \mathcal{C}, ..., T^k I + T^{k-1} t + ... + Tt + t = \mathcal{C}\} - \{\mathcal{C}_1 + ... + \mathcal{C}_{k-1}\}$.

These repetitive steps generate partitions based on the length of consecutive iterations sharing data reuse. This partitioning technique requires a halting condition such that the number of steps of the algorithms, k, can be determined and so does determines the number of partitions that it creates. As mentioned in [35], the value of k can be chosen as follows:
(a) if after the k^{th} repetition, the entire iteration space ($\mathcal{D}$) is completely partitioned. This value of k can be pre-computed. When $\mathcal{T}$ is represented in the form:

$$\mathcal{T} = \begin{pmatrix} T & t \\ 0...0 & 1 \end{pmatrix}$$

$\mathcal{D}$ is completely partitioned after k repetitions, if $\mathcal{T}^k$ is an identity matrix.

(b) if value of k is preset, the algorithm stops after the k repetitions and put the rest of the iterations in $\mathcal{C}_{k+1}$.

The partitions categorized as either $\mathcal{DC}_i$ or $\mathcal{C}_i$, where $1 \leq i \leq k$, are disjoint partitions spread across the iteration space. Therefore, the partitions labelled as $\mathcal{DC}_i$ can be numbered based on the position of their containing iterations in the chain. In the $\mathcal{DC}_i$ partitions, the first partition containing only $\mathcal{D}_1$ iterations are labelled as $\mathcal{DC}_i^0$. The next $i-1$ partitions containing $\mathcal{C}$ iterations are labelled as $\mathcal{DC}_i^1$, $\mathcal{DC}_i^2$,...,$\mathcal{DC}_i^{i-2}$ and $\mathcal{DC}_i^{i-1}$. The last partition in the chain containing $\mathcal{D}_2$ iterations is labelled as $\mathcal{DC}_i^i$. The same naming paradigm is followed for $\mathcal{C}_i$ partitions. These i partitions are labelled as $\mathcal{C}_i^0$, $\mathcal{C}_i^1$,...,$\mathcal{C}_i^{i-2}$ and $\mathcal{C}_i^{i-1}$. The number of iterations in the partitions of similar type is always equal, since the iterations in the successive partitions are the *images* of the iterations in the previous partition.

4.3 Orchestrating Formation of the Partitions

4.3.1 Premature Halting

An indiscriminate application of the algorithm introduce overhead at run-time due to large number of small sized partitions, which is not considered in the halting conditions defined above. In fact, such halting conditions do not take into the account the fact that the increase in the number of partitions increases the control statement overhead in the restructured loop-nest. This leads to partitions with very few iteration, whose gain in performance from better locality is overshadowed by the control overhead needed to manage such partitions. Also, the technique confines to loop nests with non-uniform reuse, but does not explore its applicability on uniform reuse loop nests like stencils, which are explored in the next chapter.

In this work, a termination method for this technique is proposed so that the control statement overhead does not overshadow the speedup gained through maximizing locality, by predicting minimum tile size, so that speedup remains intact. Specially in loop-nests where the longest chain of linked iteration is very long, i.e., T^k generates an identity matrix for a very high value of k, say k_{max}, it is critical to find an optimal value of $k < k_{max}$ to protect gained speedup from increasing control overhead. This is applicable to most loop nests with one dimensional non-uniform reuse pattern. Therefore, the algorithm is halted after partitioning for $\mathcal{T}^k$ and the remaining iterations form partition $\mathcal{C}_{k+1}$. From the experiments, it is determined that the algorithm must be halted if the number of iterations in generated partitions is below 25×25.

4.3.2 Multi-Level Tiling

The partitions generated on each repetition of this technique are labelled as $\mathcal{DC}_i$ and $\mathcal{C}_i$, where $1 \leq i \leq k$. Partitions labelled as $\mathcal{DC}_i$ or $\mathcal{C}_i$ are set of separate and distantly located partitions of the iteration space. The execution order of these partitions influences the improvement in locality or improved cache hit-miss ratio at a certain cache level. A single partition targets the improvement in locality in the smallest cache with the least expensive data transfer cost, ideally L1 cache. The set of partitions in $\mathcal{DC}_i$ or $\mathcal{C}_i$ targets a larger cache that can be either L2 or L3 cache. This technique guarantees that for loops with non-uniform reuse pattern, the cost in terms of time spent in fetching data for reuse is reduced by making it available in closest possible cache level.

4.4 Multi-Reference Statements

In this work the technique is also extended to statements having multiple references to the array as seen in CPU benchmarks and also to stencil computations that exhibit fixed pattern reuse in multiple directions. Every pair of temporal reuse relations lead to different partitions which on combining would generate a single partition. Reuse along multiple directions create a complex network of iterations linked by $\mathcal{T}$, therefore it is important to eliminate reuse relations such that iterations do not link to themselves by either $\mathcal{T}$ or $\mathcal{T}^2$. For example, the reuse vector $\vec{v}_{i,j-1}$ and $\vec{v}_{i,j+1}$ link themselves by $\mathcal{T}^2$. Therefore, one of them must be eliminated. Also, $\vec{v}_{i,j}$ must be eliminated since it links to itself by $\mathcal{T}$. One drawback of the original algorithm is that some pairs of reuse vectors produce partitions which consume the entire iteration space like $\vec{v}_{i,j+1}$ and $\vec{v}_{i+1,j}$. These pairs are eliminated. The aim is to find the '*pair*' (best set of two references) from all the references that generate the best possible partitions for maximizing locality.

Another heuristics to choose the *pair* is to select it based on the amount of reuse in the partitions that it creates. A ***reuse count function*** as shown in Equation 4.4 is used to predict the amount of reuse in the partitions can be appended in the original technique. This step involves choosing the best *pair* out of every set of two references - from those left after eliminating the redundant references - based on the amount of reuse that can be calculated from size of $\mathcal{DC}$s and $\mathcal{C}$s sets. When the algorithm is prematurely halted to reduce control statement overhead as described in the previous section, the residual iterations that form

$\mathcal{C}_{k+1}$ are not counted towards the reuse.

$$Reuse(\Gamma_\alpha,\ \Gamma_\beta) = \sum_{i=1}^{k} i \times |\mathcal{DC}_i^0| + \sum_{i=1}^{k} i \times |\mathcal{C}_i^0| \tag{4.4}$$

4.4.1 Reuse between Multiple Statements

This technique can also be extended to multiple statements enclosed in a loop-nest. Since, reuse of data from an array might occur between references spanning across multiple statements. These multiple references can be reduced to the best *pair* of references exploiting the maximum locality.

4.5 Locality on Parallel Execution of the Partitions

The iteration in the loop-nests without any loop-carried dependences (RAW, WAR and WAW) can be executed in parallel without any constraints. But tiling such loops can improve the performance by improving locality so that the cost of data transfer is reduced. During parallel execution more fetches from private memory and lesser fetches from the shared memory improves the performance. Scheduling each kind of partitions ($\mathcal{DC}_i$ and $\mathcal{C}_i$, $1 \leq i \leq k$) on different threads achieves the improvement in locality, since each thread finds the required data in private memory.

4.6 Code Generation paradigm

The code generation for these partitions begins by analyzing the polyhedron representation for each partition. This polyhedron representation contains the constraints (boundary hyperplanes) that define the affine boundaries for the partitions. These constraints are then scanned using the Fourier-Motzkin algorithm implemented in Polylib [39] and also using tools like CLooG[10][16]. Cloog generates code by scanning the polyhedrons and performs the union of distinct polyhedron to produce code with the least control statement overhead. The work in [35] suggests a methodology to scan just the initial partition from each category i.e. $\mathcal{DC}_i^0$ for the $\mathcal{DC}_i$ type partitions and $\mathcal{C}_i^0$ for $\mathcal{C}_i$ type partitions. The next steps is to derive the subscripts for the next iterations in the link using the reuse relation $\mathcal{T}$. Let, I, a column vector, represent the iterations in the $\mathcal{DC}_i^0$. The subscript for the iterations in the following partitions $\mathcal{DC}_i^1$, $\mathcal{DC}_i^2$,..., $\mathcal{DC}_i^i$ are derived from $\mathcal{T}(I)$, $\mathcal{T}^2(I)$,..., $\mathcal{T}^i(I)$ respectively. The locality is exposed in the successive statements since there is reuse between I and $\mathcal{T}(I)$ iteration, then in $\mathcal{T}(I)$ and $\mathcal{T}^2(I)$ iteration, etc.

This methodology is efficient unless the value of k is high in which case it enormously expands the code size. The loop-nest for $\mathcal{DC}_i$ and $\mathcal{C}_i$ partitions encloses $i+1$ and i statements respectively. For some value of k, the code will have a minimum of k loop-nests for either $\mathcal{DC}$ or $\mathcal{C}$ type partitions and maximum of $k \times 2$ (k $\mathcal{DC}$ plus k $\mathcal{C}$) loop-nests. Each of them containing statements between 0 and k. For a higher value of k, a better solution is to find the union of the polyhedron representing a type of partitions ($\mathcal{DC}_i$ or $\mathcal{C}_i$) to generate code.

Also, since each partition in $\mathcal{DC}_i$ or $\mathcal{C}_i$ type partitions contain equal iterations, they tend to form similar geometries. These geometries are recurring patterns and hence code generation for them requires slight modification in the boundary conditions of the control statements. These modification can be captured to form a basis for iterating through each partition of a particular type. Hence, reducing the total count of loop-nests in the code.

An important part of the speedup comes from the optimal computational paradigm of these partitions which include re-partitioning the generated partitions to reduce boundary check overheads. This is performed by computing these partial partitions and scanning them so as compute multiple partitions in a single loop.

For generating parallel code, the use of OpenMP® Sections is proposed. It allows the *selective mapping* of a certain type of partitions onto a single thread. This improves the locality of each thread which in turn reduces the fetching of same data from shared memory on multiple threads.

4.7 Limitations

The limitations of this technique are:

- Inapplicability to loops with references that do not generate an invertible *reference matrix* R and hence an invertible $\mathcal{T}$. The current state of $\mathcal{Z}$-Polyhedral model could not eliminate this limitation of the technique.

- Inability to generate a schedule of the partitions for a loop-nest with data dependence without destroying locality. If the execution of partitions as per the technique violates any data dependence, then modifying the execution order without violating dependence disrupts locality.

4.8 Overall Algorithm

Algorithm 1 Polygonal Tile Generation

```
1: **Input**: A loop-nest with potential reuse on a dataset (array).
2: Eliminate set of references that link iterations to themselves by either $\mathcal{T}$ or $\mathcal{T}^2$. (Section 4.4)
3: for each set of two references ($\Gamma_\alpha$, $\Gamma_\beta$) to the array do
4:     Define the Reuse Relation $\mathcal{T}$ using the two references $\Gamma_\alpha$ and $\Gamma_\beta$.
5:     Generate coarse partitions of the iteration space ($\mathcal{D}$):
6:     $\mathcal{P}_1$ ($Image(\mathcal{T},\mathcal{D})$), $\mathcal{P}_2$ ($Image(\mathcal{T}^{-1},\mathcal{D})$) and $\mathcal{L}$ (No reuse).
7:     Categorize $\mathcal{P}_1$ and $\mathcal{P}_2$ into:
8:     $\mathcal{C}=\mathcal{P}_1 \cap \mathcal{P}_2$, $\mathcal{D}_1=\mathcal{P}_1 - \mathcal{P}_2$ and $\mathcal{D}_2=\mathcal{P}_2 - \mathcal{P}_1$.
9:     while $\mathcal{D}$ is not completely partitioned do
10:        Create partitions ($\mathcal{DC}_i$ and $\mathcal{C}_i$) that have iterations linked by relation $\mathcal{T}^i$.
11:        if Iterations in the generated partitions is below $25 \times 25$ then
12:            $k = i$ (Since the algorithm is halted, $k$ is set to $i$.)
13:            Put rest of the iterations in $\mathcal{C}_{k+1}$.
14:            break
15:        end if
16:        Increment $i$.
17:    end while
18: end for
19: Remove the set of references that produce a single partition which consume the entire iteration space. (Section 4.4)
20: On the remaining set of references, apply the **Reuse Count Formula** (Equation 4.4) to estimate the amount of reuse.
21: Choose the *pair* having the maximum reuse in their polygonal partitions for code generation.
22: Scan the polygonal partitions using the Fourier-Motzkin algorithm to generate the boundaries for the partitions.
23: Use the code generation tools like CLooG with modifications so as to generate array subscripts using the function $\mathcal{T}^i(I)$.
24: **Output**: Polygonally tiled iteration space that maximizes locality.
```

Chapter 5

Case Studies and Experiments

In this chapter, four case studies on loop-nests with non-uniform reuse pattern are presented. These studies are supported with experiments and comparisons with state-of-the-art loop transformation techniques.

5.1 Experimental Setup

In the following sections, four case studies are presented. The loop-nests shown in these studies exhibit different styles of reference. These styles are:

- **Two Dimensional Non-Uniform Reuse**: The reuse pattern in the loop-nest varies along both dimensions of a two dimensional iteration space. This occurs when both of the references to the array has more than one iteration variables that govern the access pattern, it leads to a non-uniform reuse along both the dimensions.
- **One Dimensional Non-Uniform Reuse**: The reuse pattern varies along a single dimension. This occurs when only one of the references has two or more iteration variables to govern the access pattern.
- **Uniform Reuse**: Symmetrical reuse is present in various scientific computations like stencil computations. The reuse of data in these computation is generally among neighboring iterations along a certain direction(s) that lead to the uniform reuse pattern.

- **Multiple References**: Many benchmarks like PolyBench[40] exhibit loop-nests with multiple references to an array in a single statement. These references lead to reuse along various directions. Therefore, it requires analysis of all the references before partitioning such iteration space for locality.

For each study, the performance of the Polygonal Iteration Space Partitioning technique is compared against code optimized using the state-of-the-art Polyhedral Model based code optimizers Polly (LLVM Plugin)[38] for serial execution and PLuTo[12, 37] for parallel execution. Polly is chosen for parallel execution because it generates better schedules for regular tiles on parallel execution and supports Diamond tiling. The compiled codes are analyzed for performance on Intel's Sandy-Bridge Core i7-2600 CPU @ 3.40GHz. The processor has 4 cores (8 threads) with 256 KB L1 I/D cache, 1024 KB L2 cache and 8 MB LLC. Hardware performance counters were analyzed for measuring performance metrics.

5.2 Case Study 1: Two Dimensional Non-Uniform Reuse Pattern

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

Figure 5.1: Case 1: Loop-Nest with Two Dimensional Non-Uniform Reuse

In the loop-nest shown in Fig. 5.1, the references to the array Y can be represented as:

$$\Gamma_{i,\ i+j+3} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \mathbf{I} + \begin{pmatrix} 0 \\ 3 \end{pmatrix},\ \Gamma_{i+j,\ j} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \mathbf{I} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Therefore, the temporal reuse relation $\mathcal{T} = (T, t)$ can be calculated using Equation 4.3 as:

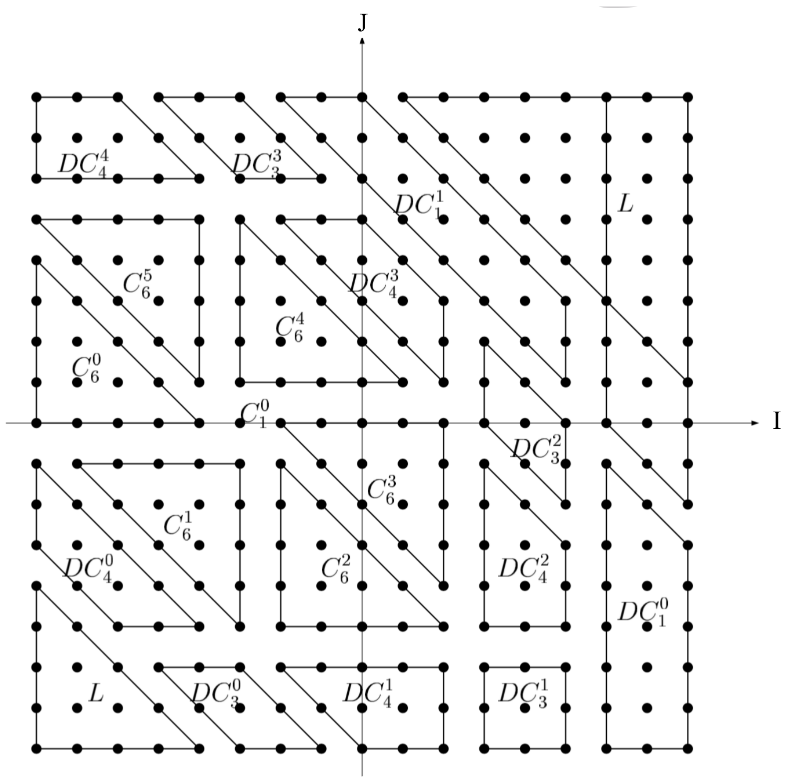
$$T = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix},\ t = \begin{pmatrix} -3 \\ 3 \end{pmatrix}$$

For this case k comes out to be 6, since T^6 is an identity matrix. Therefore, the partitioning process would terminate after six repetitions of the core algorithm. The remaining iterations in $\mathcal{C}$ are placed in partition $\mathcal{C}_6$ as described in the technique. The graphical representation of the partitioned iteration space is shown in Figure 5.2a [35]. The partitioning algorithm generates a fixed number of partitions, which is independent of the matrix size. Hence, the partitions generated from this technique are scalable with the dataset size. Since, the maximum value of k is 6, it generates a small number of partitions which suggests that the control statement overhead will have negligible effect on performance. Therefore, there is no need to apply the halting condition described in Section 4.3 in this case. Hence, the maximum value must be chosen to obtain the finest partitions with the maximum reuse.

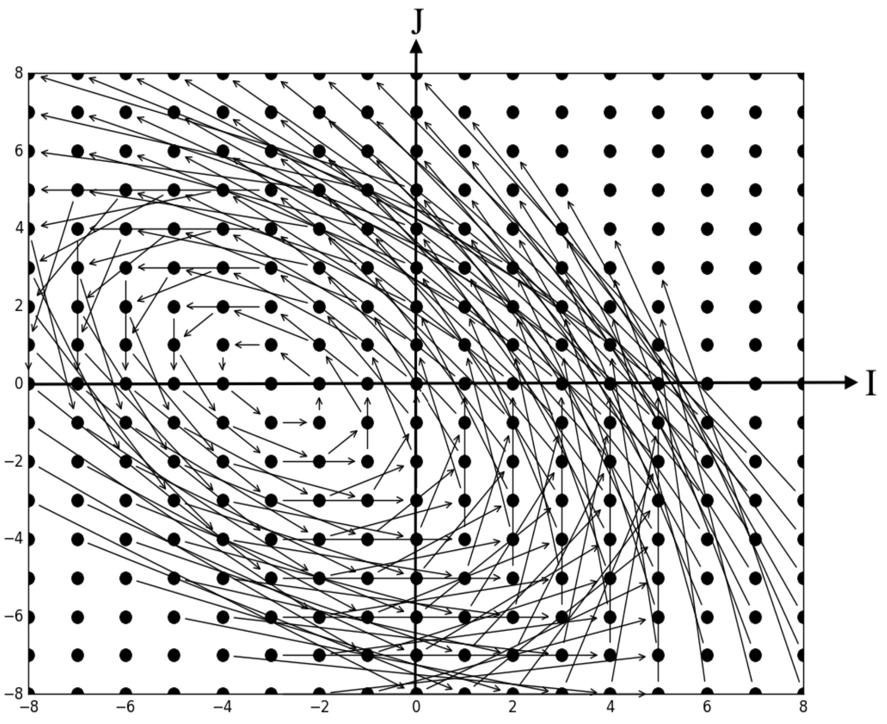
```
for (i = -N; i <= -4; i++) {
    for (j = MAX(-N+3,-i-N-3); j <= -i-N-1; j++) {
        X[i][j]              = Y[i][i+j+3]  * Y[i+j][j];
        X[-j-3][i+j+3]       = Y[-j-3][i+3]  * Y[i][i+j+3];
        X[-i-j-6][i+3]       = Y[-i-j-6][-j]  * Y[-j-3][i+3];
        X[-i-6][-j]          = Y[-i-6][-i-j-3]  * Y[-i-j-6][-j];
        X[j-3][-i-j-3]       = Y[j-3][-i-3]  * Y[-i-6][-i-j-3];
    }
}
```

Figure 5.3: Index calculation for $\mathcal{DC}_4$ using reuse relation($\mathcal{T}$).

Since the partitions of a particular type (say $\mathcal{DC}_1$) has multiple disjoint but equivalent partitions ($\mathcal{DC}_1^0$, $\mathcal{DC}_1^1$, etc.) - in terms of shape and number of iterations - the function $\mathcal{T}^i$(I), where $0 \leq i \leq 6$, can be used to determine the iterations in the other partitions which exhibit reuse as described in Section 4.6. The code shown in Fig. 5.3 presents the application of the function to compute array subscripts for $\mathcal{DC}_4$ partitions. This optimization reduces the control statement overhead, as well as increases the temporal locality due to consecutive data accesses in subsequent iterations. Also, because of this there are less memory accesses and therefore there is a constant 35% decrease in instruction count even when the input size scales up. The overall increase in the cache hits is shown in Fig. 5.4.



(a) Polygonal partitions of the iteration space for Case 1 - $\mathcal{T}^6 = I$ (identity).



(b) RAR dependence in the loop-nest

Figure 5.2: Partitions of the iteration space in Case Study 1.

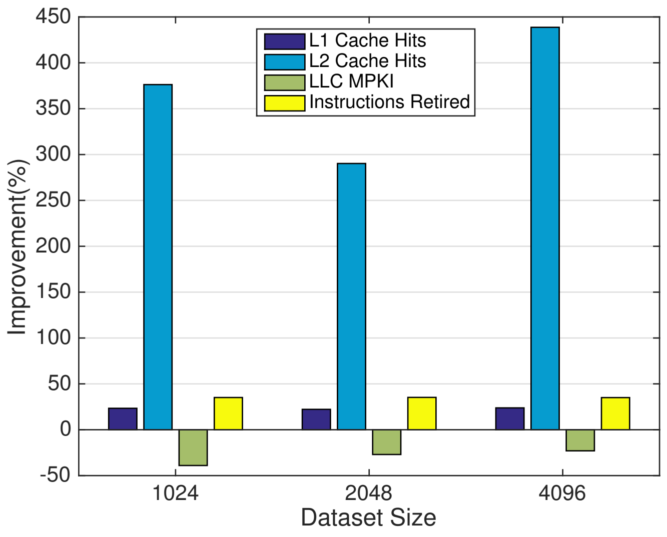


Figure 5.4: Case Study 1 - % of Improvement in L1 and L2 hits, LLC misses and Instructions Retired

The serial code optimized using the technique shows up to 1.19x speedup (Fig. 5.10a). For parallel execution, each type of partition is executed on a different thread using OpenMP® Sections so as to maximize locality on a core. On parallel execution across 8 threads, the speedup is even higher (up to 3.17x) due to the *selective mapping* of the partitions as shown in Fig. 5.11. This optimized code is compared against the tiled code from Polly and Pluto. The two tools generate rectangular tiles for the given program, since both of them do not use the information from RAR dependence to optimize code for locality, unlike the proposed technique. Experimental results show scalability of performance with the input size because even though the number of partitions remains constant, the size of the partitions scales with the input size.

5.3 Case Study 2: One Dimensional Non-Uniform Reuse Pattern

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

Figure 5.5: Loop-Nest with One Dimensional Non-Uniform Reuse

The references to array Y for this case, shown in Fig. 5.5, are as follows:

$$\Gamma_{i,\ j} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \mathbf{I} + \begin{pmatrix} 0 \\ 0 \end{pmatrix},\quad \Gamma_{i,\ i+j+N} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \mathbf{I} + \begin{pmatrix} 0 \\ N \end{pmatrix}$$

Therefore, the temporal reuse relation $\mathcal{T} = (T, t)$, assumes the following form, according to Equation 4.3:

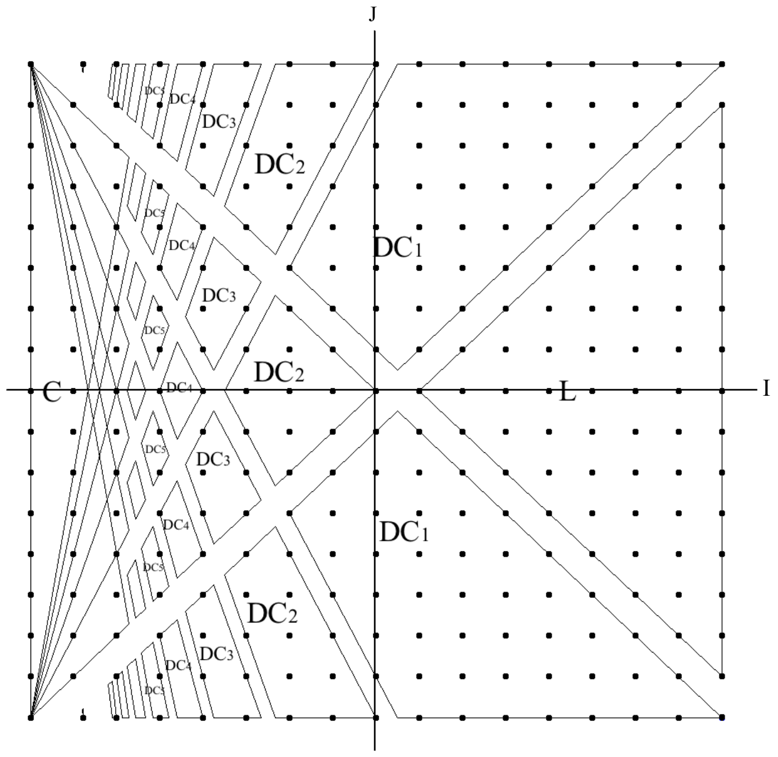
$$T = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix},\quad t = \begin{pmatrix} 0 \\ -N \end{pmatrix}$$

For this case, the maximum value of k is too high. It is dependent on the variable N which is a representation of the dataset size.

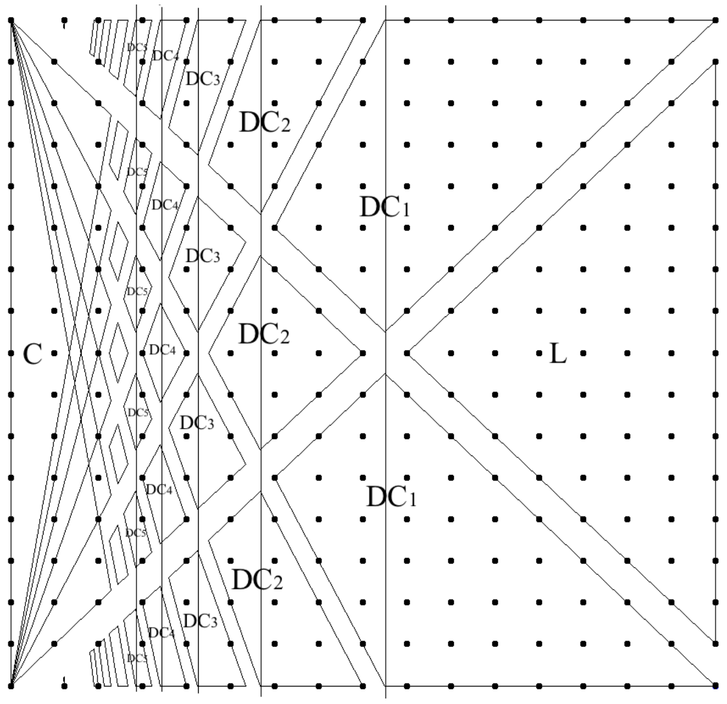
$$T^k = \begin{pmatrix} 1 & 0 \\ -k & 1 \end{pmatrix},\quad t = \begin{pmatrix} 0 \\ -kN \end{pmatrix}$$

Due to the fact that there is reuse only in the dimension J, refer to Figure 5.6a, the maximum value that k can reach is $2N - 1$. As the algorithm moves towards $-I$ direction, it forms smaller partitions. This leads to the drawback of the original algorithm. As described in the Section 4.3, an optimal value for k must be chosen such that program performance is maximized due to a better locality exploitation and the achievable speedup is not diminished by the control statement overhead.

Therefore, it is proposed that the partitioning algorithm must halt as soon as tile size reduces below 25×25 iterations. This is deduced from the experimental data as shown in



(a) Reducing partition size



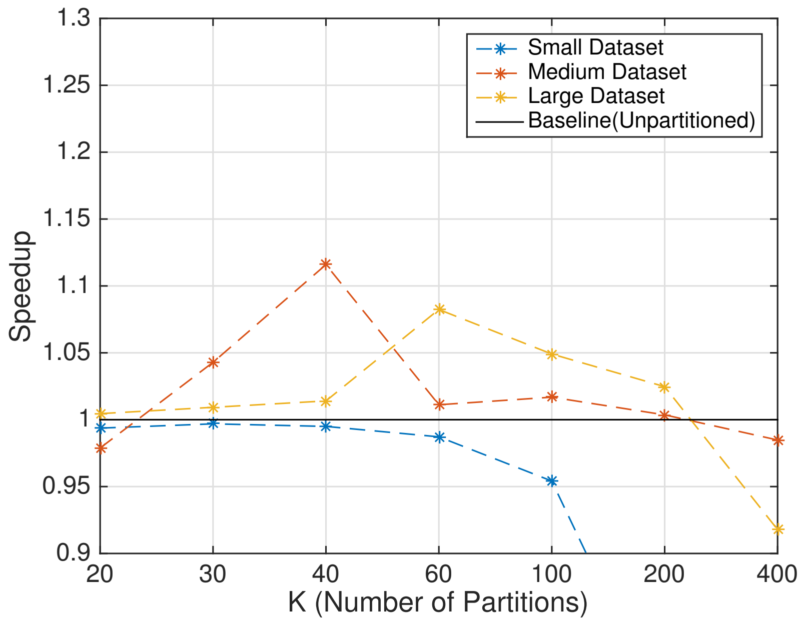
(b) Computational Wave-front

Figure 5.6: Partitions of the iteration space in Case Study 2.

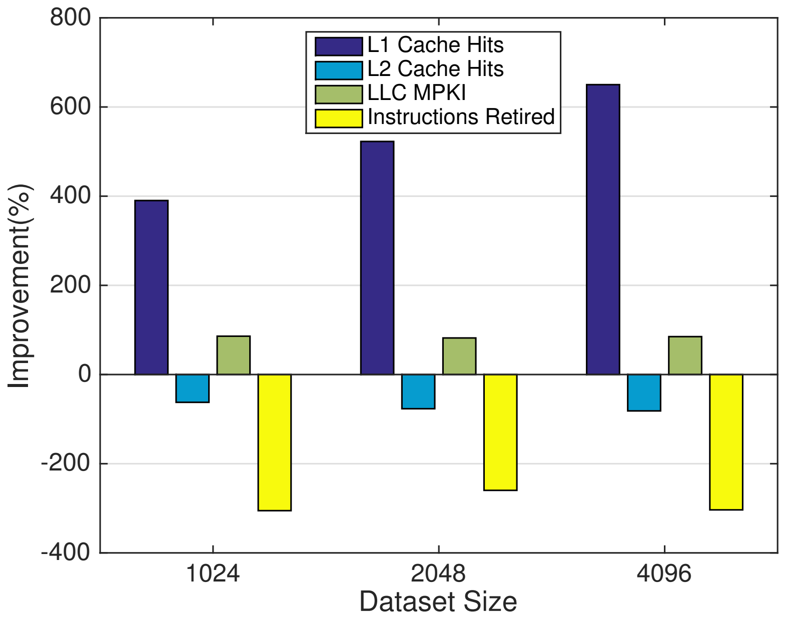
Figure 5.7a. The optimal value of k was found to be around 30 in a small dataset (N=1024), 40 in a medium dataset (N=2048), and 60 in a large dataset (N=4096). As shown in Figure 5.7a, the speedup is significantly related to the selection of k. If a value of k is chosen to be lower than the optimal value, the loop execution experiences a performance degradation due to low locality exploitation. On the other hand, if a value of k is chosen to be larger than the optimal value, the loop execution experiences a performance degradation due to control statement overhead.

Another important contribution to the achieved speedup comes from a code generation optimization which is discussed in Section 4.6. If partitions are executed similarly as in Case Study 1, the control statement overhead will inhibit achieve the maximum speedup achievable. By further splitting and executing them in a variable step wave-front (Fig. 5.6b) the control overhead is reduced - because the loop boundary conditions are simplified. This method does not conform to the originally proposed method of computing partitions of similar reuse pattern together inside single loop nest. This wave-front method execute different partition types together inside outer-most loop. It also improves spatial locality due to reuse on same cache-line for multiple partition-types. The increase in cache hits as shown in Fig. 5.7b is evident of improvement in locality.

Both of the Polyhedral Model based optimizers generate rectangular tiles in this case too because they ignore RAR dependence for improving locality. On serial execution, the maximum speedup of 1.13x is achieved for the medium dataset (Fig. 5.10a). Whereas, on parallel execution the speedup improves with the size of the dataset reaching maximum of 2.27x (Fig. 5.11).



(a) Speedup vs Number of Partitions.



(b) % of Improvement in L1 and L2 hits, LLC misses and Instructions Retired

Figure 5.7: Case Study 2: Optimal Number of Partitions and Improvement in Hardware Counters

5.4 Case Study 3 (Seidel-2D) and Case Study 4 (Jacobi-2D): Stencil Computations with Uniform Reuse Pattern and Multiple References

```
Partial loop-nest exposing reuse:
for (i = 1; i < N; i++) {
  for (j = 1; j< N; j++) {
  A[i][j]=(A[i-1][j-1]+A[i-1][j]+
           A[i-1][j+1]+A[i][j-1]+
           A[i][j]+A[i][j+1]+
           A[i+1][j-1]+A[i+1][j]+
           A[i+1][j+1])/9.0;
  }
}
```

(a) Seidel-2D

```
Partial loop-nest exposing reuse:
for (i = 1; i < N; i++) {
  for (j = 1; j< N; j++) {
    B[i][j]=(A[i][j]+A[i][j-1]+
             A[i][1+j]+A[1+i][j]+
             A[i-1][j])*0.2 ;
  }
}
```

(b) Jacobi-2D

Figure 5.8: Loop-Nest with Uniform Reuse Pattern and Multiple References

Case Study 3 and 4 are stencil benchmarks taken from the PolyBench[40]. Case Study 3 (Seidel stencil) from Fig. 5.8a has multiple references in 8 directions. Therefore, the heuristics mentioned in Section 4.4 must be applied to choose the best two references for creating partitions. The reuse vectors $\vec{v}_{i,j-1}$ and $\vec{v}_{i,j+1}$ link themselves by $\mathcal{T}^2$. Therefore, one of the reuse relations must be eliminated. The same applies to ($\vec{v}_{i-1,j-1}$, $\vec{v}_{i+1,j+1}$),($\vec{v}_{i-1,j}$, $\vec{v}_{i+1,j}$) and ($\vec{v}_{i+1,j-1}$, $\vec{v}_{i-1,j+1}$). Reference $\vec{v}_{i,j}$ must also be removed since its combination with any other $\vec{v}$ generates multiple equivalent partitions along $\vec{v}$. Therefore, references $\vec{v}_{i,j+1}$, $\vec{v}_{i+1,j+1}$, $\vec{v}_{i+1,j}$, $\vec{v}_{i+1,j-1}$ and $\vec{v}_{i,j}$ must be eliminated.

One drawback of the original algorithm is that some pairs of reuse vectors produces partitions which consume the entire iteration space, i.e. the two references ($\vec{v}_{i-1,j-1}$, $\vec{v}_{i,j-1}$) link every iteration in the domain. Therefore, this pair of references must be eliminated in addition to the pairs ($\vec{v}_{i-1,j}$, $\vec{v}_{i-1,j-1}$), ($\vec{v}_{i-1,j}$, $\vec{v}_{i-1,j+1}$) and ($\vec{v}_{i-1,j-1}$, $\vec{v}_{i-1,j+1}$). Finally, $\vec{v}_{i,j-1}$ and $\vec{v}_{i-1,j}$ are left and they create the partitioning as shown in Figure 5.9.

The two stencils show different performance results due to different amount of reuse among

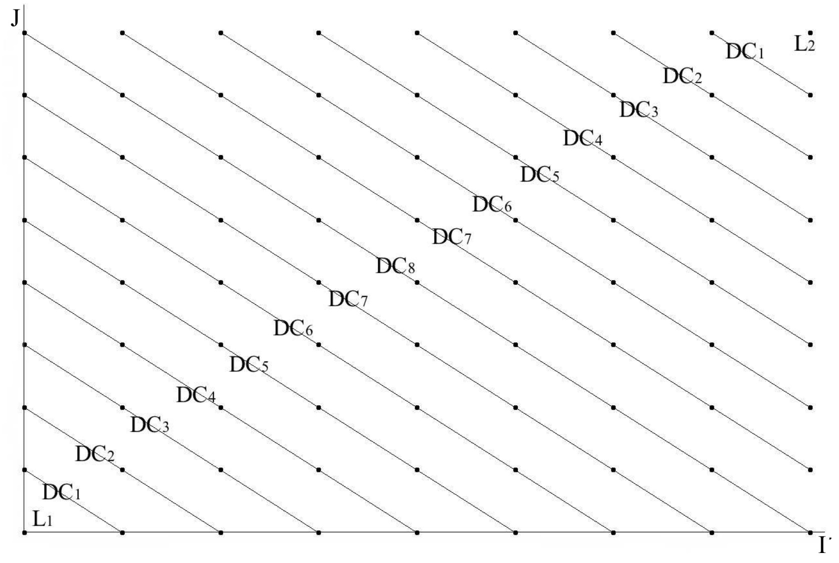


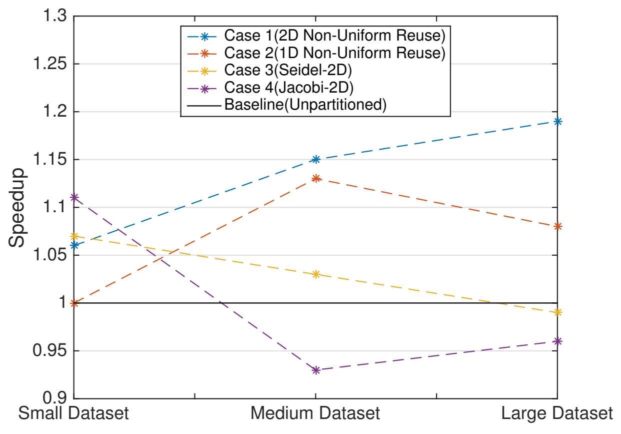
Figure 5.9: Partitions of the iteration space for Seidel and Jacobi stencils.

iterations in the partitions. In the case of Seidel-2D, there is more reuse between consecutive iterations inside a single sub-partition than Jacobi-2D, due to additional reuse on $\vec{v}_{i+1,j-1}$ and $\vec{v}_{i-1,j+1}$ in Seidel-2D.

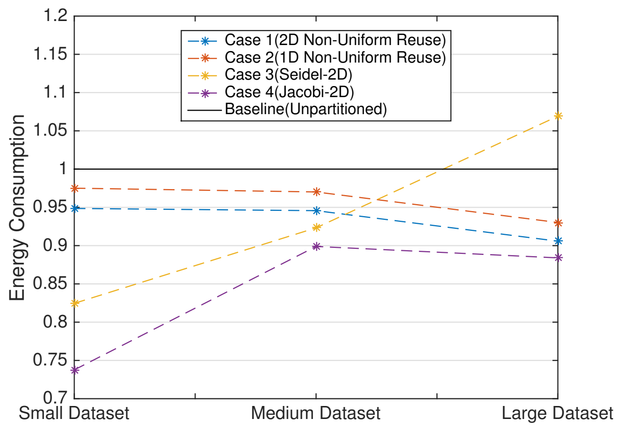
5.5 Speedup and Reduction in Energy Consumption

5.5.1 Serial Execution

The performance of the polygonally tiled code, compiled with LLVM (option: -O3 -fno-inline-functions), is compared against Polly - an optimizer for LLVM - optimized code (option: -O3 -polly -polly-vectorizer=stripmine -fno-inline-functions, tile size = 32×32). A significant amount of energy reduction (Fig. 5.10b) is noticed using counters from `Likwid` tool, which can be partially explained by the significant increase in L1 and L2 cache hits. This could provide advantage on battery powered devices, since this technique consumes less energy, with or without an improvement in performance.



(a) Speedup (Serial Execution)



(b) Energy Consumption Ratio

Figure 5.10: Performance and Energy consumption improvement.

5.5.2 Parallel Execution

The polygonally tiled code is compared for performance against the code optimized using Pluto-0.11.4 (options: –tile –parallel, tile size = 32 × 32) that generate OpenMP® code. Both codes are compiled with Intel's ICC-15.0.4 compiler (options: -O3 -xHost -ansi-alias -ipo -fp-model precise -fno-inline-functions) and are executed across 8 threads.

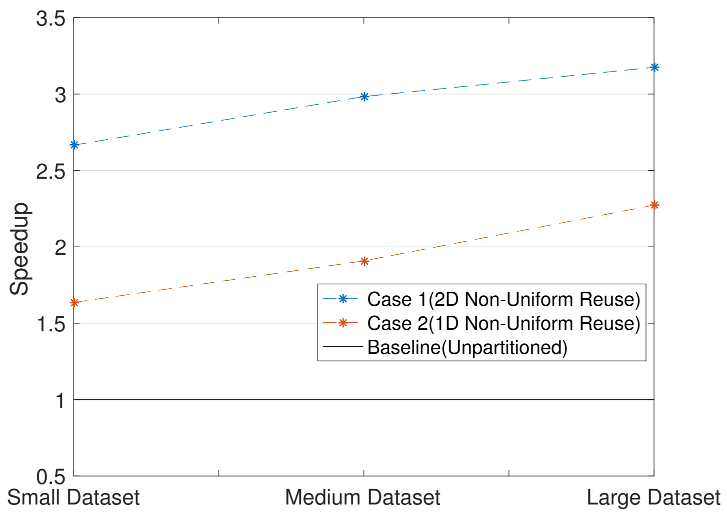


Figure 5.11: Speedup (Parallel Execution): Case Study 1 and 2

Stencils computations show poor result on parallelization using the proposed technique. This is due to the fact that Pluto generates rectangular and diamond shaped tiles (option: –part-diamond-tile) which provides better locality than the proposed technique. These tiles are better suited for stencil computations since the reuse patterns resembles the tile shapes which allows more reuse inside the tiles and less communication across the tiles. Also, diamond tiling[6] gives an extra advantage over rectangular tiling because of the concurrent start to more than one tiles.

Chapter 6

Conclusion

In this work, a polygonal tiling technique is presented, which in contrary to the current techniques, is not constrained to either the shape or the size of tiles that needs to be predetermined. Instead, the shapes and sizes are governed by the reuse pattern of the loop-nests. The goal of this technique is to speedup the execution of loop-nests with non-uniform reuse patterns. The proposed technique partitions the iteration space and schedules the partitions to ensure maximum reuse of data without being flushed out of the caches. The optimal number of partitions is determined as to speedup the performance. As the loops that are currently captured by the Polyhedral Model are compute intensive loops, such a speedup corresponds to a prominent reduction in the associated energy consumption.

Experiments on a set of loops exhibiting either non-uniform or uniform reuse patterns show that a significant portion of the achievable speedup is missed when applying traditional loop tiling to such loops. Speedup is significant for loops with non-uniform reuse pattern on serial execution as shown in the case studies. Benefits of the presented polygonal tiles is even greater for multi-threaded execution for such loops. High speedup (up to 3.17x) is achieved and it consistently improves on increasing the input size.

Bibliography

[1] A. Agarwal, D. Kranz, and V. Natarajan. Automatic Partitioning of Parallel Loops for Cache-Coherent Multiprocessors. In *Parallel Processing, 1993. ICPP 1993. International Conference on*, volume 1, pages 2–11, Aug 1993.

[2] A. Agarwal, D. A. Kranz, and V. Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(9):943–962, Sep 1995.

[3] N. Ahmed, N. Mateev, and K. Pingali. Tiling Imperfectly-nested Loop Nests. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 31–31, Nov 2000.

[4] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), Williamsburg, Virginia, USA, April 21-24, 1991*, pages 39–50, 1991.

[5] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[6] 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. IEEE Computer Society Press.

[7] U. K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[8] U. K. Banerjee. *Dependence Analysis.* Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[9] C. Bastoul. Efficient Code Generation for Automatic Parallelization and Optimization. In *Proceedings of the Second International Conference on Parallel and Distributed Computing*, ISPDC'03, pages 23–30, Washington, DC, USA, 2003. IEEE Computer Society.

[10] C. Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.

[11] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *International Conference on Compiler Construction (ETAPS CC)*, Apr. 2008.

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

[13] U. K. R. Bondhugula. *Effective automatic parallelization and locality optimization using the polyhedral model*. PhD thesis, The Ohio State University, 2008.

[14] L. Carter, J. Ferrante, and S. F. Hummel. Hierarchical tiling for improved superscalar performance. In *Parallel Processing Symposium, 1995. Proceedings., 9th International*, pages 239–245, Apr 1995.

[15] P. Clauss. Handling memory cache policy with integer points countings. In *Euro-Par'97 Parallel Processing: Third International Euro-Par Conference Passau, Germany, August 26–29, 1997 Proceedings*, pages 285–293, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[16] CLooG: The Chunky Loop Generator. `http://www.cloog.org`.

[17] P. Feautrier. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming*, 20, 1991.

[18] P. Feautrier. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International Journal of Parallel Programming*, 21(5):313–347, 1992.

[19] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[20] M. Griebl, C. Lengauer, and S. Wetzel. Code Generation in the Polytope Model. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, pages 106–, Washington, DC, USA, 1998. IEEE Computer Society.

[21] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. Hybrid Hexagonal/Classical Tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 66:66–66:75, New York, NY, USA, 2014. ACM.

[22] T. Grosser, A. Groesslinger, and C. Lengauer. Polly - Performing Polyhedral Optimizations on a Low-level Intermediate Representation. *Parallel Processing Letters*, 22(04), 2012.

[23] G. Gupta and S. Rajopadhye. The Z-polyhedral Model. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 237–248, New York, NY, USA, 2007. ACM.

[24] A. Hartono, M. M. Baskaran, J. Ramanujam, and P. Sadayappan. DynTile: Parametric tiled loop generation for parallel execution on multicore processors. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.

[25] 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. ACM.

[26] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann Publishers Inc., 2002.

[27] D. Kim, L. Renganarayanan, D. Rostron, S. Rajopadhye, and M. M. Strout. Multi-level tiling: M for the price of one. In *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–12, Nov 2007.

[28] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88, 2004.

[29] W. Li and K. Pingali. A Singular Loop Transformation Framework Based on Non-singular Matrices. *Int. J. Parallel Program.*, 22(2):183–205, Apr. 1994.

[30] A. W. Lim, G. I. Cheong, and M. S. Lam. An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 228–237, New York, NY, USA, 1999. ACM.

[31] A. W. Lim and M. S. Lam. Maximizing Parallelism and Minimizing Synchronization with Affine Partitions. *Parallel Comput.*, 24(3-4):445–475, May 1998.

[32] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and Array Contraction Across Arbitrarily Nested Loops Using Affine Partitioning. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 103–112, New York, NY, USA, 2001. ACM.

[33] B. Meister. Periodic Polyhedra. In *Compiler Construction, 13th International Conference CC 2004, Part of ETAPS 2004*, pages 134–149. Springer, 2004.

[34] B. Meister. Projecting Periodic Polyhedra for Loop Nest Analysis. In *CPC*, pages 13–24, 2004.

[35] B. Meister, V. Loechner, and P. Clauss. The Polytope Model for Optimizing Cache Locality. Technical report, Technical Report RR 00-03, ICPS-LSIIT, 2000.

[36] D. A. Padua and M. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Commun. ACM*, 29(12):1184–1201, 1986.

[37] PLUTO: An automatic parallelizer and locality optimizer for affine loop nests. `http://pluto-compiler.sourceforge.net`.

[38] Polly: LLVM Framework for High-Level Loop and Data-Locality Optimizations. `http://polly.llvm.org`.

[39] PolyLib: A library for manipulating parameterized polyhedra. `http://pluto-compiler.sourceforge.net`.

[40] L.-N. Pouchet. PolyBench/C 4.1. `http://web.cse.ohio-state.edu/~pouchet/software/polybench/`.

[41] W. Pugh and D. Wonnacott. Eliminating False Data Dependences Using the Omega Test. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 140–151, New York, NY, USA, 1992. ACM.

[42] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of Efficient Nested Loops from Polyhedra. *Int. J. Parallel Program.*, 28(5):469–498, Oct. 2000.

[43] P. Quinton, S. Rajopadhye, and T. Risset. On Manipulating Z-Polyhedra Using a Canonical Representation. *Parallel Processing Letters*, 07(02):181–194, 1997.

[44] J. Ramanujam. Non-unimodular Transformations of Nested Loops. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Supercomputing '92, pages 214–223, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[45] J. Ramanujam. Beyond Unimodular Transformations. *J. Supercomput.*, 9(4):365–389, Dec. 1995.

[46] L. Renganarayana and S. Rajopadhye. A Geometric Programming Framework for Optimal Multi-Level Tiling. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, pages 18–18, Nov 2004.

[47] L. Renganarayanan, D. Kim, S. Rajopadhye, and M. M. Strout. Parameterized Tiled Loops for Free. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 405–414, New York, NY, USA, 2007. ACM.

[48] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, Chichester, 1986.

[49] S. Verdoolaege. *isl: An Integer Set Library for the Polyhedral Model*, pages 299–302. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[50] M. J. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[51] J. Xue. *Loop Tiling for Parallelism.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.