Automatic Correctness Checking for Affine Constrained Loop Tiling Transformations

---

Russell Rivera

May 15, 2020
Version: Spring Final Draft
Advisor: Dave Wonnacott

Abstract

The growth in speed seen in modern processors over the past couple decades has created a need for programs that carefully manage the data fed to CPUs. Often, it is the case the processors are so fast, the memory used to supply them with information cannot keep up. Advanced iteration space transformations for stencil computations have found clever ways of parallelizing computationally heavy workloads while managing memory bandwidth through methods such as loop tiling, which streamline cache usage. Additionally, these methods offer scalable parallelism with respect to problem size. While automatic transformations for complex loop structures are still being researched, the same mathematical framework for checking the correctness of programmer implemented transformations should naturally enable such automation. This paper offers insight into the literature surrounding the Polyhedral Model, Iteration Space Transformations, and the mathematical background on which they are formed. Later, this paper seeks to motivate an investigation of whether non-symbolically-tiled, parameterized iteration space transformations, such as diamond tiling, can be verified for correctness with existing mathematics and programming techniques.

Acknowledgements

There are too many people to thank, with too little space to do so. I would like to thank my parents for their continuous support, regardless of the many (unwise) decisions I have made. Thank you to my advisors and teachers, who have helped me make the most of a last minute major switch, and to everyone who helped show me the beauty of mathematics.

Contents

1 Introduction

1.1 Modern Uses for Parallelism

The speed of modern processors compared to the bandwidth of fast access memory is a key driving factor of research in parallelizing computational workloads ([1], [3], [4], [9], [13]). Specifically, the sheer speed of modern processors is bottlenecked by the availability and performance of fast memory, such as cache. In this sense, increasing processor speed is less meaningful today than creating programs are able to make the most of memory bandwidth by splitting a single job over multiple processors. This trend is reflected in modern processor architecture: newer processors are shifting to favor multi-threaded and multi-core designs rather than the single-core ones.

For systems programmers, optimization over multiple processors requires a balancing of caring for memory bandwidth, while creating enough parallelism to outweigh the additional costs of constructing such a program. Additionally, there is another added layer of difficulty in ensuring the correctness of the conversion from a sequential to a parallel program. This paper reviews the challenges of converting a computationally expensive class of programs called Stencil Computations into equivalent parallelized programs.

Moving forward, this paper will first explain at a high level the steps involved with transforming a sequential Stencil Computation into something that can be run in parallel. It will dive into a much more detailed explanation of how the code of a stencil computation can be analyzed to produce affine (linear) constraints that will define whether or not the computation can be transformed. We then compare various methods in which these computations are transformed, while bringing several important contributing factors into focus: first, the significance of being able to maximize memory bandwidth with something called locality. Second, the ability to ensure the parallelism is not hindered by excessive computational overhead. Afterwards, we discuss the differences in philosophies of several of these approaches, and the relevance of making this kind of code-parallelism transformation an automatic feature in modern compilers.

At the end of this paper, we present plans to build a novel loop transformation correctness checker using existing infrastructure from Polly, an LLVM-IR to LLVM-IR optimization tool, and the ISL/ISCC calculator. We have determined such a program should be theoretically possible to build, and could provide a practical tool for researchers that need to confirm the correctness of transformations that may be possible to be optimized by an automatic compiler, but may not be worth the overhead, or are not able to be detected.

Literature Review

2

2.1 Framing the Problem

Briefly, it is worth understanding what we mean exactly when we refer to parallelized computing, and the goals of parallel computing. At first glance, a user may conclude that the benefit of parallelizing a workload is that all workloads of that size and complexity can be made faster and faster by breaking it into equivalent, parallelized components. There are two philosophies surrounding how to interpret parallelism.

Perspective one: Consider a problem that has a fixed sized input. In other words, there is some bound for which this input size will never grow beyond. To what extent does parallelism help solve the problem faster? Assuming that a given portion of the job can be parallelized, of course, we could execute this portion of the job in parallel. This way, the more processors we use, the faster our program can theoretically run. But this behavior can be measured asymptotically as well. Earlier we supposed that a *portion* of the job could be parallelized. This means regardless of the number of processors or cores we can use, the remaining un-parallelized portion of the job will not be affected. Thus, there is the sense that adding more processors results in diminishing returns, since after a certain point, we approach the amount of time required for only the un-parallelized portion of our workload. This does not even include the overhead of preprocessing required to produce a parallel program in the first place. From this perspective, parallelism seems to reach a dead end in terms of its usefulness.

Perspective two: Now consider an unfixed input size, but now there is a fixed amount of time that we have to run our computation. If we increase the number of processors we use as the size of the problem's input increases, does the rate at which we would need to add processors grow at a reasonable rate? In other words, if we have problem that requires 1,000 computations, and we run it over 100 cores, will we achieve a similar (or acceptable) runtime to a larger instance of that problem that requires 10,000 computations, but we run it on 1,000 cores?

These two perspectives are important, since they define the kind of problem researchers are trying to solve. The first perspective, known more formally as $Amdahl's\ Law$ offers a key insight that it is very unlikely that there are ways to

continuously decrease the work time of certain jobs [1]. However, $Gustavson's\ Law$, the second perspective, offers the insight that although the amount of work on a fixed job can be asymptotically bound, if the problem size is allowed to grow with the number of processors, there may be a quantifiable relationship that describes how the number of processors must increase with respect to time in order to keep runtimes of all jobs at some constant value [8]. In this paper, we look to explore parallelism through the perspective of Gustavson's Law; our questions will revolve around whether parallelism can scale with job size.

2.2 Stencil Computations, Dependence, and Imperfect Loop Nests

Stencil computations are a class of computationally expensive, iterative tasks that calculate new state values based off of other previously calculated values [2], [3], [4], [11], [14]. These kinds of computations are common in applications such as simulating the surface of an object interacting with heat or light. Most often, these computations use some sort of array or 2D array (a matrix) to store values [2][10]. During runtime, a stencil computation may read and write to a given location in that array multiple times, as well as update the value of a given entry in that array based off of the adjacent or neighboring entries. Models that attempt run stencil computations in parallel often abstract the sequential updating of values within the array as a "flow" of information. For example, a single array cell a_i may be calculated using values from preceding array cells a_{i-1} and a_{i-2}. In this sense, information "flows" from array cells a_{i-1} and a_{i-2} to array cell a_i. Specific dependencies will be defined more precisely in a following section. In the meantime, this section will illustrate how dependencies, specifically value flow dependencies, (denoted as δ_v^f), effect the overall outcome of a computation. The figure below shows two different stencil computations that result in two different values.

Figure 1

```
// Example of Single Assignment Loop Nest
for (t = 0; t < T; t++) {
   for (i = 1; i < N-1; i++) {
         A[i] = (A[i-1] + A[i] + A[i+1]) * (1.0/3);
    }
};
```

```
// Example of an Imperfect Loop Nest
for (t = 0; t < T; t++) {
   for (i = 1; i < N-1; i++) { // calculate new values
```

```
            new[i] = (A[i-1] + A[i] + A[i+1]) * (1.0/3);
        }
       // update A with new values from above
       for (i = 1; i < N-1; i++) {
            A[i] = new[i]; //Note the new A[i] is dependent on A[i-1], A[i],
                A[i+1]
        }
};
```

We refer to the statement responsible for updating the value stored at an array cell as the loop body. Notice in the code above that the loop bodies for both pieces of code are very similar. The first piece of code uses singular assignment in the loop body. This means that updated array values are completed in a single assignment. We refer to the kind of implementation shown in example 2 as an imperfect loop nest ([2], [3], [10], [11]). In example 2, the given code updates all indices from 1 to $N-1$, for every iteration t, for $0 \leq t < T$. We can think of iterations of t as being a "timestep". In other words, we can think of a given iteration t as being a given point in time, when the array has a collective state determined by all values of the array cell $A[i]$ of $1 < i < N-1$.

Importantly, this distinction means that an entry is only updated with values that are from a previous state, e.g., an entry for state t is only produced with values from the $t-1$ state. This is particularly important since we are accessing $A[i+1]$ when computing the $A[i]$ term. For example, consider example 1 in the code above. Let $A[i]$ be the ith entry in array A. Suppose that we are at some iteration t. In order to produce the values for the $t+1$ iteration, we would expect to only use the values from the previous iteration t. However, if we use the scheme from the first example, we are updating the row of A as we go, which means after we have computed $A[i]$, and move to update the new value of $A[i+1]$, we access $A[i]$ again, since $A[i] = A[(i+1)-1]$. But the value of $A[i]$ no longer represents the value of $A[i]$ at iteration t, it represents the value of $A[i]$ at iteration $t+1$. This does not preserve the computational ordering the same way that example two does. Since the code in example two writes its values to a separate array, then overwrites all previous values in array A, it maintains that the values in the $t+1th$ iteration are only calculated with the values from the tth iteration, not from a mix of both. The use of imperfect loop nests is specifically to handle the case when a computation at iteration for a given index i is dependent on another index or indices j, where $i < j \leq n$, where n is the upper bound on the number of iterations. When there is not such a "forward access", loops can be "fused" to the benefit of parallelism and simplicity:

Figure 2:

```
// Imperfect Loop Nest
for (int t = 0; t < 3; t++) {
   for (int i = 0; i < 3; i++) {
        new[i] = A[i] + A[i-1];
    }
   for (int n = 0; n < 3; n++) {
        A[n] = new[n];
    }
}

// Perfect Loop Nest (through Fusion Transformation - equivalent)
for (int t = 0; t < 3; t++) {
   for (int i = 0; i < 3; i++) {
        A[t][i] = A[t-1][i+1] + A[t-1][i-1];
    }
}
```

Perfect and imperfect loop nests show the importance of ordered operations in stencil programs. While there are some cases where imperfect nests can be transformed into perfect ones, in other cases, the array accesses of our computation restrict how we can iterate through the loop. This distinction introduces several underlying themes in parallel computing. First, we have the idea of how we iterate through a space. Second, we get a sense of kinds of dependencies between values that are read from an array, and values that are written to an array - we call these relationships dependencies. A good way to think about dependencies is to think about the relationship between $A[i-1]$ and $A[i]$ in the previous example, and consider the relationship between the two values. The value of $A[i]$ is dependent on $A[i-1]$; we need to make sure that if we parallelize this loop, $A[i]$ still has access to the value at $A[i-1]$, and $A[i-1]$ is calculated before $A[i]$. In the next section, we look at how we can start to formalize these relationships between values, and use them to produce something that can be parallelized.

2.3 Introduction to the Polyhedral Model

Finding a way to parallelize stencil computations requires that either a programmer (or a compiler) uphold the dependencies between values during a program's execution. In other words, as we discussed in the previous section, we must preserve the order of certain sub-calculations, in order to make our algorithm not only fast, but also correct. We can also think of correctly 'ordered' calculations as being 'forward in time'. Just as we defined a computation in the previous section that only used

values from the previous iteration, we need to find a way of defining and preserving the order for these kinds of subproblems.

The Polyhedral Model serves as a basis for many complex calculation orderings and reorderings, as well as code generation that can accurately describe and preserve data dependencies within stencil computations ([2-4], [7], [10], [13], [14]). The root of this model is based in both linear integer programming, and linear algebra. In this section, we describe several powerful abstractions that allow us to define a powerful mathematical model known as the Polyhedral Model. These sub-abstractions include iterations space of a program, dependence relations for computations, such write accesses and read accesses, and space tiling. As we begin to layer these abstractions, our analysis will become increasingly more technical.

One of the most critical pieces to understanding the Polyhedral Model is the concept of iteration space. At a high level, iteration space is an abstraction of loop variable bounds that defines a range of computations that a program may perform. For example, a program with a single for loop that iterates from zero to n is said to have a one-dimensional iteration space with over values 0 to n. There are several useful mathematical abstractions to this. Micheal Wolfe describes iteration space as the finite Cartesian space with dimension equal to the number of loops nested in a program ([7], [8], [12]). For example, at any given iteration of the program below, we can represent any iteration as the set of ordered integer pairs (i, j) such that $1 \leq i \leq I$ and $1 \leq j \leq J$.

Figure 3: Example Credit: [4],[8]

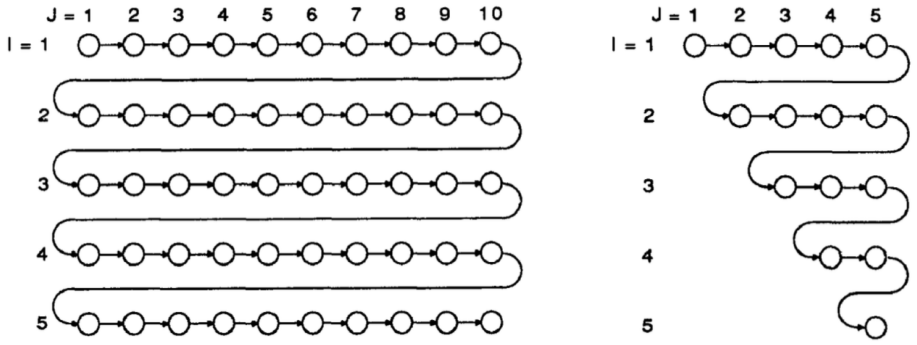
```
for (int i = 1; i <= I; i++) {
   for (int j = 1; j <= J; j++) {
        M[i][j] = M[i-1][j] + M[i][j-1];
    }
}
```

This is a powerful abstraction, as we can project how a program iterates onto the familiar space of a Cartesian plane. Just as we can represent a given iteration of a program with a nested pair of loops as a point on the Cartesian plane, we may represent any given iteration of a program with n nested loops as a point in n-dimensional space. Notably, since the values of a loop are confined to the integers and concerns a specific range of integers, we can think of the iteration space of a program with n loops as some subset of ^{n}.

The introduction of n-dimensional space leads to a very clean application of linear algebra and vector spaces ([1], [4], [8], [13], [14]). Just as we represented

individual iterations of a program as points in space, we can also think of these iteration instances as vectors relative to some initial basis. The iteration space of a nest of n loops can be represented as the set of all iteration vectors $\vec{v}$ that can be described by a linear combination of basis vectors $\{\vec{b_1}...\vec{b_n}\}$ and integer weights $c_1...c_n$. Thus, each iteration of the program above can be described as an iteration vector $\vec{j}$, where the components of $\vec{j}$ are its coordinates relative to a valid basis of the iteration space, and all components of $\vec{j}$ are bound by the largest integer that a loop accesses during iteration ([2],[5],[10],[14],[19]). While often it is intuitive to think about points in the iteration space relative to the standard basis (e.g., the x and y axes), the iteration space is not necessarily rectangular, meaning that we cannot assume that the iteration space expands equally in all dimensions. Additionally, iteration space may be expressed relative to any basis that spans the usable space, not just the standard basis.

Figure 4: Differently Shaped Iteration Spaces



The figure above shows two separate two-dimensional iteration spaces, a rectangular one on the left, and a triangular one on the right. In addition to different iteration space shapes, the lexicographic order of iteration variables within a loop statement is also important for determining how a loop will iterate through its iteration space. In other words, the lexicographical ordering of loop variables, (in the above example i, and j) can be swapped to have the loops iterate through the 2D array column by column rather than row by row. We call the first way, iterating through columns, Column Major Order, and the latter, iterating through rows, Row Major Order. The example code below shows a modified version of figure three, showing lexicographical change in the code.

```
// note the statement within the loop nest was originally:
// M[i][j] = M[j-1][i] + M[j][i-1];
for (int i = 1; i <= I; i++) {
   for (int j = 1; j <= J; j++) {
        M[j][i] = M[j-1][i] + M[j][i-1];
    }
```

```
}
```

Such changes in code can be subtle, but offer flexibility for changing how a program may iterate over an altered iteration space.

Using the iteration space as an abstraction for what is happening at the program level, we can begin to define a more rigorous mathematical framework. Specifically, we can abstract further from the idea of points in space, and consider a set of vectors in some 2D space. We can define the iteration space as the set of all vectors that satisfy the equation $A\vec{j} \leq \vec{\alpha}$, where A is a matrix describing access relations of the statement nested within all of the loops, and $\vec{\alpha}$ is a vector representing symbolic constraints of the loop (in the case of the above example, I, J) [8].

$$A\vec{j} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \vec{j} \leq \begin{bmatrix} I \\ J \\ -1 \\ -1 \end{bmatrix} \tag{2.1}$$

Explicit multiplication confirms that our constraints are accurately represented:

$$A\vec{j} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \leq \begin{bmatrix} I \\ J \\ -1 \\ -1 \end{bmatrix} \tag{2.2}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} i \\ j \\ -i \\ -j \end{bmatrix} \leq \begin{bmatrix} I \\ J \\ -1 \\ -1 \end{bmatrix} \tag{2.3}$$

$$\begin{bmatrix} i \\ j \\ -i \\ -j \end{bmatrix} \leq \begin{bmatrix} I \\ J \\ -1 \\ -1 \end{bmatrix} \tag{2.4}$$

Combining these inequalities, we can confirm that this equation accurately describes values of i and j that are used in our loop during computation in the form of a solution vector $\vec{j}$ to the equation $A\vec{j} \leq \vec{\alpha}$:

The takeaway from all of this math is simply to show that our abstractions from the range of variables used in loops to high-dimensional spaces in ^{n} actually hold water. Both the coordinate interpretation of iteration space, and the vector interpretation of iteration space provide a fantastic framework for accurate and reproducible modeling. In the next section, we explore how we can take advantage of this mathematical framework of iteration and computational space to define how variables are accessed in stencil computations. We will then apply mathematical abstraction in order to define data dependencies, similar to the ones described in the previous section, in terms of affine and linear constraints.

2.4 Polyhedra, Describing Dependencies More Precisely

In this section, we elaborate on the kinds of dependencies that often exist between values in a stencil computation. Throughout the remainder of the paper, we will primarily be focused on a type of dependency called a value-based flow dependency. Another dependency that we will describe, a memory-based flow dependency, is also useful in formalizing dependencies between arrays. Combining these definitions with the framework of the Polyhedral Model, we can explicitly map how information is passed from one vector in the iteration space of a program to another. Consider the code below:

```
for (int i = 1; i < N - 1; i++) {
     for (int j = 1; i < N - 1; j++) { }
}
```

For each pair of possible iteration variables i and j, we can create a one to one correspondence between the tuple representing a given iteration where, and an iteration vector in $Z^2 \times Z^2$. For example, an arbitrary iteration in the above program represented by the tuple (i, j), where $1 \leq i, j < N - 1$, also maps to the iteration vector $\vec{q} = \begin{bmatrix} i \\ j \end{bmatrix}$. This abstraction applies to more general cases as well. Given a program with n nested loops, we can produce a unique n-tuple containing the values of each of the iteration variables used within the loop nest. Such a tuple can then be mapped by a one to one correspondence to an iteration vector of composed of

values of the iteration vector. Using this abstraction, we can symbolically represent an access to an array $A[i][j]$ as the iteration tuple (i, j) or as the iteration vector $\vec{q}$.

In the Polyhedral Model, a value-based flow dependency, denoted δ_v^f, symbolically describes a relation between two iterations that take place in a loop. Formally, we may say that there exists a value flow dependence from $\vec{u}$ to $\vec{v}$ if vector $\vec{v}$ requires a value produced at $\vec{u}$. In terms of array values, we can think of this as occurring when one array cell requires a value from another array cell in order to compute its next value ([1-4], [10], [13], [14]). In other words, a value flow dependence from $\vec{u}$ to $\vec{v}$ shows the transfer of information from a write at $\vec{u}$ to a read at $\vec{v}$. This kind of description of a value flow dependency is referred to as a "current flow" style of dependence, since the arrow points from data source to sink. Confusingly, the convention also exists to describe flow of information from sink to source. These kinds of descriptions are referred to as "weathervane" flow. These two are often conflated or not explicitly defined in papers - from this point forward, when value based flow dependencies are described, we will use the "current flow" convention.

A value-based anti-flow dependence, denoted δ_v^-, describes a relation between two iterations such that both access the same memory cell, and one overwrites a value the other needed. Formally, an iteration $\vec{u}$ is said to be anti-dependent to an iteration vector $\vec{v}$ if $\vec{v}$ overwrites a value that $\vec{u}$ needs during the course of computation. Consequently, such dependencies are relations from a read-access ($\vec{u}$) to a write-access ($\vec{v}$).

A value-based output dependency, denoted δ_v^o, describes when one iteration overwrites the value that another iteration vector has written. Given an iteration vector $\vec{u}$, $\vec{u}$ is dependent on $\vec{v}$ with respect to output value if $\vec{u}$ and $\vec{v}$ access the same memory cell, and $\vec{v}$ overwrites the value that $\vec{u}$ wrote. Such a relation is defined from a write-access to another write-access.

For each value based dependency, we can also describe a corresponding memory-based dependency. These dependencies describe relations between two iteration vectors with respect to the memory cells that store their values, rather than describing the flow of the values themselves. While the difference is subtle, these dependencies are also important for program correctness checking.

A memory-based flow dependency (δ_m^f) exists from vector $\vec{u}$ to vector $\vec{v}$ if vector $\vec{v}$ uses a memory cell that might contain a value that was written by vector $\vec{u}$.

A memory-based anti-dependency (δ_m^-) exists from a vector $\vec{u}$ to vector $\vec{v}$ if vector $\vec{v}$ overwrites a memory cell that $\vec{u}$ might need.

A memory-based output dependency (δ^o_m) exists from a vector $\vec{u}$ to vector $\vec{v}$ if vector $\vec{v}$ overwrites a memory cell containing a value that $\vec{u}$ might have written.

Taking these definitions, we now provide a brief example of each of these dependencies on single variables. These dependencies are conceptually the same, however instead of relating iteration vectors to other vectors, the dependencies simply exist between two variables. For example, in a value-based flow dependence, such a relation will be defined from one variable x to another y, if y requires a value produced or possessed by x. Consider the following C code:

```
[1] int i = 1;
[2] int j = 2;
[3] int k = j;
[4] int l = k + i;
[5] j = j + 5;
[6] int m = j + 1;
[7] j = m + j;
```

Consider the variable i in line $[1]$ of the code above. When looking for a value flow dependence, we are looking to find a pair of variables, in which one requires the value that the other has. In line $[1]$, the variable i is defined and declared. The variable i is not reused until line $[4]$, where it is used to compute the integer variable l. In this case, there is a value dependency from the variable i used in line $[1]$ to the i used in line $[4]$. The Polyhedral Model takes this principle and applies it to specific array cells (or iteration vectors or iteration tuples) rather than singular variables. If we change the notation by which we define each row of this example program, we can more clearly define value flow relations between them.

```
[set_i] int i = 1;
[set_j] int j = 2;
[set_k] int k = j;
[set_l] int l = k + i;
[set_j1] j = j + 5;
[set_m] int m = j + 1;
[set_j2] j = m + j;
```

We can now define the value flow dependence previously described as between lines $[1]$ and $[4]$ as $\delta^{fv}_1 : [i] \rightarrow \{set\_i[i] \rightarrow set\_l[i]\}$, where the leftmost $[i]$ is a generic descriptor representing the variable involved in value flow. The relation is defined by the expression inside the braces. Specifically, this relation describes a mapping from the variable i in the set_i statement to the variable i in the set_l statement. Again, such a relation exists since the value i used in the set_l statement requires

the value of i in the set_i statement. For the sake of keeping the upcoming example concise, we will only introduce other dependencies as needed. We can define:

(1) $\delta_1^{fv} : [j] \rightarrow \{set\_j[j] \rightarrow set\_k[j]\}$

(2) $\delta_2^{fv} : [j] \rightarrow \{set\_j[j] \rightarrow set\_j1[j]\}$

Next, we look at some value-based anti-dependencies. Specifically, two examples are:

(3) $\delta_1^{-v} : [j] \rightarrow \{set\_k[j] \rightarrow set\_j1[j]\}$

(4) $\delta_2^{-v} : [j] \rightarrow \{set\_j1[j] \rightarrow set\_j1[j]\}$

Note in the last anti-value dependence the j on the left side of the equals sign requires the value from the j on the right side. The last value based dependencies we show are two value-based output dependencies:

(5) $\delta_1^{vo} : [j] \rightarrow \{set\_j[j] \rightarrow set\_j1[j]\}$ - note since this is an output dependence it maps from the j in set_j to the j on the left side of the equals sign in set_j1, as output dependencies are defined from a write to another write. (6) $\delta_2^{vo} : [j] \rightarrow \{set\_j1[j] \rightarrow set\_j2[j]\}$ - likewise, both j's in this dependence refer to j's on the left side of the equals sign, as an output dependency is between two writes.

Now consider some of the memory based dependencies. We can map a memory-based flow dependency from the j used in set_j to the j used in set_m, since the j at set_m needs a memory cell that might have the value of j used at set_j :

(7) $\delta_1^{fm} : [j] \rightarrow \{set\_j[j] \rightarrow set\_m[j]\}$

Next, we can define a memory-based anti-dependency from the j on the right side of the equals sign in set_j1 to the j on the right side of the equals of set_j2, since this maps a read to a write, where the write overwrites a memory cell may contain the value of the read.

(8) $\delta_1^{-m} : [j] \rightarrow \{set\_j1[j] \rightarrow set\_j2[j]\}$

Lastly, we can define a memory-based output dependency from the j used in set_j to the j used in set_j2, since $set\_j2[j]$ overwrites a memory cell that might contain the value of j at set_j :

(9) $\delta_1^{mo} : [j] \rightarrow \{set\_j[j] \rightarrow set\_j2[j]\}$

As we will see, all memory dependencies can be described as a combination of two or more value-based dependencies. Such transitivity allows for the framework of the Polyhedral Model to become incredibly powerful, as value-based dependencies can be checked against memory-based ones and visa versa. This is because dependencies provide necessary conditions for when certain operations can be reordered. If two independent operations (operations without dependencies) are reordered at a given iteration, the solution will not be changed. Importantly, this framework of dependencies that is rigorously defined by the polyhedral model will allow for special "transformations" on iteration space, that allow for increased parallelism. For now, we show how a pair of value-based dependencies form a memory-based dependency, although we omit such proof in place of a reference [19].

Consider three value dependence relations: the value-flow dependence (1), the value-based anti-dependency (3), and the value-based output dependency represented in (6). We expect that the composition of these relations would produce a corresponding memory flow relation as shown in (9). Notice how this shows how information is "flowing" from one use of the variable j to another.

$$\delta_1^{fv}(\delta_1^{-v}(\delta_2^{vo})) = \delta_1^{mo}$$
$$\delta_1^{-v}(\delta_1^{vo})) = \{set\_k[j] \rightarrow \{set\_j1[j] \rightarrow set\_j2[j]\}\}$$
$$= \{set\_k[j] \rightarrow set\_j2[j]\}$$

$$\delta_1^{fv}(\delta_1^{-v}(\delta_2^{vo})) = \{set\_j[j] \rightarrow \{set\_k[j] \rightarrow set\_j2[j]\}\}$$
$$= \{set\_j[j] \rightarrow set\_j2[j]\}$$
$$= \delta_1^{mo} : \{set\_j[j] \rightarrow set\_j2[j]\}$$

While the proof is omitted, this example shows how composing two or more value-based dependencies can determine where memory dependencies in a program, which is useful for a code-generator, such as a human or compiler, to determine whether changing the order of array accesses (by reordering the way a loop iterates through an array) may change the final outcome of the program. At a high level, all computations done with these relations show how information "flows" from the top of the program described above. In the next section, we see how the definition of various dependence relations dive further into the mathematical abstraction that describes the Polyhedral Model, and how dependencies are interpreted using vectors instead of relations of tuples.

2.5 Applying the Polyhedral Model to Arrays and Loop Nests

Each of the dependencies above formalize the relationship of memory reads and writes through a set of linear constraints that form a well-defined polyhedron in a program's iteration space, hence the name ([2],[4],[5],[10],[13]). In this section, we describe the mathematical formalism behind modeling value-based flow dependencies using an example program. While we exclude other dependency modeling, the following method applies to all dependence types.

Consider again our two dimensional loop nest from the previous example:

Figure 5: Example credit [8]

```
// where T is a two dimensional array (matrix)
for (int i = 1; i <= I; i++) {
   for (int j = 1; j <= J; j++) {
        S1: M[i][j] = M[i-1][j] + M[i][j-1];
    }
}
```

Take note of the statement in the loop body labeled $S1$ in the code above. Depending on the kind of stencil computation being used, there may be various statements in the loop bodies of a program. Since there can be multiple statements per nest, dependencies can exist between array accesses in a single statement (intra statement dependencies), there can also exist unique dependences between accesses in one statement and accesses in another (inter-statement dependencies). In this case, we are only concerned with intra statement dependencies, as there is only one statement in the loop body.

When describing the value-based flow dependence, we want to describe when a write ($M[i][j]$), and a read ($M[i-1][j]$ or $M[i][j-1]$), access the same memory cell, i.e., express an explicit relationship between when $M[i-1][j]$ and $M[i][j-1]$ read a value was previously written by $M[i][j]$. In other words, our value-flow dependence will describe a relation from a write $M[i][j]$ to the reads $M[i-1][j]$ and $M[i][j-1]$, depicting the flow of information. Examining Statement $S1$, we can create a pair of equations using our loop variables i, j in order to describe a moment in the program's execution that a given memory cell $M[i][j]$ is accessed.

Such dependencies can be described in vector form ([2], [4], [14]). Consider the flow dependence between the write access $M[i][j]$ and read $M[i-1][j]$. The information

flow represented can be thought of as a relation between the two iteration vectors: $R : \begin{bmatrix} i-1 \\ j \end{bmatrix} \rightarrow \begin{bmatrix} i \\ j \end{bmatrix}$ for some $1 \leq i \leq I$ and $1 \leq j \leq J$. We can express a dependence vector $v_{\delta 1}$ between these two iteration vectors, as a vector pointing from the respective write to respective read. One such vector is $v_{\delta 1} = \begin{bmatrix} i \\ j \end{bmatrix} - \begin{bmatrix} i-1 \\ j \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$. This abstraction is best interpreted as a flow of information. The equation:

$$\begin{bmatrix} i-1 \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} i \\ j \end{bmatrix} \tag{2.5}$$

describes how the information created at by $M[i][j]$ is used by $M[i-1][j]$; the vector $v_{\delta 1}$ simply abstracts this information flow to vectors in iteration space. This interpretation is much more useful for checking when two iteration vectors in the polyhedron are dependent or not, as we will see shortly.

If we combine our description of iteration space with the value flow dependencies we have just computed, the resulting set of linear inequalities will produce a polyhedron in ${}^2\times^2$. For clarity in the rest of the example, we differentiate the different values of i and j used on either sides of the equals sign in statement $S1$. Given an iteration vector for which the array write $M[i][j]$ is reached, we use the information from the iteration vectors with corresponding array cells $M[i-1][j]$ and $M[i][j-1]$. However, i and j are not the same for $M[i][j]$ as in $M[i-1][j]$ and $M[i][j-1]$. We rewrite the statement:

M[i][j] = M[i-1][j] + M[i][j-1]; $\rightarrow$ M[i'][j'] = M[i][j'] + M[i'][j];

to emphasize this difference. This reformulation redescribed the relation R between iteration vectors above as $R' : \begin{bmatrix} i \\ j \end{bmatrix} \rightarrow \begin{bmatrix} i' \\ j \end{bmatrix}$. This modification allows us to more easily express the relation with different variables $i' = i+1,\ j' = j+1$. For our dependence polyhedron, we can rewrite such constraints on the values of i and i', and j and j' to be a pair of inequalities:

$$i' = i+1 \rightarrow i' - i = 1 \rightarrow (i' - i \leq 1) \wedge (-i' + i \leq -1)$$
$$j' = j+1 \rightarrow j' - j = 1 \rightarrow (j' - j \leq 1) \wedge (-i' + i \leq -1)$$

This conversion helps the matrix form equation of the polyhedron keep the from $A\vec{x} \leq b$, just as we had when we described the computation space as $A\vec{j} \leq \vec{\alpha}$. We define the flow dependence polyhedron of statement $S1$ in the program shown in

Figure 6 as $\mathcal{P}_\delta$, as shown below: Note, the values in the matrix below the partition line are the constraints added to our definition of computation space:

$$A\vec{j} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \leq \begin{bmatrix} I \\ J \\ -1 \\ -1 \end{bmatrix} \tag{2.6}$$

$$\mathcal{P}_\delta = \left[\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ \hline 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \\ \hline 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 \end{array}\right] \begin{bmatrix} i \\ i' \\ j \\ j' \end{bmatrix} \leq \left[\begin{array}{c} I \\ J \\ -1 \\ -1 \\ \hline I \\ J \\ -1 \\ -1 \\ \hline 1 \\ 1 \\ -1 \\ -1 \end{array}\right] \tag{2.7}$$

Having defined the dependence polyhedron, we can now multiply out our equation to see how the program has manifested in this matrix representation.

$$\begin{cases} i \leq I \\ j \leq J \\ 1 \leq i \\ \underline{1 \leq j} \\ i' \leq I \\ j' \leq J \\ 1 \leq i' \\ \underline{1 \leq j'} \\ i - i' \leq 1 \\ j - j \leq 1 \\ i' - i \leq -1 \\ j' - j \leq -1 \end{cases} \rightarrow \begin{cases} 1 \leq i \leq I \\ \underline{1 \leq j \leq J} \\ 1 \leq i' \leq I \\ \underline{1 \leq j' \leq J} \\ i' = i + 1 \\ j' = j + 1 \end{cases} \tag{2.8}$$

Looking back at our source code in **Figure 5**, we can confirm that these bounds match the iteration bounds as well as the dependencies inside statement $S1$. Using the constructed polyhedron, we can now define iteration vectors within the polyhedron as sources and sinks of information flow. In other words, just as we showed $\begin{bmatrix} i-1 \\ j \end{bmatrix}$ and $\begin{bmatrix} i \\ j \end{bmatrix}$ were dependent previously, and information flowed from $\begin{bmatrix} i-1 \\ j \end{bmatrix}$ to $\begin{bmatrix} i \\ j \end{bmatrix}$ with respect to a value flow, we can describe a set of vectors, on source $\vec{s}$ and one sink $\vec{t}$ as vectors in the polyhedron if there exists a dependence vector describing the information flow between them, and $\vec{s}$ satisfies $A\vec{s} \leq \vec{\alpha}$ and $\vec{t}$ also satisfies $A\vec{t} \leq \vec{\alpha}$. In other words, if there exists a dependence vector between the two, and both are legal vectors within the computation space ([1]).

2.6 Iteration Space Transformations and the Polyhedral Model

The result of all the mathematical framework described in the polyhedral model is a general, but accurate representation of "instance-wise" dependencies between different reads and writes in a stencil computation ([2], [3], [4], [14], [18]). Here, instance wise refers to a given iteration within the iteration space of a program. In other words, we can use the polyhedral model as a reference for the dependencies of an arbitrary read and write at an arbitrary iteration within a computation space. This is extremely powerful, as we can use this model as a reference to identify whether an iteration space transformation satisfies the same constraints as the original program. This creates not only a framework for ensuring correctness, but a framework for program modification. There are a number of libraries and tools, such as the Omega Project, ISCC, and ClooG that can translate transformations on dependence polyhedra back into executable code ([4], [5], [14]). Moving forward, this section will provide an overview of iteration space transformations and an example of code generation before diving into more advanced literature on automatic dependency analysis.

The following section introduces a significant amount of linear algebra that may obscure the intuition of for readers without experience in the dark arts. In addition to showing the mathematical intuition behind iteration space transformations, it is important to recognize the goal of what all of this math is trying to achieve. The goal of describing dependencies in the polyhedral model is to enable us to define data dependencies and bounds on the iteration space of a program. Using this information, we can attempt to rearrange the iteration space in ways that enable parallel execution. These transformations must respect the constraints and

dependencies established in the polyhedral model, otherwise a new program based off of the new, transformed iteration space may produce a different answer from the original program.

It is also important to remember what points in iteration space actually represent. A point or dynamic instance in the iteration space of a program (i, j) also represented by an iteration vector $\vec{j}$ with components i, j, can be thought of as both a given point in time of a program's execution, i.e., when the loop variables are i and j $1 \leq i \leq I \wedge 1 \leq j \leq J$, OR as a data access of a 2D array M, at the index $M[i][j]$. We can abstract an iteration space transformation as a mapping that describes when a specific data accesses occur in a parallelized program as compared to the original one. For example, if in the original version of a parallelizable program P there is a data access $M[i][j]$ at iteration (i, j), a correct iteration space transformation of P would tell us when the access $M[i][j]$ would happen when the program is run in parallel.

2.7 An Example of an Iteration Space Transformation

This section will go through the details of determining iteration space transformation before reviewing the higher-level interpretation of the mathematics. Notably, we are concerned with transformations that have non-trivial parallelism. In some special cases, loops may be more easily parallelized if certain dependencies are independent of other iteration values. In other words, given a pair of nested loops like in **Figure 7**, if there are no dependencies between an iteration in $T[i][j]$ and $T[i+1][j]$ for all values of i, j, then the computation of rows $1 \leq i \leq I$ may be computed on I different processors. This is rarely the case, however. In the next example, we will build off of our example program introduced in **Figure 7** to show a more complex transformation. For clarity, we repeat **Figure 7 (again)** below:

Figure 7

```
// where T is a two dimensional array (matrix)
for (int i = 1; i <= I; i++) {
   for (int j = 1; j <= J; j++) {
        S1: T[i][j] = T[i-1][j] + T[i][j-1];
    }
}
```

Consider the following iteration space transformation T on an arbitrary iteration vector $\vec{j} = \begin{bmatrix} i \\ j \end{bmatrix}$ in the computation space of the program P above. Suppose that we have determined (through calculations that will be explained in a later section) that an appropriate iteration space transformation T maps $\vec{j}$ to a vector $\vec{k} = \begin{bmatrix} i+j \\ i-j \end{bmatrix}$. Formally, we can express this as:

$$T\left(\begin{bmatrix} i \\ j \end{bmatrix}\right) = \begin{bmatrix} i+j \\ i-j \end{bmatrix} \tag{2.9}$$

The matrix corresponding to this transformation can be found manually by finding the images of the standard basis vectors in Z^2:

$$T\left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad T\left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}\right) = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \tag{2.10}$$

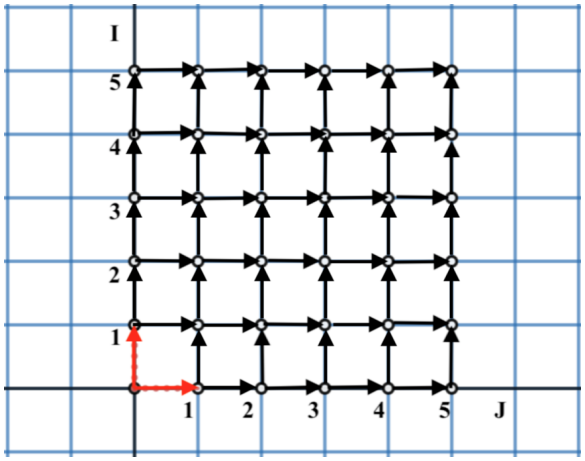
Since all vectors in Z^2 can be expressed as a linear combination of the two basis vectors $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$, taking the images of these vectors under T allows us to construct a matrix that will transform an arbitrary iteration vector $\vec{j}$ in the Z^2, relative to this standard basis. We can confirm this simply be checking to see if the transformation holds for $\vec{j}$:

$$T = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} i+j \\ i-j \end{bmatrix} \tag{2.11}$$

The transformation T can also be thought of as a change-of-coordinates transformation, or a transformation of a space relative to a new basis. In Linear Algebra, a set of basis vectors can describe all other possible vectors within a space through what is called a linear combination. Linear combinations describe vectors in terms of other vectors, and in this case specific integer weights. These new basis vectors are can be used as hyperplanes, which, as we will see, are used to apply tiling transformations on iteration space to increase locality. For now, we can think of the bases/hyperplanes, (shown in each respective diagram in red), as guides or constraints along which other iteration space transformations can occur to ensure that additional transformations work relative to the new basis. Hyperplanes will be fully defined during the discussion on Loop Tiling and Diamond Tiling, as they are not critical for understanding this example.

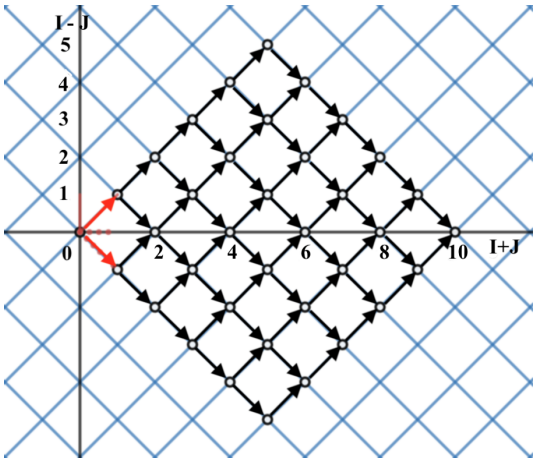
Visualizing the transformed iteration spaces provides more insight into how our transformation. **Figure 8** and **Figure 9** show the iteration space of P before and after the application of the transformation T, and the space after the transformation respectively:

Figure 8



Untransformed Iteration Space of P, arrows indicate flow dependences where the arrow points from a read to a write. I = J = 5

Figure 9



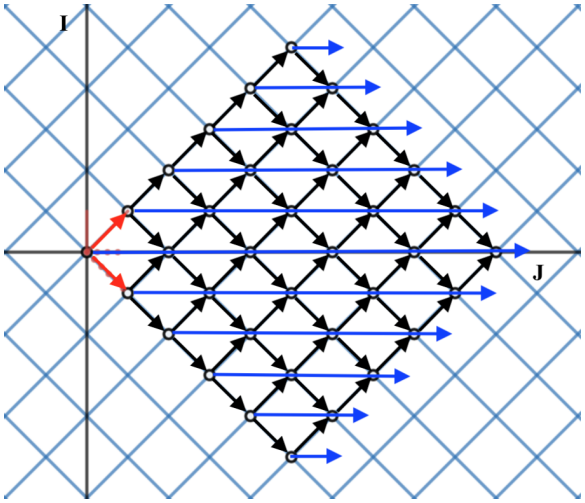
Transformed iteration Space of P, arrows indicate flow dependences where the arrow points from a read to a write. I = 5, J = 10

It is worth recognizing why we can even use parallelism on our example program in the first place. **Figure 8** shows the flow dependencies in the iteration space depicted as arrows traveling from a read access to a write access. Checking our code,

we can see that our program starts at iteration $(0, 0)$. Additionally, we can see in **Figure 8**that the value produced at iteration $(0, 0)$ is used to compute the values needed for iterations $(0, 1)$ and $(1, 0)$. The normal loop proceeds to calculate $(0, 2)$ after $(0, 1)$, as instructed by the inner loop, without starting on $(1, 0)$ at all. If the program had calculated the value for iteration $(0, 1)$ and $(1, 0)$ instead of just $(0, 1)$, it could have calculated the values for iterations $(1, 1), (2, 1)$, and $(1, 2)$ the next step, rather than only calculating $(0, 2)$.

This presents the opportunity to create a queue of multiple processors that can begin computations at different stages of the computation. Such a transformation allows for a kind of parallel execution referred to as wavefront or pipeline parallelism ([1], [4], [9],[14]). This form of parallelism allows for multiple processors to be used as more information for computation becomes available. The diagram in **Figure 10** below shows when during the stages of the computation more processors can be applied.

Figure 10



The arrows in blue represent the work that can be done by an independent processor using the transformed iteration space of the original program. The black arrows, which also represent dependencies, can now be thought of as inter-processor communication. Starting from $(0, 0)$ on one processor in our transformed iteration space, we can add two processors to handle computing the values at iterations $(1, 1)$ and $(1, -1)$. After these values have been calculated, we can start processors at $(2, -2)$, $(2, 2)$, and $(2, 0)$, since their dependencies have already been computed by the processors that calculated the values at $(1, 1)$ and $(1, -1)$. In this way, different sets of iteration coordinates without direct dependencies between them (e.g. $(0, 0)$ and $(2, 2)$) can be executed on different processors, so long as those processors communicate between iterations. Code derived from this transformation would

iterate over the programs iteration space in such a way that would enable more processors to work on the computation as it went on.

While achieving parallelism is a good thing, it does come at a cost in this example. There are several points that can make pipelined parallelism a hard sell, depending on the situation. First is the penalty of interprocessor communication, which in this example must happen at least twice per iteration. This adds a heavy time penalty to code running pipeline parallelism, as each iteration must wait until all required processes are complete, and all information has been transferred to the right processor ([12],[19]).

Second, the amount of work that can be done at a given time is bottlenecked by the fact we can only start parallel execution on one processor. This problem is two-fold, since in the beginning, threads must wait for the necessary information to be calculated for all processors to run, and near the end of the computation, there are fewer and fewer values that processors can actually compute. This can be seen in the shape of the iteration space in **Figure 9**. Notice that not all blue arrows (processors) are used from start to finish. During the middle of the computation, when there is the most work to be done, there are more processors. However, at the start and end, the number of processors is bottlenecked. These problems are referred to as pipeline fill and pipeline drain ([2],[12]).

2.8 Importance of Locality in Parallelism

Moving forward, we start to build up our model of parallelism to encompass the hardware constraints discussed in the introduction of the paper. While the Polyhedral Model has a lot to offer in terms of theoretical framework, in practice, parallelizing programs requires a significant amount of programmer level implementation in order to make the most of multiple processors. In particular, programmers can develop heuristic, hardware sensitive approaches to implementing parallelism that rely on the memory hierarchy of computers. As we will see, such implementations can be applied on top of iteration space transformations through the use of methods called loop tiling. Diamond Tiling, which will be introduced slightly later, takes implementation a step further, allowing multiple processors to start computations simultaneously.

Locality is the tendency for a processor to request the same or adjacent locations in memory in a given period of time ([2], [4], [12] ,[19]). There are two kinds of locality, spatial and temporal locality ([12]). The former describes the tendency of a processor to access data at and around given memory location multiple times

throughout a computation, i.e., if an item is referenced, items physically near that item in memory are likely to be accessed ([12]). The latter refers to the tendency of a processor to access a specific area in memory multiple times throughout a given computation, i.e., a referenced item is likely to be referenced again soon ([12]).

In a memory hierarchy, data are copied from adjacent levels of the memory hierarchy. The smallest unit of information that can be either present or not present in a cache is a **block (line).** When data requested by the processor, if the required block is present in the upper level, this is called a **hit**. If the data are not loaded into the upper level, it is called a **miss.** When a miss occurs, lower level memory in the hierarchy is accessed and the data are transferred between adjacent levels.

- The **Hit Rate** of a cache is the ratio of memory accesses found in a level of the memory hierarchy. (1 - miss rate)
- The **Miss Rate** of a cache is the ratio of memory accesses not found in a level of the memory hierarchy.
- The **Hit Time** is the time it takes to access the upper level of memory, which includes the time needed to determine whether the access is a hit or miss. (search time)
- The **Miss Penality** is the time needed to replace a block in the upper level with a corresponding block in the lower level and deliver this block to the processor.

The term **Cache** refers to the level of the memory hierarchy between the processor and main memory. When a cache miss occurs, the cache fetches the missing block from main memory and assign it to a location. The big idea behind caching is prediction: caching relies on principles of locality in order to exploit faster data access.

A potential bottleneck on the speed of parallel computations is related to how fast data can be read in and out of memory, or the speed at which data can be communicated between concurrently running processors. For stencil computations, we want to ensure that data required is cached before it is used, to help prevent costly cache-misses during runtime. While iteration space transformations allow a computation to take advantage of multiple processors, if these programs cannot keep up with the rate at which multiple processors are requesting information, the benefits of parallelized computation can be greatly diminished. The introduction of locality to parallelized iteration space transformations introduces another layer of added complexity. On one hand, we must confirm that the transformations established above do not alter the dependencies of any entry in any given way, i.e., satisfy the constraints for each entry. Now, managing locality requires that a given computation also maximizes the use of data that can be stored in cache, on top of

establishing subsets of the original computation that can be executed independently and then recombined. Moreover, should processors need to exchange information, we want to maximize the amount of information they can exchange at once to prevent unnecessary overhead costs for inter-processor communication.

Lastly, there is the issue of locality and scalability through cache usage. Wonnacott et al. described how pipelined parallelism does not always allow for scalable locality [19]. This translates to faster memory accesses even as the size of a problem increases, since frequently used values will still be held in cache longer than less frequently used ones. Likewise, more computation relevant data in cache allows a program to run faster by supplying processors with the values they need faster. Moving forward, each of these problems will be addressed in the coming sections, where we will introduce contemporary solutions improve the efficiency of parallel transformations.

The next sections build off of the use of techniques such as iteration space transformations, dependency analysis and loop constraints to produce hardware-minded optimizations.

2.9 Loop Tiling and Diamond Tiling

At a high level, loop tiling changes how a program such as a stencil computation iterates through its iteration space. More specifically, loop tiling is a form of iteration space transformation that is focused on maximizing memory locality for computation heavy programs. The goal of loop tiling is to ensure that required values for computations are stored in cache and not displaced before they are needed in the computation. This is done by "tiling" the iteration space of a loop by iterating over well defined subsections of the iteration space. By tiling, specific values are loaded into cache with the hope that when another read/write access requires a value to be looked up, the value is still or already available to the processor. **Figure 11** showcases this concept below:

```
// Non-tiled version
for (int i = 1; i <= I; i++) {
   for (int j = 1; j <= J; j++) {
        S1: T[i][j] = T[i-1][j] + T[i][j-1];
    }
}

// Tiled Version
for (int i = 1; i <= I; i += tile_size) {
   for (int j = 1; i <= J; j += tile_size) {
```

```
        for (int t_width = i; t_width < i + tile_size; t_width++) {
           for (int t_height = j; t_height < j + tile_size; t_width++) {
              S1: T[i][j] = T[i-1][j] + T[i][j-1];
           }
        }
    }
}
```

Here, the outer two loops are incremented by tile size rather than one each iteration. Nested within these outer two loops is another pair of loops that iterates within the subsection defined by the tile size. In this way, one can think of the outer two loops as iterating over the different tiles of an iteration space, and the inner two loops iterating through each of the tiles individually.

Notably, tile size is often symbolic, meaning its value is not known until runtime. Since cache sizes and architecture of a computer matter in determining how much information can be stored in cache, optimal tile size section is notoriously difficult to calculate and is often subject to trial and error, especially when implemented manually.

Diamond Tiling is a special iteration space transformation that manages memory locality through loop tiling, with the added benefit of providing concurrent start up under certain conditions. Concurrent startup bypasses the pipeline fill and drain introduced by wavefront or pipelined parallelism, as multiple processors can begin computations as soon as the program starts.Diamond tiling works by tiling iteration space along hyperplanes, which are often related to the bases of the iteration space. Such hyperplanes are defined as an affine $n-1$ dimensional subspace of an n dimensional space ([5]). Affine describes the mathematical form of the subspace as being some linear expression plus some constant value ([5]). An example of a hyperplane of a plane is any line that is on the surface of that plane - in our previous example, our basis vectors for our transformed iteration space are also examples of hyperplanes. Mathematically, we can express the hyperplane of a statement (in a loop at iteration i) as:

$$\phi_{S_i} = \mathbf{h}\vec{j} + h_0$$

where $\mathbf{h}$ is a vector of with $n-1$ components oriented perpendicular to the bases (hyperplanes) of the iteration space, $\vec{j}$ is an iteration vector, and h_0 is a constant shift component ([14]). If tiling hyperplanes are linearly independent (they cannot be expressed as a linear combinations of each other) and all dependencies between

two iterations are non-negative (forward in time), they are valid and satisfy the equation:

$$\phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s}) \geq 0,\ \langle s,t\rangle \in \mathcal{P}_\delta, \forall e \in E(\mathcal{P}_\delta)$$

where $\vec{s}, \vec{t}$ are source and sink nodes of dependencies, ϕ_{S_j}, ϕ_{S_i} are hyperplanes, and $\mathcal{P}_\delta$ is the dependence polyhedron, where $e \in E(\mathcal{P}_\delta)$ represents and edge (dependence) of the polyhedron.

The performance gains form concurrent startup as well as cache-based optimizations makes Diamond Tiling one of the most desirable iteration space transformations for practical applications. Moving forward, this paper hopes to establish a methodology for using the polyhedral model and tools such as ISCC in order to verify the correctness of diamond tile code that has been generated either by a machine or a programmer. In the next section, this paper discusses how programs, including Diamond Tiled ones, can we dissected and interpreted as a series of relations and sets, which we can use to find key dependence information for an iteration space transformation.

2.10 ISCC and the Relation Interpretation of Dependencies

In this section, we will discuss how code can be analyzed using the framework of the polyhedral model to assure that memory-based value flow dependencies are not violated by in iteration space transformation. ISCC, a affine counting calculator developed by Sven Verdoolaege, allows for users to interactively manipulate sets of integer tuples [17]. Constraints on such tuples must be affine, or existentially quantified. Using ISCC, programmers can manually dissect and reconstruct flow dependencies of programs, as well as model the changes in flow dependences when certain constraints and domains are changed. This becomes a powerful tool, as it allows different flow dependencies to be calculated and re-expressed through a series of mappings, compositions, and inversions. ISCC users can create relations of flow dependencies using different array accesses within a loop statement. In many programming languages such as C, Java, and Python, the anatomy of a loop statement can be split into two distinct parts: the part that iterates, and the part that is iterated upon.

Chapel, a language designed around scalable parallelism, separates these two pieces of a loop into different abstractions. An iterator is a piece of a program that defines and yields the iteration space of an algorithm. Following an iterator is often a

standard for loop, which iterates over specific values yielded by the iterator. The ability to separate the iterator from the rest of the loop allows for extremely complex movement through the iteration space to be defined without distracting programmers from seeing what the loop is actually doing. In the following examples, code will be presented in Chapel to reflect this abstraction, as the translation from queries to ISCC and the Chapel code allows for a cleaner description of dependence analysis. In the following section, we will examine how we can use ISCC to determine whether two programs will produce an equivalent output for all valid inputs. Consider the following to programs, one with a tiled iteration space.

Figure 12

```
iterator orig(T, N) {
   for (int i0 = 1; i0 < T; i0++) {
      for (i1 = 1; i1 < N; i1++) {
         yield(a, b);
      }
   }
}

for (a, b) in origin(T, N) {
   A[(a+1)%2][b] = ( A[a%2][b-1] + A[a%2][b] + A[a%2][b+1] ) / 3;
}
```

Figure 13

```
iterator tile(T, N) {
   for (int c0 = 0; c0 <= floord(N - 1, 8); c0 += 1)
      for (int c1 = 0; c1 <= floord(T - 1, 8); c1 += 1)
         for (int c2 = 8 * c1; c2 <= min(T - 1, 8 * c1 + 7); c2 += 1)
            for (int c3 = 8 * c0; c3 <= min(N - 1, 8 * c0 + 7); c3 += 1)
               yield(c2, c3)
}
for (a, b) in tile(T, N) {
   A[(a+1)%2][b] = ( A[a%2][b-1] + A[a%2][b] + A[a%2][b+1] ) / 3;
}
```

Using the memory-base flow dependence analysis described in section 2.4, we will demonstrate that these two programs in figures 12 and 13 actually produce the same output for all valid inputs. Recall that in the previous method, we showed how memory-based dependencies can be described as two or more value-based dependencies. Likewise, we can use memory-based dependencies to derive value-based dependencies. For determining correctness, we care about determining the value-

based flow dependencies. Given two different memory-based flow dependencies and an output dependence of some program, we can calculate extract the value-based flow dependence by taking the difference of one memory-flow dependence m_1 and the relational intersection of another memory flow dependence m_2 and an output dependence o_1 - in other words $m_1 - (m_2 * o_1)$, where m_1, m_2, o_1 are relations representing memory flow and output flow dependencies, respectfully. In the next section, we briefly discuss an algorithm for computing memory based flow dependencies on the example programs above. We then use those dependencies with the expression above to confirm that their output is

2.10.1 An Algorithm for Finding Memory Based Flow Dependencies in Loops

In this section we will begin to introduce the ISCC notation for describing sets and relations of integer-k tuples. Programs that are viable for optimization by the polyhedral model, like the ones in figure 12, can be decomposed into several different relations that accurately capture the flow of information. Consider the the iterator for both programs in figure 12. We can represent the iteration space of the iterator as the set of all tuples $(i0, i1)$ where $1 <= i0 < T$ and $1 < i1 < N$. In ISCC, we would represent all such tuples as:

iterationSpace2D := [T, N] -> { yield[i0, i1] : 1 <= i0 < T and 1 <= i1 < N };

where [T, N] is a generic descriptor for all pairs in the bounds the set. In practice, we can think of the iterator yielding a multi-set (a set that allows multiple instances of the same element) of tuples that represent the combinations of i0 and i1 that the program will use during its execution. In order to map an iterator yield to an execution, we need to create a relation from a yield tuple to an execution tuple. For the rest of this example, we continue using the second program in figure 12 as a reference for building these relations. Construction for the first program is exactly the same. We can map a single yield (i0, i1) to the execution (a, b), as described by the "for (a, b) in origin(T, N)" statement, where (a, b) belongs to the multi-set of the iterator origin(T, N).

yield2D := [T, N] -> { yield[i0, i1] -> exe[a, b] : a = i0 and b = i1 };

Note here how we declare that a = i0, and b = i1, as this describes when a given yield produces a given execution. The next step is to determine the set of all iteration vectors whose head and tail are both executed within the bounds of the iteration space. We do this by applying the yield relation yield2D to the set of iteration vectors in the set iterationSpace2D. By taking the cross product of the resulting set and itself

again, we can define all legally bound executions in the iteration space as a relation from some exe[a, b] $\rightarrow$ exe[a', b'].

boundCross2D := unwrap((yield2D(iterationSpace2D) cross yield2D(iterationSpace2D)));
= [T, N] -> { exe[a, b] -> exe[a', b'] :
0 < a < T and
0 < b < N and
0 < a' < T and
0 < b' < N };

Next, we need to define both a relation from a given execution to a write access of the array A, and another relation from a given execution to a read access of A. Such relations are defined as:

exeToRead := [T, N] -> { exe[a, b] -> A[a mod 2, b - 1] };
exeToWrite := [T, N] -> { exe[a, b] -> A[(a + 1) mod 2, b] };

Notice how we are mapping a given execution exe[a, b] to the array cell that is written to for the write relation, A[(a+1) mod 2, b], and from an execution exe[a, b] to one of the three array cell read accesses on the left hand side of the equals sign in the loop body, A[a mod 2, b - 1]. Note that we also could have chosen A[a mod 2, b] or A[(a - 1) mod 2, b + 1] for the read accesses - this makes no difference for the sake of example. As mentioned earlier, we will be using current flow arrows to describe the direction in which information moves from array cell to array cell, thus we want to create a relation from a write access to a read access. We can do this by composing the write relation with the inverse of the read relation, to produce:

exeToRead^{-1}= [T, N] -> { A[i0, i1] -> exe[a, b = 1 + i1] :
(i0 + a) mod 2 = 0 and
0 <= i0 <= 1 };

currentFlow := (exeToWrite . (exeToRead^{-1}));
= [T, N] -> { exe[a, b] -> exe[a', b' = 1 + b] : (1 + a + a') mod 2 = 0 };

Now, we can take the relational intersection of the current flow relation and the boundCross2D relation, producing the set of all current flow vectors bound by the iteration space.

boundCF2D := boundCross2D * currentFlow;

Lastly, we want to assure that all executions arrive lexicographically in order with respect to their values of a, and b. For instance, if a = 1, b = 2, and a' = 3, b'

= 1, we would want to be sure that the execution exe[a, b] came before exe[a', b'], as doing so preserves the original ordering as used by the loops in the original program.

forward2D := [T, N] -> { exe[a, b] -> exe[a', b'] : a < a' or (a = a' and b < b') };

Finally, by taking the intersection of the relation for bound current flow arrows, and the relation for legal orderings of executions, we can define a memory based flow dependence for the second program in figure 12: (note there are two produced)

forwardBoundCF2D := boundCF2D * forward2D;
= [T, N] -> { exe[a, b] -> exe[a', b' = 1 + b] : (1 + a + a') mod 2 = 0 and
0 < a < T and
0 < b <= -2 + N and
a' > a and
0 < a' < T };

We can repeat this process with a different read access relation (by choosing one of the other array accesses, in this case A[a mod 2, b]) to produce another memory flow dependence of the same program:

memDep2 := [T, N] -> { exe[a, b] -> exe[a', b' = -1 + b] : (1 + a + a') mod 2 = 0 and
0 < a < T and
2 <= b < N and
a' > a and
0 < a' < T };

Lastly, we can produce an output dependency by changing the current flow relation in the process above with the relation produced by joining the write relation with its own inverse:

currentFlow := (exeToWrite . (exeToWrite^{-1}));

or [T, N] -> { exe[a, b] -> exe[a', b' = b] : (a + a') mod 2 = 0 }

and receive the value-based output dependence:
outDep1 := [T, N] -> { exe[a, b] -> exe[a', b' = b] : (a + a') mod 2 = 0 and
0 <= a < T and
0 <= b < N and
a' > a and
0 <= a' < T };

Now, if we apply the value flow calculation: memDep1 - (memDep2 * outDep1):

originalValFlow := [T, N] -> { exe[a, b] -> exe[a', b' = -1 + b] : (1 + a + a') mod 2 = 0 and
0 < a < T and
2 <= b < N and
a' > a and
0 < a' < T };

Applying this entire process of reducing two memory based dependencies and an output dependency, but now with the tiled program, we produce the following value-flow dependence:

tiledValFlow := [T, N] -> { exe[a, b] -> exe[a', b' = -1 + b] : (1 + a + a') mod 2 = 0 and
0 < a < T and
2 <= b < N and
a' > a and
0 < a' < T };

which is equal to the value flow dependencies that was calculated for the untiled version of the program in Figure 12. This equivalence of value-based flow dependencies shows that the two programs are equivalent.

There may be some cases in which a direct comparison between two flow values is impossible. In this example, we were lucky in that our value-based flow dependences were actually the same. In reality, we may have to build another relation that maps one iteration space to another iteration space, in order to confirm that the two are actually equivalent. In the example programs given in figure 12, both programs used symbolic constraints [T, N]. It is possible that a transformation might express a new flow dependence with a different number of constraints. In terms of transformed code, this means that transformed loops may have a different number of loops in the nest (dimensions) than they did previously. For example, supposing that the code in figure 12 did not actually produce the same relation from our algorithm alone, we could construct a comparison relation:

T24 := [T, N] -> exe[i0, i1] -> yield[c0, c1, c2, c3] : i0 = c2 and i1 = c3 ;

This relation describes a map from on iteration vector (i0, i1) in a two-dimensional iteration space to another iteration vector (c0, c1, c2, c3) in a four dimensional iteration space. We can construct relations such as these to "put one space in terms of another space", that allows a valid comparison to be drawn. The underlying

mathematical reasoning behind this is based in linear algebra, where we would be describing iteration space transformations as linear transformations between different bases and or vector spaces. In this sense T24 is like a transition matrix between different vector spaces.

3 Programmer Interactive Loop Optimization Correctness Checking with Polly

For this semester, we are proposing to implement an experimental loop transformation correctness checker that builds off of the infrastructure of Polly, an LLVM-IR to LLVM-IR optimization and analysis tool based off of the polyhedral model. While automatic target-code optimizers exist in compilers like PLuTo, there is no way for programmers interactively check whether a proposed tiling or loop transformation is equivalent to another. In certain applications, providing easy correctness verification of user-created transformations allows programmers to implement code optimizations that are either too simple for (and not worth the overhead of optimizing automatically), or are currently beyond the automatic capabilities of modern compilers. Additionally tools that allow for correctness checking could be useful in developing automatically parallelizing compilers with more robust feature sets. In some instances the detection algorithms responsible for detecting usable code may fail to identify or format relevant portions of code that need to be optimized. Allowing programmers to select which portions of code to compare bypasses this issue, at least until compilers with a wider set of applicable parallelism are more mainstream.

This project utilizes several major tools in order to perform correctness checking. First, we plan to use clang to compile C source code into optimized LLVM-IR, which is then interpreted by Polly. By using Polly, we hope to leverage the internal ISCC/ISL library to calculate dependencies using the algorithm that we showed at the end of the literature review. More specifically, if give two pieces of code, we want to use the framework provided by Polly to detect, extract, and analyze loop dependencies, before we compare them with the value-flow dependence algorithm. We anticipate that this implementation will likely take the form of either a command line tool, or a callable flag that can be used when invoking Polly. In this sense, our program will allow programmers to "check their work" should the need arise to do tedious, and bug-prone optimizations by hand.

3.1 Limitations on Symbolic Tiling Analysis

The scope of this project would likely be constrained by modern limitations on transformations such as Diamond Tiling, where loop constraints and bounds can vary depending on a symbolic tile size (tau), we suggest an implementation that can handle calculated correctness for values of tau determined at compile time.

Recall that loop tiling partitions the iteration space of a loop nest in such a way that chunks or "tiles" of the iteration space are iterated over. The benefit of this is that we can often have different tiles run on different processors, which greatly speeds up calculations by breaking up the workload. By symbolic tiling, we mean to say that the dimensions of the tile used are unknown at compile time. For example, the ISCC relation below represents a non-symbolic diamond tiling of a 2D iteration space, where the tile size (tau) is equal to eight. Recall that a diamond tiling of an n-dimensional iteration space partitions the iteration vectors in that space along n, n - 1 dimensional hyperplanes. A symbolic tiling would use tau in place of 8.

Figure 11

```
iterationSpace2D := [T, N] -> { [i0, i1] : 0 <= i0 < T and 0 <= i1 < N };

// tau = 8
tile2D := { [i0, i1] -> [k0, k1, i0, i1] :
                exists r0, r1 :
                (0 <= r0 < 8) and (8*k0 + r0 = 0*i0 + 1*i1) and
                (0 <= r1 < 8) and (8*k1 + r1 = 1*i0 + 0*i1) }; // (tau*k1 +
                    r1) is symbolic

diamondTile2D := tile2D * iterationSpace2D;

codegen(diamondTile2D);

// output of codegen(diamondTile2D);

for (int k0 = 0; k0 <= floord(N - 1, 8); k0 += 1)
  for (int k1 = 0; k1 <= floord(T - 1, 8); k1 += 1)
    for (int i0 = 8 * k1; k0 <= min(T - 1, 8 * k1 + 7); i0 += 1)
      for (int i1 = 8 * k0; i1 <= min(N - 1, 8 * k0 + 7); i1 += 1)
        (i0, i1);
```

The relation tile2D represents how an arbitrary iteration vector $\begin{bmatrix} i0 \\ i1 \end{bmatrix}$ may be transformed along the hyperplanes $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ and $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$. Specifically, the equations:

$$8 * k0 + r0 = 0 * i0 + 1 * i1 \tag{3.1}$$

$$8 * k1 + r1 = 1 * i0 + 0 * i1 \tag{3.2}$$

represent how the a loop would move through the iteration space at each iteration $(i0, i1)$.

We can unpack how this iteration occurs by examining equations (3.1) and (3.2). Consider (3.1) - here, k0 represents the current tile that is being iterated over. Since tau (tile size) is eight, 8*k0 will allow us to jump to the k0-th tile. The variable r1 represents where we are in tile k0. The expression 8*k0 + r1 describes a location in the iteration space that is at the r1-th location in the k0-th tile. This expression is set to be equal to the dot product of a given iteration vector $\begin{bmatrix} i0 \\ i1 \end{bmatrix}$ and the hyperplane $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

The significance of tiling along this hyperplane is that the movement defined by the equations in this example move along the y-axis. At a high level, $(0 <= r0 < 8)$ and $(8 * k0 + r0 = 0 * i0 + 1 * i1)$ describes the valid y-coordinates of all iteration vectors in the diamond tiled space, and $(0 <= r1 < 8)$ and $(8 * k1 + r1 = 1 * i0 + 0 * i1)$ represents all the valid x-coordinates. Taking the relational intersection with the iteration space provides a result that is bound by the limits (T, N) of the iteration space, which gives us the tiling for the code in figure 11.

We will not be attempting to handle any cases of symbolic tiling, as this problem is currently beyond the infrastructure provided by polly. Instead, this project focuses on allowing researchers and programmers to more readily access the computational infrastructure already supported by Polly.

3.2 Design Details and the Polly Pipeline

In this section, we discuss the details of how we see our project being implemented into Polly, in addition to a high level overview of how Polly carries out its optimiza-

tions. We have broken down our algorithm into a process consisting of four steps: static control part (SCoP) detection, dependency extraction, dependency analysis, and equality checking. To see how the relevant pieces of Polly fit into this model, we will break down Polly's optimization process into a series of steps, and show how they are relevant to our own algorithm. Additionally, will briefly show how programmers can access relevant pieces of Polly to perform loop analysis through CLI commands.

3.2.1 Static Control Part Detection in LLVM and C Code

The first portion of our algorithm is dependent on Polly's built in semantic analysis of static control parts. In this context, a static control part (SCoP) is simply a piece of a program that semantically represents code that is relevant to optimizations using the Polyhedral Model [7][15][16]. In the context of Polly, SCoP's are determined by both code location in the source file, as well as its semantic meaning [7]. Formally, static control parts can be represented as a subgraph of a control flow graph, but this is beyond the scope of this paper [7]. Importantly, SCoP's are not determined syntactically in Polly. This is because a SCoP that appears to be syntactically valid is not necessarily semantically valid - for example, the potential SCoP:

Source: [7]

```
for (int i = 0; i < n + m; i++) {
   A[i] = i;
}
```

takes the form of a loop, which syntactically appears like something that may need to be optimized, but semantically, we can see that it does not alter control flow in any way that suggests that optimization is necessary, such as multiple accesses to the array, or any form of branching logic. In other words, semantically, there is no "interesting" data flow [7].

In this next example, we show an input for which there is more interesting, although still relatively simple SCoP detection. This input consists of a pair of loops and an inline function. Here, the inline function only acts as a way to simulate a Chapel-style iterator, i.e., separate the loop body code from the code responsible for iterating. Inspection of the loop bounds and the inline function reveals that both "stencil_fake_iterator" and "stencil_original" are actually the same program. Notice that the programmer can optionally mark locations with pragma scop and pragma endscop to explicitly alert the compiler to SCoP's.

```
inline void body(int t, int i, int N, double A[][N]) {
   A[(t+1) mod 2][i] = A[t mod 2][i-1] + A[t mod 2][i] + A[t mod 2][i+1] /
       3;
}

void stencil_original(int T, int N, double A[][N])
{
#pragma scop
   for (int t = 0; t < T; t++) {
      for (int i = 1; i < N-1; i++) {
         A[(t+1) mod 2][i] = A[t mod 2][i-1] + A[t mod 2][i] + A[t mod
             2][i+1] / 3;
      }
   }
#pragma endscop
}

void stencil_fake_iterator(int T, int N, double A[][N])
{
#pragma scop
   for (int t = 0; t < T; t++) {
      for (int i = 1; i < N-1; i++) {
         body(t, i, N, A);
      }
   }
#pragma endscop
}
```

In order to source C code to polly, we must first translate any input into LLVM-IR. As suggested by the Polly Documentation [15], we are using clang to do this with the following commands:

Source [16]:

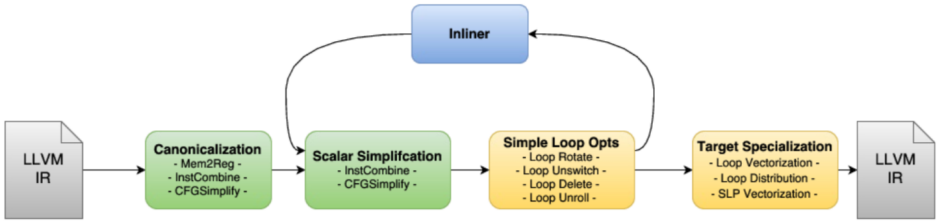
> clang -O3 -S -emit-llvm <FILENAME> -o source_conversion.s
> opt -S -polly-position=early source_conversion.s > source_code.preopt.ll

Notably, the execution order and the inclusion of specific routines in Polly can be bypassed or activated according to command flags. There are several important flags in these commands. In the first command, we convert C source code into LLVM-IR, a low level programming language that is very similar to assembly. It is a requirement that we use the [-O1 | -O2 | -O3] flag with clang when emitting

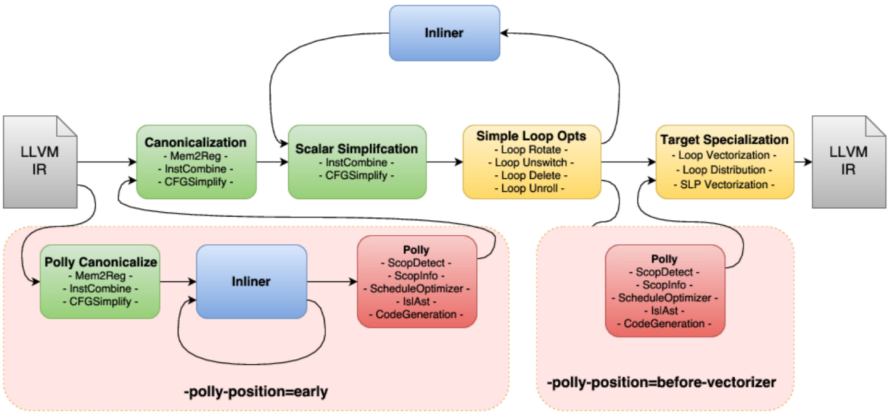
LLVM, as Polly does not even attempt to optimize and run on code not in this form [15][16]. In a standard pass through the Polly pipeline, canonicalization is the process of applying -mem2red, -instcombine, -cgsimplify, and some loop unrolling operations [16]. These operations are designed to allow Polly to primarily focus on scalar optimizations later in the pipeline [15].

We then invoke Polly through opt, which uses the -polly-position=early flag to prevent the canonicalization from seriously altering the source code. The process of canonicalization is responsible for several early optimizations that preserve the semantic meaning of static control parts, however, they may alter the code syntactically in some way. By running -polly-position=early and the when we use the command -polly-canonicalize, we reduce the number of transformations applied by polly to the IR code, meaning that detected SCoP's should still be recognizable to a programmer comparing detected SCoP's to their original source code [14]. This works by changing the order in which Polly runs its optimizations, after the initial conversion from C to LLVM-IR. The figures below show the difference between how information is passed to the rest of Polly:

Source: [16]



(Above) The standard Polly pipeline - fewer optimizations are done, output IR should be very similar/identical to original code.



(Above) Using the -polly-position=early flag

The use of the early polly position flag is important because it should theoretically allow us to access the value flow analysis from the original source program, rather than an optimized or heavily modified version. We hope that some of the code associated with these flags will provide the semantic analysis required for the detection and extraction of information from static control parts. The extraction portion of the algorithm will likely be handled by pieces of source code behind the -polly-ast, -analyze, -q, and the -polly-process-unprofitable flags to extract and analyze the translated source code. We elaborate more on these other flags and their properties in the next section.

3.2.2 SCoP Extraction and Dependence Analysis

In a standard Polly pass, extraction of SCoP's is handled by the -polly-ast flag:

```
> opt -polly-position=early -polly-ast -analyze -q source_code.preopt.ll -polly-process-
unprofitable > detected.out
```

This command allows to a programmer to view detected SCoP's from a given input program. We hope to utilize the source code associated with this flag for the extraction portion of our algorithm. The -polly-ast flag in combination with the -analyze flag tells Polly to only analyze and not to optimize input code, and to output human-readable SCoP representations on the abstract syntax tree. The -polly-process-unprofitable flag is also important because it enables optimization and analysis on code that may be "too simple" and not worth the computational overhead to optimize in the first place. Again, it is important that we try and keep the original dependencies in tact as much as possible throughout this process, which means the -analyze flag is relevant to our program [16].

In addition to SCoP's it is also important that programmers are able to recognize where specific dependencies come from in their source code. The -polly-dependence flag outputs the detected memory-based flow dependences for SCoP's. These internal objects allow us to extract and analyze the the memory flow dependencies required for our own flow equivalence algorithm. For the printing out dependencies, we can use the command:

```
> opt -polly-dependences -analyze source_code.preopt.ll -polly-process-unprofitable
> deps.out
```

An example output for the "stencil_original" function above is:

```
    Statements {
        Stmt0
            Domain :=
                [p_0, p_1, p_2] -> { Stmt0[i0] : i0 >= 0 and 2i0 <= -4 +
                    p_0 };
            Schedule :=
                [p_0, p_1, p_2] -> { Stmt0[i0] -> [i0] };
            ReadAccess :=     [Reduction Type: NONE] [Scalar: 0]
                [p_0, p_1, p_2] -> { Stmt0[i0] -> MemRef0[o0] : p_1 + 2i0
                    <= o0 <= 1 + p_1 + 2i0 };
            ReadAccess :=     [Reduction Type: NONE] [Scalar: 0]
                [p_0, p_1, p_2] -> { Stmt0[i0] -> MemRef0[o0] : p_1 + 2i0 <
                    o0 <= 2 + p_1 + 2i0 };
            ReadAccess :=     [Reduction Type: NONE] [Scalar: 0]
                [p_0, p_1, p_2] -> { Stmt0[i0] -> MemRef0[o0] : 2 + p_1 +
                    2i0 <= o0 <= 3 + p_1 + 2i0 };
            MustWriteAccess := [Reduction Type: NONE] [Scalar: 0]
                [p_0, p_1, p_2] -> { Stmt0[i0] -> MemRef0[o0] : p_2 + 2i0 <
                    o0 <= 2 + p_2 + 2i0 };
    }
```

Notably, the human readable form of the above polly object contains usable ISCC/ISL input. We are hoping to avoid doing text-based comparisons for correctness, as there is more than one way to write equivalent expressions in ISCC, especially with inequalities. Instead, we aim to run ISL queries on the polly objects that store dependencies in order to calculate program equivalence. The default dependence type for Polly analysis with the -polly-dependence flag is memory-based flow dependences. Significantly, Polly also appears to have some support for extracting value-flow dependences in addition to purely memory based ones, as evidenced by some Polly source code:

```
/* Source code from Polly */
DependenceInfo.cpp:

  61  static cl::opt<enum AnalysisType> OptAnalysisType(
  62    "polly-dependences-analysis-type",
  63    cl::desc("The kind of dependence analysis to use"),
  64    cl::values(clEnumValN(VALUE_BASED_ANALYSIS, "value-based",
  65                          "Exact dependences without transitive
    dependences"),
  66               clEnumValN(MEMORY_BASED_ANALYSIS, "memory-based",
  67                          "Overapproximation of dependences")),
  68    cl::Hidden, cl::init(VALUE_BASED_ANALYSIS), cl::ZeroOrMore,
```

```
69     cl::cat(PollyCategory));
```

which can be invoked by the following command flags [16]:

> opt -polly-dependences-analysis-type=value-based > clang -mllvm -polly-dependences-analysis-type=value-based

Depending on the usability of this feature, and whether we are able to calculate value-flow dependencies within Polly using the source code of DependenceInfo.cpp, we may only need to provide an algorithm that takes two value-flow dependencies and determines if they are equivalent via ISL, rather than extracting memory-based flow dependences and calculating value-flow dependences ourselves. However, we have been unable to confirm that whether the value-based flow dependences being optimized or preserved as their original values as extracted from SCoP's. The -polly-dependences-analysis-type=value-based did not output have any visual (human readable) confirmation of the different value dependencies that are being extracted and analyzed; we hope to confirm whether or not the code behind this flag is a viable option for our project. In the case that it is not, we also present a brief outline of all of how we expect all pieces of our project to come together.

3.2.3 Equivalence with the Value-Flow Algorithm

The following is an overview of an algorithm that we plan to use to determine whether two programs are equivalent with respect to output. The overall idea of this algorithm is to take two input programs, and produce their respective polly object representations. Using the extraction techniques and polly objects described in the previous sections, we hope to extract the information necessary to compute value-based flow dependencies for each input. With this information, we then check to see if the two produced value flows are semantically the same. At this point in time, the outline of the pseudocode below only represents the high level outline of the program - we expect implementation details to change as we explore more of the Polly code base. This last piece will likely involve queries to the ISL/ISCC module within polly, and will be the bulk of the original implementation rather than repurposing existing Polly code.

```
list<scop_obj> detect(string input code)
{
   /* code from Polly -polly-ast and -polly-scop */
}
```

```
/* optionally, we can use the infrastructure provided by Polly if it
    allows, otherwise, use the algorithm we provided in literature review
    */
dep_obj create_val_flow(scop_obj scop)
{
   dep_obj mem_dep1 = scop.get_mem_dep(read_access, write_access);
   dep_obj mem_dep2 = scop.get_mem_dep(read_access, write_access);
   dep_obj out_dep = scop.get_out_dep(write_access);

   /*  value flow is the difference of one memory-based flow dependence and
       the intersection of
        another memory dependence and an output dependence */

   dep_obj val_flow = mem_dep1 - (mem_dep2 * out_dep);
   return val_flow
}

bool equivalent(dep_obj val_dep1, dep_obj val_dep2)
{
   /* semantic comparisons through ISL queries */
}

void main(string input_code)
{
   list<scop_obj> scops = detect(input_code);
   list<dep_obj> val_deps;

   for (scop : scops) {
      dep_obj dep = create_val_flow(scop);
      val_deps.pushback(dep);
   }

   // compare adjacent values - since we use bitwise AND assignment with
       &=, if any value is false, eq will be false
   bool eq = true;
   for (dep1, dep2 : val_deps) {
      eq &= equivalent(dep1, dep2);
   }
   return eq;
}
```

Successful implementation of all four pieces of the program should produce a novel of a programmer-interactive loop transformation checker.

4 Analysis of our Proposal for Polly, Failure of Experimental Designs

After a considerable amount of work gaining some familiarity with the Polly codebase, we have determined that our proposed project is not currently compatible with the existing Polly Infrastructure. Despite research into the Polly pass pipeline, we were unable to isolate the individual mechanisms responsible for detection, extraction, and analysis of flow dependencies in a meaningful way. While pieces of relevant code where found, we were unable to run certain functions independently in a way that allowed us to use them as modular pieces of our own program. Code that we attempted to use also behaved drastically differently than we anticipated, which we hypothesize was likely due to either due to optimizations that came from preprocessing done with clang, and possibly by Polly, as there was little to no way to identify which static control parts had been optimized, in addition to how they were optimized without running memory dependence analysis on the machine-generated outputs by hand. This was one of the predominant challenges that prevented us from successfully utilizing the existing codebase.

4.1 Static Control Parts Detected with Different Loop Dimensions than Input Code

In the previous section, we discussed the -polly-position=early flag was important for preserving the extracted dependencies from LLVM. After exploring the polly codebase, and confirming that the infrastructure within the LLVM pipeline would support our implementation, we encountered two different complications in our attempted approach to use Polly to detect and extract dependencies. First, the we discovered that the -polly-position=early flag did not preserve the actual dependencies to the extent that we expected. In all cases that we tested, even three and four dimensional loops underwent a process of loop unrolling, a process in which the number of dimensions of a loop nest is decreased. Although these transformations are done before the significant automatic optimizations in Polly, they still change the syntax of the source code enough that it becomes very difficult for a programmer (especially one not familiar with loop transformations) to actual distinguish what SCoP's are actually being optimized. Furthermore, we found that regardless of whether we used

the -polly-position=early flag, the detected SCoP would be the same. Below, we have a piece of example code and the output shown from including and excluding the polly-position flag. Below, we show the detected SCoP's, and both memory-based and value-based flow dependencies calculated from the SCoP in the basic "stencil_original" program.

```
void stencil_original(int T, int N, double A[][N])
{
#pragma scop
        for (int t = 0; t < T; t++) {
                for (int i = 1; i < N-1; i++) {
                        A[(t+1)%2][i] = A[t%2][i-1] + A[t%2][i] +
                            A[t%2][i+1] / 3;
                }
        }
#pragma endscop
}
```

Notice that while the original program has two loop dimensions, (two nested loops), whereas the extracted SCoP only contains a loop with one dimension. In addition to changing the polly-position flag, we found that this form of loop unrolling would still occur when we modified the optimization levels within clang to the lowest setting (-O1). It should be noted that all of these examples utilize LLVM produced by clang on the lowest compatible optimization setting, -O1.

```
:: isl ast :: stencil_original :: %9---%21

if (1 && 0 == ((p_0 >= 3 && p_1 <= -1) || (p_0 >= 3 && p_2 <= -1)))

    {
      Stmt0();
      for (int c0 = 0; c0 < p_0 - 2; c0 += 1)
        Stmt2(c0);
    }

else
    {  /* original code */ }
```

This reveals several issues with our approach. First, the fact that SCoP detection did not change with the polly-position flag suggests that clang may also be performing loop optimization when emitting LLVM in addition to or instead of Polly itself. This is further supported by the fact that the LLVM that enters polly, regardless of the

degree of optimization indicated to clang by the [-O1 | -O2 | -O3] options, produces the same output when extracting SCoP's and finding dependencies. Since Polly does not recognize LLVM from clang that is generated without a -O optimization flag, it is not possible to give compatible input that is close enough to the original code, i.e., sufficiently un-optimized, to Polly. At this junction, it may be beyond the scope of this project, (as well as counterproductive) to eliminate all forms of pre-polly optimization by investigating or manipulation clang internals.

The inability to keep accurate internal representations of dependencies and static control parts (relative to their original source code inputs) is problematic for the usability of our program. While the optimizations run by Polly and clang are undoubtedly correct, using machine generated code does not effect our ability to analyze correctness, it poses the additional challenge of unpacking the various internals that Polly and clang are making in order to test and confirm dependency equivalence. This poses a large challenge to implementation, as even with a simple example of an optimizable program, like the example above, our use of Polly and clang have not only changed the syntactical structure of the program, but also renamed all variables. Notice that in our original program, induction variables have names t and i, whereas in the optimized code, we see references to p_0, p_1, and c0. It also appears that Polly is using inline functions for loop bodies with "Stmt0()" and "Stmt2(c0)" - however, we have been unable to confirm whether this is accurate or not due to the inability to access the attributes of polly SCoP objects during testing.

This development also makes it impossible for us to apply custom-built relations that compare two relations that may be defined over different iteration spaces. For example, we showed at the end of the literature review section that if two produced value flow relations have different symbolic constraints or dimensions, we may need to construct a special relation to compare the relations. Such a program for stencil original and stencil fake iterator would simply be a relation from one two-dimensional space to another: (exe[t, i] -> exe[t', i'] : t = t' and i = i'). But notice that this unique relation does not work on the internally represented version of polly, which only utilizes one dimension due to loop unrolling. This would make our relation look similar to (exe[i] -> exe[t, i] : constraints on t, i). Consequently, our fallback for comparing relations that may appear drastically different would need to change to handle internal polly representations of input programs, rather than our original programs.

As we would expect from our experience with the SCoP detection, since extracted dependencies relied on the internal representations of SCoP's extracted from source LLVM, the ISL style relations that we expected from our hand calculated examples are also semantically different. Below, we see an extraction of the memory flow dependence of our

```
Printing analysis 'Polly - Calculate dependences' for region: '%9 => %21'
    in function 'stencil_original':
   RAW dependences:
      [p_0, p_1, p_2] -> { Stmt2[i0] -> Stmt2[o0] : i0 >= 0 and o0 > i0
          and -1 - p_1 + p_2 + i0 <= o0 <= 1 - p_1 + p_2 + i0 and o0 <= -3
          + p_0 }
   WAR dependences:
      [p_0, p_1, p_2] -> { Stmt2[i0] -> Stmt2[o0] : i0 >= 0 and o0 >= -1 +
          p_1 - p_2 + i0 and i0 < o0 <= 1 + p_1 - p_2 + i0 and o0 <= -3 +
          p_0 }
   WAW dependences:
      [p_0, p_1, p_2] -> { }
   Reduction dependences:
      [p_0, p_1, p_2] -> { }
   Transitive closure of reduction dependences:
      [p_0, p_1, p_2] -> { }
Printing analysis 'Polly - Calculate dependences' for region: '%9 => %20'
    in function 'stencil_original':
```

In this case, we are interested in the RAW dependences, which are the value-based flow dependencies. Our own calculations for the source input program, "stencil_original" is a two symbolic bounds ([T, N]) over a two dimensional iteration space, whereas the Polly appears to have three symbolic bounds ([p_0, p_1, p_2]) over a one dimensional iteration space.

memDep1 := [T, N] -> exe[a, b] -> exe[a', b' = 1 + b] : (1 + a + a') mod 2 = 0
and
0 < a < T and
0 < b <= -2 + N and
a' > a and
0 < a' < T ;

memDep2 := [T, N] -> exe[a, b] -> exe[a', b' = -1 + b] : (1 + a + a') mod 2 = 0
and
0 < a < T and
2 <= b < N and
a' > a and
0 < a' < T ;

```
outDep1 := [T, N] ->  exe[a, b] -> exe[a', b' = b] : (a + a') mod 2 = 0 and
0 <= a < T and
0 <= b < N and
a' > a and
0 <= a' < T ;


valFlow := memDep2 - (outDep1 * memDep1);
valFlow;


/* our value based dependence */
valDep := [T, N] ->  exe[a, b] -> exe[a', b' = -1 + b] : (1 + a + a') mod 2 = 0 and
0 < a < T and
2 <= b < N and
a' > a and
0 < a' < T ;
```

While these two dependences may be 'equivalent' in the sense that Polly is optimizing and extracting information form a transformed piece of machine generated code, for our purposes, there is too little continuity between the code that the programmer puts into clang and Polly, and what is actually being analyzed under the hood. During this point in time during our work with Polly, we recognized that this lack of continuity caused by both Polly internals and clang internals made our proposed project much more difficult, and far less accessible than we had anticipated. As a result, we stopped our project and decided to document what we learned about Polly's internals, as well as our experience using different Polly flags.

5 Conclusion and Future Work

While we were unable to implement a working prototype from our proposal, we have several contributions on how individuals looking to produce automatic tiling correctness checkers may want to continue if they are going to use the Polly infrastructure. The crux of the problem that we faced during our experimentation was the lack of continuity between the code we put into Polly, and the information that we got out of Polly. Recall that this was mainly due to the fact that were unable to keep track of the value dependences specific to the input program, and not the pre-optimized LLVM that was output by clang.

From here, we see two different approaches that would help to mitigate this problem for future researches. If researchers are looking to continue to use Polly, it is important that they be able to precisely control when optimizations are applied both from clang and Polly. We recommend that prospective researchers investigate whether clang or another C compiler actually has the ability to produce unoptimized LLVM from source code. If this is possible, it may be possible to pass these unaltered dependencies directly to Polly, although, we suspect that this may be difficult since Polly (rightfully so as an partially automatic compiler) makes many optimizations during execution which can greatly speed up output code. In this sense, it may be very difficult to make use of the Polly infrastructure without serious exploration of the codebase, and a in depth low level understanding of how Polly chooses to optimize programs. If there is a canonical form that preserves original dependencies and is still acceptable by Polly, researchers may have a way forward so long as they are able to accurately use the feature set of Polly.

Another possible option would be to implement a limited "compiler" that only recognized SCoP's for loop bodies. Henry Mohr, who also worked with Dave Wonnacott for his senior thesis, looked into loop tiling extraction from Tiger source code. Creating a miniaturized compiler-style analyzer with a grammar that revolves around SCoP's may allow researchers to accurately control and monitor extracted loop information, such as dependences. Source C code could be translated into LLVM-IR, and then fed into Polly through command line piping, or through direct integration into Polly's codebase. However, as we discussed before, measures would need to be taken to assure that SCoP's are determined semantically in addition to syntactically, as certain loop programs without sufficiently complicated loop dependencies should

be excluded from analysis to prevent unnecessary computation. Since such SCoP detection may not be possible with conventional parsing and grammars, it may be worth investigating how SCoP's are detected in Polly, and applying a similar approach to any new programs used for detecting SCoP's within a more limited and controlled context.

Until automatic compilers become a reality, we hope to find ways to support other researchers with tools that allow them to more readily access forms of code optimization. By using existing methods for calculating and checking value-based flow dependencies, we have show that it is still theoretically possible for a manual, programmer interactive program to be used for flow equivalence checking to be implemented. Furthermore, such a program would only be restrained by symbolic and non-affine tiling - the same current restrictions on state-of-the-art automatic compilers. In terms of future research, we suggest that that research take a further look into finding more accurate methods of SCoP detection and extraction, that would start building the infrastructure for more automatic parallelism. More accurate SCoP detection would mitigate the need for manual tools, as there would be fewer edge cases that would cause compilers to fail to optimize or skip optimizing certain SCoPs.

6 Bibliography

[1] Amdahl, Gene M. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities.” In Proceedings of the April 18-20, 1967, Spring Joint Computer Conference on - AFIPS ’67 (Spring), 483. Atlantic City, New Jersey: ACM Press, 1967. https://doi.org/10.1145/1465482.1465560.

[2] Bandishti, Vinayaka, Irshad Pananilath, and Uday Bondhugula. “Tiling Stencil Computations to Maximize Parallelism.” In 2012 International Conference for High Performance Computing, Networking, Storage and Analysis, 1–11. Salt Lake City, UT: IEEE, 2012. https://doi.org/10.1109/SC.2012.107.

[3] Bertolacci, Ian J., Catherine Olschanowsky, Ben Harshbarger, Bradford L. Chamberlain, David G. Wonnacott, and Michelle Mills Strout. “Parameterized Diamond Tiling for Stencil Computations with Chapel Parallel Iterators.” In Proceedings of the 29th ACM on International Conference on Supercomputing - ICS ’15, 197–206. Newport Beach, California, USA: ACM Press, 2015. https://doi.org/10.1145/2751205.2751226.

[4] Bondhugula, Uday, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. “Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model.” In Compiler Construction, edited by Laurie Hendren, 4959:132–46. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. https://doi.org/10.1007/978-3-540-78791-4$_9$.

[5] Bondhugula, Uday, Albert Hartono, J. Ramanujam, and P. Sadayappan. “A Practical Automatic Polyhedral Parallelizer and Locality Optimizer.” In Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, 101–113. PLDI ’08. New York, NY, USA: ACM, 2008. https://doi.org/10.1145/1375581.1375595.

[6] Grosser, Tobias, Armin Groesslinger, and Christian Lengauer. “POLLY — PERFORMING POLYHEDRAL OPTIMIZATIONS ON A LOW-LEVEL INTERMEDIATE REPRESENTATION.” Parallel Processing Letters 22, no. 04 (December 2012): 1250010. https://doi.org/10.1142/S0129626412500107.

[7] Grosser, Tobias, Hongbin Zheng, and Raghesh Aloor. “Polly - Polyhedral Optimization in LLVM,” n.d., 6.

[8] Gustafson, John L. “Reevaluating Amdahl’s Law.” Commun. ACM 31, no. 5 (May 1988): 532–533. https://doi.org/10.1145/42411.42415.

[9] “How to Manually Use the Individual Pieces of Polly — Polly 11.0-Devel Documentation.” Accessed March 6, 2020.
https://polly.llvm.org/docs/HowToManuallyUseTheIndividualPiecesOfPolly.html.

[10] Irigoin, F., and R. Triolet. “Supernode Partitioning,” 319–29. ACM, 1988. https://doi.org/10.1145/73560.73588.

[11] Mohammadi, Mahdi Soltan, Michelle Mills Strout, Tomofumi Yuki, Kazem Cheshmi, Eddie C. Davis, Mary Hall, Maryam Mehri Dehnavi, Payal Nandy, Catherine Olschanowsky, and Anand Venkat. “Sparse Computation Data Dependence Simplification for Efficient Compiler-Generated Inspectors.” In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2019, 594–609. Phoenix, AZ, USA: ACM Press, 2019.
https://doi.org/10.1145/3314221.3314646.

[12] Patterson, David A., and John L. Hennessy. Computer Organization and Design, Fifth Edition: The Hardware/Software Interface. 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[13] Pugh, William. “The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis.” In Proceedings of the 1991 ACM/IEEE Conference on Supercomputing - Supercomputing ’91, 4–13. Albuquerque, New Mexico, United States: ACM Press, 1991. https://doi.org/10.1145/125826.125848.

[14] Pugh, William, and Evan Rosser. “Iteration Space Slicing and Its Application to Communication Optimization,” October 15, 1998.
http://drum.lib.umd.edu/handle/1903/869.

[15] The Polly Team. “Polly Documentation,” August 8, 2018.
https://polly.llvm.org/docs/index.html.

[16] “Using Polly with Clang — Polly 11.0-Devel Documentation.” Accessed March 6, 2020. https://polly.llvm.org/docs/UsingPollyWithClang.html.

[17] Verdoolaege, Sven. “Counting Afne Calculator and Applications,” n.d., 6.

[18] Wolfe, M. “More Iteration Space Tiling.” In Proceedings of the 1989 ACM/IEEE Conference on Supercomputing - Supercomputing ’89, 655–64. Reno, Nevada, United States: ACM Press, 1989. https://doi.org/10.1145/76263.76337.

[19] Wonnacott, David. “Achieving Scalable Locality with Time Skewing.” International Journal of Parallel Programming 30, no. 3 (June 1, 2002): 181–221. https://doi.org/10.1023/A:1015460304860.

[20] Wonnacott, David G. “A Retrospective of the Omega Project,” n.d., 13.

[21] Wonnacott, David G, Michelle Mills Strout, and Fort Collins. “On the Scalability of Loop Tiling Techniques,” n.d., 9.