Improving High Level Synthesis Optimization Opportunity Through Polyhedral Transformations

Wei Zuo[2,5], Yun Liang[1]*, Peng Li[1], Kyle Rupnow[3], Deming Chen[2,3] and Jason Cong[1,4]
[1]Center for Energy-Efficient Computing and Applications, School of EECS, Peking University, China
[2]University of Illinois, Urbana-Champaign, USA
[3]Advanced Digital Science Center, Singapore
[4]Computer Science Department, University of California, Los Angeles, USA
[5]School of Information and Electronics, Beijing Institute of technology,China
{weizuo,dchen}@illinois.edu,{ericlyun,peng.li}@pku.edu.cn
k.rupnow@adsc.com.sg,cong@cs.ucla.edu

ABSTRACT

High level synthesis (HLS) is an important enabling technology for the adoption of hardware accelerator technologies. It promises the performance and energy efficiency of hardware designs with a lower barrier to entry in design expertise, and shorter design time. State-of-the-art high level synthesis now includes a wide variety of powerful optimizations that implement efficient hardware. These optimizations can implement some of the most important features generally performed in manual designs including parallel hardware units, pipelining of execution both within a hardware unit and between units, and fine-grained data communication. We may generally classify the optimizations as those that optimize hardware implementation within a code block (intra-block) and those that optimize communication and pipelining between code blocks (inter-block). However, both optimizations are in practice difficult to apply. Real-world applications contain data-dependent blocks of code and communicate through complex data access patterns. Existing high level synthesis tools cannot apply these powerful optimizations unless the code is inherently compatible, severely limiting the optimization opportunity.

In this paper we present an integrated framework to model and enable both intra- and inter-block optimizations. This integrated technique substantially improves the opportunity to use the powerful HLS optimizations that implement parallelism, pipelining, and fine-grained communication. Our polyhedral model-based technique systematically defines a set of data access patterns, identifies effective data access patterns, and performs the loop transformations to enable the intra- and inter-block optimizations. Our framework automatically explores transformation options, performs code transformations, and inserts the appropriate HLS directives to implement the HLS optimizations. Furthermore, our framework can automatically generate the optimized communication blocks for fine-grained communication between hardware blocks. Experimental evaluation demonstrates that we can achieve an average of 6.04X speedup over the high level synthesis solution without our transformations to enable intra- and inter-block optimizations.

*Corresponding Author

FPGA'13, February 11–13, 2013, Monterey, California, USA.

Categories and Subject Descriptors

B.5.2 [**Hardware**]: [Design Aids] — automatic synthesis

General Terms

Algorithm, Design, Performance

Keywords

Polyhedral, High Level Synthesis, FPGA

1. INTRODUCTION

FPGAs have long been adopted for computation acceleration, especially in domains that also demand power and energy efficient computing. Their highly flexible architecture enables significant optimization opportunities. Designers can implement fine-grained computation units, highly parallel architectures, fine-grained pipelining of computation units, efficient communication structures and customized memory and compute unit partitioning. However, the flexibility that is a strength for optimization opportunities is also a challenge for efficient programmability. FPGA implementation remains a significant challenge for prospective users — manual design at register transfer level (RTL) is error-prone and difficult to debug. Such manual design often takes weeks and months in comparison to the hours or days to implement an algorithm in software. Furthermore, efficient manual design typically requires significant specialized knowledge — efficient FPGA implementations must consider architecture-specific parameters and implement functions for efficient mapping to the FPGAs' fixed-size allocation quota (LUT, BRAM, DSP, ...).

High level synthesis (HLS) seeks to address these problems. HLS offers automated translation from high level languages (e.g., C,C++, SystemC, Haskell and CUDA) to register transfer level (RTL) implementations to reduce the design effort, automated optimization to reduce the requirement of design knowledge, and FPGA-specific mapping to automate low-level optimization choices. Thus, HLS promises to be a critical bridging technology that offers the latency and power/energy benefits of FPGA-based hardware acceleration at the design effort (and expertise) of software development. State-of-the-art HLS tools cover a wide range of input source code and achieve high-quality results [8].

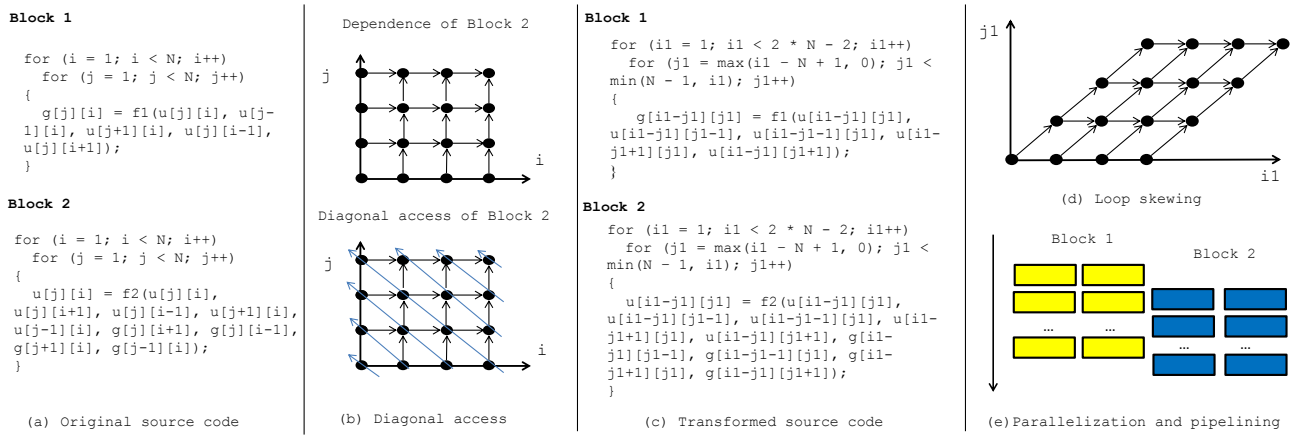


Figure 1: An example of two data-dependent blocks. In sub-figure (e), we assume the parallelization degree is 2.

These tools have achieved significant improvement; however, recent studies show that although these tools can offer high quality designs for small kernels, there is still a significant performance gap between HLS and manual design for real-world complex applications [23, 17, 10]. For example, [23, 17] demonstrated a 40X difference between HLS and the manual design for a high-definition stereo matching implementation. Small kernels (often used as simple HLS benchmarks) contain a single block (a loop nest), but real-world applications often contain many data-dependent blocks that communicate through complex data access patterns. Whereas a video processing kernel may be a single nested loop performing a median filter, a real application would employ a sequence of data-dependent blocks, including blocks such as image up/down sampling, cost aggregation, and energy minimization. Video processing applications are often structured so that each processing step is a loop nest (block) that operates on array inputs and produces array outputs such that block $i+1$ reads the array variables written by block i. As seen in efficient manual RTL implementations of these algorithms, it is crucial to minimize the communication granularity, pipeline the data-dependent blocks, and duplicate compute units to improve both throughput and latency. However, existing HLS tools fail to enable intra-block parallelization and inter-block pipelining when the data access patterns are complex [23, 17] — and although advanced commercial tools *support* these optimizations, they are not always able to efficiently identify the opportunity and transform source code to enable such optimizations.

Code transformations to enable these important and powerful optimizations are thus critical for HLS tools. One of the primary goals of HLS is to reduce design effort; thus, HLS tools must efficiently support a variety of code — including source code written by software engineers with little hardware expertise, and software not written for use with HLS. Software programmers choose data access patterns based on properties such as intuitive ordering, cache locality, and code modularity. Even software written for HLS may still contain loops that demand transformation; efficient inter-block optimization may demand non-intuitive iteration ordering such that even experienced hardware designers may have difficulty seeing an intuitive ordering of blocks that enables optimization. Thus, it is quite natural that the default data access pattern does not support intra-block parallelization or inter-block pipelining. Nevertheless, data access patterns can be altered via loop transformations such as permutation, reverse, and skewing [24]. Using these transformations, we can enable intra-block parallelization by reordering loop iterations so that successive iterations are independent; similarly, we can enable inter-block pipelining by ensuring that block i produces data in the order that block $i+1$ consumes data (by transforming block i, block $i+1$ or both). Both intra-block parallelization and inter-block pipelining are currently supported by existing HLS tools. The problem is rather that these optimizations cannot always be enabled with the default data access patterns. Thus, the goal of this work is to enable efficient integrated use of intra- and inter-block optimizations through loop transformation.

Motivating Example. We illustrate the concept of intra- and inter-block optimization using the *Denoise* [11] application in medical imaging (Figure 1). The source code in Figure 1 (a) is modified to highlight the data access pattern, with computation arithmetic omitted for clarity. The application is composed of two blocks (loop nests), where the first block writes to the g array that is subsequently read by the second block. Both blocks are representative of stencil codes where the array update operation follows a fixed input dependence pattern that reads neighboring elements. In this example, only the second block has data-dependence between loop iterations, as shown in Figure 1 (b), and the first block writes the data array g in the same order that the second block consumes array g (column order). Thus, in this example, intra-block parallelization is available in the first block and inter-block pipelining is available by default between the blocks, but a transform is required to enable parallelization in the second block. However, note that if the second block is transformed to enable parallelization, then the first block must also be transformed in order to retain the opportunity for inter-block pipelining.[1]

Table 1: Comparison of different implementations

	Implementation	Cycles	Frequency	Speedup
1	w/o transform, w/o opt	5408	160MHz	1
2	w/o transform, w/ opt	1809	182MHz	3.40
3	w/ transform, w/ opt	250	230MHz	31.09

In Table 1 we can see the performance in cycles and speedup over the original source for this example. By default, the original source code supports some optimization, as seen in the second implementation — inter-block pipelining improves the performance, but the latency of the individual blocks become the performance bottleneck. Note that in the second implementation, we do not perform partial parallelization (e.g., parallelize the first block only) because if the throughputs are not matched, the buffering between

[1]For efficient implementation of inter-block pipelining, we also implement memory partition optimization and customized communication blocks which will be discussed in section 3.3.

the blocks would not be feasible. In the third implementation, the loop transformation is used to alleviate the dependence problem and retain the opportunity for inter-block pipelining. In Figure 1 (c) and (d), we perform loop skewing on both blocks to traverse the loop in a diagonal fashion from the bottom-right to top-left, as shown in Figure 1 (b). This loop skewing both preserves the data dependence and enables parallelization of the inner loop because the iterations on the same diagonal line are independent. The first block can also use the loop skewing transformation safely because it does not have any data dependencies. The optimized execution schedule is shown in Figure 1 (e). This transformation significantly improves the speedup opportunity from about 3X to 31X speedup. In this case, some optimizations were supported by default, but there was still significant speedup opportunity by ensuring that both optimizations were enabled. In the experiments section, we will demonstrate that some source codes support neither parallelization nor pipelining by default, but loop transformations can enable both intra-block parallelization and inter-block pipelining.

As this brief motivation discussion demonstrates, real-world applications commonly contain multiple data-dependent blocks with communication through complex data access patterns. For such applications, the powerful optimization functions of existing high level synthesis tools may not be supported by default due to the data access patterns. In this paper we develop an integrated method to model, analyze, and transform block data access patterns to support these important and powerful optimizations. Using polyhedral models [24, 13, 20, 6, 5], we represent the loop iteration space, dependencies, data access patterns, and loop transformations in a linear algebraic form. For loops amenable to this algebraic representation, the polyhedral model allows us to analytically determine valid loop transformations that best support both intra-block parallelization and inter-block pipelining. This paper advances the state-of-the-art of high level synthesis with

- An automated polyhedral model-based framework that systematically identifies effective access patterns and applies appropriate loop transformations that enable intra- and inter-block optimizations.
- An automated framework to generate communication interfaces between blocks.

We demonstrate that our automated framework achieves an average of 6.04X speedup over HLS with all available optimizations but without transformations to enable.

This paper is organized as follows. Section 2 briefly provides the background of the polyhedral model and introduces the useful notation. Section 3 presents our framework, which defines data access patterns, identifies data access patterns for effective optimizations, and discusses the implementation of fine-grained data communication modules. Section 4 presents experimental results. Section 5 discusses related work, and conclusions are presented in Section 6.

2. BACKGROUND AND NOTATION

In this section we briefly introduce the polyhedral model and the notation that we will use in this paper. A detailed description of polyhedral models can be found in [5, 6, 13, 20]. In this work we are using the polyhedral model to consider data access patterns for communication between sets of loop nests, and to optimize this communication ordering in order to perform fine-grained communication and enable parallelization within a loop nest and pipelining between loop nests through loop transformation. In particular, in this work we consider programs that consist of a sequence of data-dependent blocks, where each block is a loop nest containing multiple statements, and each statement may have multiple accesses to data arrays.

2.1 Polyhedral Model

Polyhedral models can be used to represent execution information of a program's loop nests, such as the loop iteration domain, statement/iteration dependencies, array access functions, and scheduling functions (execution order).

DEFINITION 1 (**Polyhedron**). *The set of all vectors $\vec{x} \in Z^n$ such that $A\vec{x} + \vec{b} \geq 0$, where $A \in Z^{m \times n}$ and $\vec{b} \in Z^m$.*

Each row of A represents a half-space that limits $A\vec{x} + \vec{b}$ to non-negative values, and the *polyhedron* is the intersection of the m half-spaces represented by the m rows of A. A bounded *polyhedron* is a *polytope*.

DEFINITION 2 (**Iteration Vector and Domain**). *The iteration vector $\vec{i}$ of a loop nest represents an execution instance of the loop nest. $\vec{i}$ contains the values of the loop indices of all the surrounding loops. The iteration domain D_L of loop nest L is the set of all iteration vectors that satisfy the loop-bound constraints. Because an iteration domain is bounded by these loop-bound constraints, it is a polytope.*

DEFINITION 3 (**Schedule Function**). *Given a m-dimensional loop nest, a d-dimensional ($1 \leq d \leq m$) schedule $\boldsymbol{F}(\vec{i})$ is defined as*

$$\begin{aligned} \boldsymbol{F}(\vec{i}) &= \mathrm{S}\vec{i} + \vec{o} \\ &= \begin{pmatrix} C_{11} & C_{12} & \dots & C_{1m} \\ C_{21} & C_{22} & \dots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{d1} & C_{d2} & \dots & C_{dm} \end{pmatrix} \vec{i} + \begin{pmatrix} C_{10} \\ C_{20} \\ \vdots \\ C_{d0} \end{pmatrix} \end{aligned}$$

where $\mathrm{S} \in Z^{d \times m}$, $\vec{o} \in Z^d$, and $C_{ij} \in Z$. The schedule function takes the iteration vector $\vec{i}$ as input and produces an ordering of the loop iterations using the matrix S and an offset vector ($\vec{o}$).

Thus, the schedule function creates a particular ordering of iterations by mapping each iteration in the domain to a timestamp, where mapping m dimensions into fewer (d) timestamps implies that some of the iterations can be performed in parallel. The schedule function has the additional benefit that the timestamp ordering also corresponds to a lexicographic ordering. For example, for iteration vectors $\vec{i_1}$ and $\vec{i_2}$, $\vec{i_1}$ is scheduled before $\vec{i_2}$ if $\mathbf{F}(\vec{i_1}) < \mathbf{F}(\vec{i_2})$.

DEFINITION 4 (**Transformation Function**). *In the polyhedral model, the transformation function is represented as a sequence of schedules applied to perform the transformation.*

In theory, any loop transformation can be represented in the polyhedral model. In this paper we focus on a set of uni-modular loop transformations [24], including loop reverse (e.g., reverses the traverse direction), loop permutation (e.g., interchange two loops), and skewing (e.g., make the inner-loop bounds dependent on the outer-loop bounds) and their composite transformations. The schedule function/execution order of iterations of a loop nest can be altered through loop transformations.

Figure 2 illustrates the polyhedral model using a concrete example. The original source code that consists of two data-dependent blocks (loop nests) is shown in Figure 2 (a). The first block writes to array B, and the second block subsequently reads from it. Without transformation, each individual block can be parallelized, but

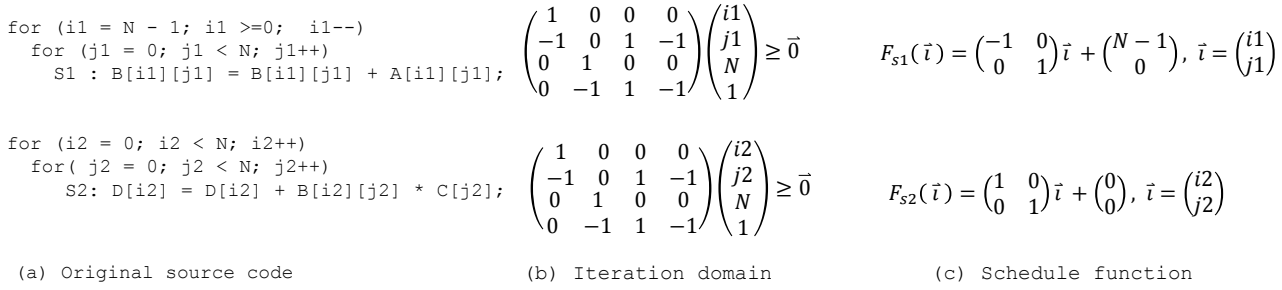


Figure 2: An example of polyhedral representation.

their execution can not be overlapped through pipelining as they access array B in different order. In Figure 2 (b), we show the iteration domain (polytope) of the two blocks which establishes the lower- and upper-bounds for each loop dimension. In Figure 2 (c), we show the scheduling function that corresponds to the original source code for each of the loop nests. In order to transform the original scheduling function to achieve a desired data access pattern, we can define a desired data access pattern and derive the necessary loop transformations; a proof that this derivation is always possible is shown in Section 3.

As described above, the polyhedral model (including iteration domain, iteration vector, scheduling function and transformation function) describes the iteration domain and order for any loop nest that has all array accesses as affine expressions of loop indices. In addition, dependency relations between iterations in the program can be represented in the polyhedral model. Two iterations of a loop nest (or two instances of blocks) are dependent if they access the same array locations, and at least one of the accesses is a write operation. These dependencies can be represented by additional rows in the iteration domain to establish constraints between the loop iterations. Loop transformations are valid only if they also preserve the additional program dependency constraints.

3. METHODOLOGY

Our integrated intra- and inter-block optimization framework takes a data-dependent multi-block program as input, and performs three steps, as shown in Figure 3. The goal of our optimization is to minimize the overall latency (maximize the performance speedup). First, we systematically define a set of data access patterns, classify them, and derive the associated loop transformations (Section 3.1). Next, for each loop transformation that validly preserves data-dependencies, we estimate the performance improvement (Section 3.2) and choose the best estimated performance. The performance estimation models both intra- and inter-block speedup and associated implementation overheads. The intra-block parallelization degree is determined by the resource usage of the program and the available resource on the implementation platform. Finally, for the chosen transformation, we automatically perform the loop transformations, insert high level synthesis directives, and generate the communication blocks that interface the data-dependent blocks. If the communication block is a simple FIFO, we automatically insert FIFO high level synthesis directives; when the communication interface requires multiple reads or a stencil pattern, we automatically customize the communication blocks (Section 3.3). The final output of the flow is an optimized RTL design.

Note that in prior works [5, 6, 13, 18, 20, 24], polyhedral models consider data access patterns for external memory accesses and loop transformations in order to optimize data localities and maximize parallelism for CPUs. Optimizations for CPU code attempt to minimize memory bandwidth and improve cache behavior. In addition, the transformed CPU code often has complex control flow that is not suitable for efficient FPGA implementation. Thus, the transformation chosen for optimization on CPU platform may be the wrong decision for our HLS optimization.

In this work, we use polyhedral model to define data access patterns for a different objective. We aim to optimize the inter-block communication and enable intra-block parallelization and inter-block pipelining through loop transformation for FPGAs using HLS. For our objective, we model the FPGA-specific features. Therefore, although the candidate data access patterns might be the same as prior work, we apply different transformations in different combinations in order to meet our optimization objective. Although the underlying techniques are all based on polyhedral models, different objectives lead to different ways to select data access patterns and loop transformations.

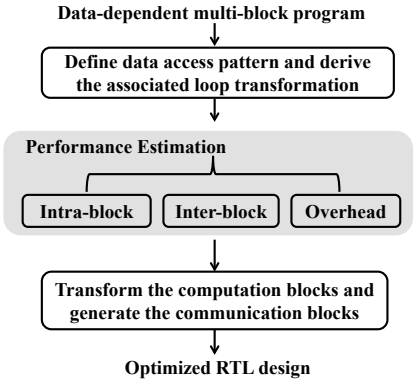


Figure 3: Optimization Framework.

3.1 Classification of Array Access Patterns

We model the array access patterns using the polyhedral model, thus we assume that all array accesses are affine expressions of loop indices and constants. The program inputs are composed of multiple data-dependent blocks where each block contains a single multi-dimensional loop nest. Let us consider a loop nest of dimensionality D that accesses an N-dimensional array. The array access pattern is defined by matrix $\mathbf{M}$ whose size is $N \times D$, where the rows (i) represent the data access pattern in dimension i of the data array, and columns (j) represent the access pattern in the loop level j.

Given the array access pattern $\mathbf{M}$, loop iteration vector $\vec{\imath}$, and constant offset vector $\vec{o}$, the array access vector $\vec{s}$ is defined as

$$\vec{s} = \mathbf{M}\vec{\imath} + \vec{o}$$

$\vec{s}$ is column vector of size N, where each row (i) represents array accesses in dimension i, and the offset vector is a constant offset into that dimension. Figure 4 shows an example of array access

pattern and vector for the writes to array B; similar access patterns and vectors could be derived for the reads from array A.

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

$$M = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad \vec{s} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Figure 4: An example of array access pattern.

In the following, we demonstrate the data access patterns for 2-dimensional arrays and 2-dimensional loop-nests. For ease of illustration, we classify the access patterns below; although, the polyhedral framework treats these access patterns in a uniform way. In the framework, we need to define the candidate access patterns that can be evaluated for the program; in this work we limit the access patterns to the simple set of access patterns below. The polyhedral model can be easily extended to handle a wide variety of additional access patterns, and our work can use any additional access patterns to estimate the performance benefit. We leave the extension to future work.

Now we will describe how to define the array access patterns using **M**. Let

$$\mathbf{M} = \begin{pmatrix} a_1 & b_1 \\ a_2 & b_2 \end{pmatrix}$$

We classify the array access patterns based on the values of a_1, a_2, b_1 and b_2. For array accesses with non-unit loop strides, we perform loop normalization as a preprocessing step so that our analysis can assume unit loop stride.

Column and Reverse Column.

$$\begin{pmatrix} \pm 1 & 0 \\ 0 & \pm 1 \end{pmatrix}$$

The array access vector can be obtained by $M\vec{i}$. Figure 5 shows the four patterns in this category, where the signs of a_1 and b_2 determine traversal direction. For example, with outer loop index i and inner loop index j, if $a_1 = 1$ and $b_2 = 1$, then the outer loop traverses increasing values of i, and the inner loop traverses increasing values of j.

Row and Reverse Row.

$$\begin{pmatrix} 0 & \pm 1 \\ \pm 1 & 0 \end{pmatrix}$$

Similar to column and reverse column, there are four patterns in this category, and the signs of b_1 and a_2 determine the traversal directions.

Diagonal Access. In this category, the loop traverses in a diagonal line fashion. We further divide this into two cases based on the slopes of the diagonal lines.

- $slope \geq 1$.

$$\begin{pmatrix} \pm 1 & N > b_1 \geq 1 \\ 0 & \pm 1 \end{pmatrix}$$

  The array access vector can be obtained by $M\vec{i}$. The slope of the diagonal line is determined by b_1, and the signs of a_1 and b_2 determine the traversal directions. When $b_1 \geq N$, the traversal order reduces to one of the column access orders. Figure 6 shows the four data access patterns for $slope = 1$.

- $slope < 1$.

$$\begin{pmatrix} N > a_1 > 1 & \pm 1 \\ \pm 1 & 0 \end{pmatrix}$$

  The slope of the diagonal line is determined by $\frac{1}{a_1}$, and the signs of a_2 and b_1 determine the traversal directions. When $a_1 \geq N$, the traversal order reduces to one of the row access orders.

All of the array access patterns defined above are unimodular matrices where $|a_1 \times b_2 - a_2 \times b_1| = 1$. Thus, all of these access patterns can be achieved by unimodular loop transformations of the block(s) [24].

Loop Transformation. Loop transformations can change the schedule (e.g., execution order) of loop iterations such that the data access pattern can be changed. Thus, here we derive the loop transformation given a desired data access pattern.

THEOREM 3.1. *The transformation function* $\boldsymbol{T}$ *required for the desired data access pattern* $\boldsymbol{M}_{des}$ *can be obtained by*

$$\boldsymbol{T} = \boldsymbol{M}_{des}^{-1} \boldsymbol{M}_{ori} \boldsymbol{F}_{ori}^{-1}$$

where $\boldsymbol{M}_{ori}$ *and* $\boldsymbol{F}_{ori}$ *are the data access pattern and schedule function of the source code without transformation.*

PROOF. Let $\vec{i'}$ be the loop iterator vector after transformation. Thus, the desired array access vector after transformation is $\mathbf{M}_{des}\vec{i'}$. Let us assume the schedule function and data access pattern of the original code are $\mathbf{F}_{ori}$ and $\mathbf{M}_{ori}$, respectively. Thus, the schedule function after transformation is $\mathbf{TF}_{ori}$. Schedule function maps the original loop iterations to a new ordering of the loop iterations. Thus,

$$\mathbf{TF}_{ori}\vec{i} = \vec{i'}$$

Thus,

$$\vec{i} = \mathbf{F}_{ori}^{-1}\mathbf{T}^{-1}\vec{i'}$$

The data accessed by the iterations before and after transformation is the same

$$\mathbf{M}_{ori}\mathbf{F}_{ori}^{-1}\mathbf{T}^{-1}\vec{i'} = \mathbf{M}_{des}\vec{i'}$$

Thus,

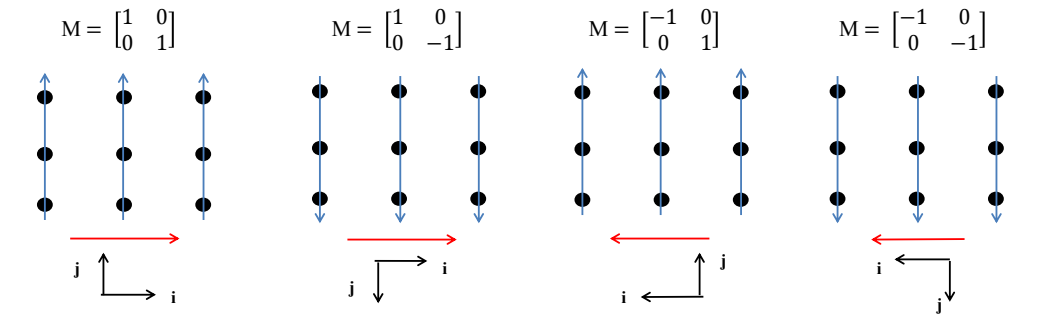
$$\mathbf{T} = \mathbf{M}_{des}^{-1}\mathbf{M}_{ori}\mathbf{F}_{ori}^{-1}$$

□

The data access pattern and loop transformations are all unimodular matrix. For unimodular matrix, we can always derive its reverse matrix.

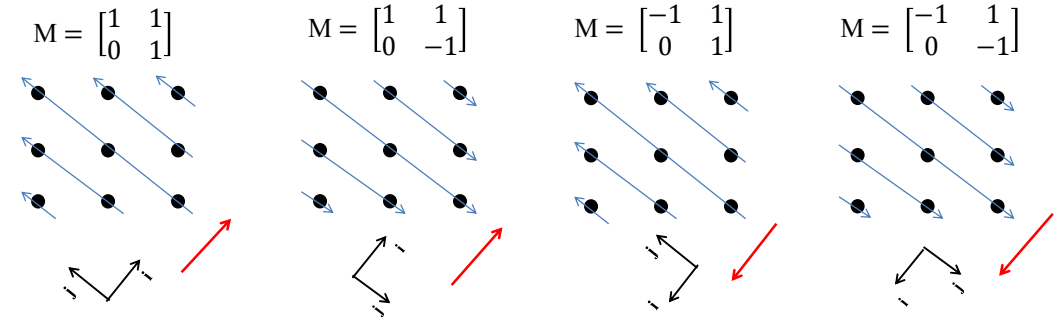
3.2 Performance Metric

In the previous subsection we defined a set of data access patterns that include row, column, and diagonal access with different slopes and directions. We also described how to derive the required loop transformation for a given data access pattern. Our design objective is to maximize the performance speedup. Now, we will discuss the process of evaluating all of the candidate data access patterns and choosing the candidate that maximizes application speedup. To perform this evaluation, we develop a performance metric that combines modeling of both intra- and inter-block speedup and their associated implementation overhead. We define that a program is a sequence of K blocks $\{b_1, b_2, \ldots, b_K\}$, and



(a) left to right (i), bottom to up (j) (b) left to right (i), top to down (j) (c) right to left (i), bottom to up (j) (d) right to left (i), top to down (j)

Figure 5: An example of column access pattern. There are 4 patterns with different traverse directions.



(a) bottom-left to top-right (i), bottom-right to top-left (j) (b) bottom-left to top-right (i), top-left to bottom-right (j) (c) top-right to bottom-left (i), bottom-right to top-left (j) (d) top-right to bottom-left (i), top-left to bottom-right (j)

Figure 6: An example of diagonal access pattern with $slope = 1$. There are 4 patterns with different traverse directions.

each block b_i has a d_i dimensional loop nest that accesses an n_i dimensional data array. In this work we restrict that the loop nest dimensionality and data-array dimensionality are equal for all sets of communicating blocks in the program, so we denote the loop nest dimensions and data array dimensions as N. The latency in clock cycles of each block b_i without intra-block optimization is lat_i. If there is any control flow between the blocks, we consider the worst-case path; thus, the total baseline program latency in clock cycles (without any optimization) is the simple sum of the blocks' latencies

$$lat_{base} = \sum_{i=1}^{i=K} lat_i$$

lat_i for each block is estimated by performing block-wise high level synthesis, and using the HLS-generated performance estimates[2]. When no optimizations are applied, we have verified that these estimates are accurate by comparing them with post-synthesis simulations. However, the HLS performance estimates can be inaccurate for modeling the parallelism, especially because the inter-block pipelining hides execution latency. Therefore, we develop an alternative performance metric that can estimate this optimization effect. Note that our performance estimation is based on clock cycles only. By default, we apply intra-block pipelining (e.g., loop pipelining within a block by setting initiation interval (II)) for all the blocks and set the same target period for all the implementations. Thus, the clock period tends to be similar across different implementations, and we ignore the effect of clock period in our estimation.

Let **P** be the set of all candidate data access patterns. Given $p \in \mathbf{P}$, we can estimate the performance of each block by the intra-block parallelization factor, and then the performance of the entire program with the inter-block pipelining that overlaps the blocks' execution. In this work, we target inter-block pipelining, and correspondingly, it always makes sense to parallelize each block to the same factor to match the blocks' throughput and minimize inter-block buffering. Thus, we can simplify the program performance estimate in clock cycles as follows

$$lat_p = \frac{lat_{base}}{S_p^{intra} \times S_p^{inter}} + cost_p$$

where S_p^{intra} and S_p^{inter} represent the intra- and inter-block speedup (discussed next), respectively and $cost_p$ represents the implementation overhead.

For each block, we can fit multiple data processing pipelines for parallel processing. In high level synthesis, duplicate data processing pipelines are implemented by unrolling the inner loop. As we discussed above, we want to match the blocks' parallelism degree; therefore we search for the maximum unrolling factor where all blocks in the program can be unrolled to the same factor. For block b_i, we use R_i to denote its resource usage. R_i is a 4-tuple that represents its resource usage in *FFs*, *LUTs*, *BRAMs*, and *DSPs*. Then the total communication cost (in resources) between block i and block $i+1$ is estimated as another 4-tuple $Comm_i$, where we use the HLS estimates for R_i and $Comm_i$. We have observed and experimentally validated that the HLS resource usage estimates correlate with actual resource after logic synthesis, although they tend to be conservatively high estimates. In particular, the HLS estimates for FF resource use and LUT resource use tend to be high. As we scale the unroll factor, this overestimate is compounded due to underestimation of resource sharing and LUT/FF packing into Slices. Therefore, we empirically determined another correlation factor tuple α that represents this scaling; unlike the communication factor, we use the same α factor for all blocks in a design and

[2]We use the commercial AutoPilot [25] HLS tool to estimate lat_i.

for all designs in our benchmark set. In total, we predict the actual resource usage as follows

$$(\sum_{i=1}^{i=K} R_i + Comm_i) \times \alpha \times par$$

where par represents the intra-block parallelization degree.

Then, given the fixed resource budget of the FPGA, we can derive the maximally allowed parallelization degree, Max_{par}. We define S_p^{intra} as follows

$$S_p^{intra} = \begin{cases} Max_{par} & \text{pattern } p \text{ enables parallelization} \\ & \text{for all the blocks} \\ 1 & \text{otherwise} \end{cases}$$

Similarly, for the inter-block pipelining factor S_p^{inter}, if we can transform all of the blocks to follow the same data access pattern for inter-block communication, then we can fully enable inter-block pipelining. Therefore, we define S_p^{inter} as follows

$$S_p^{inter} = \begin{cases} K & \text{pattern } p \text{ enables pipelining} \\ 1 & \text{otherwise} \end{cases}$$

where K is the number of blocks in the program. Note that we ignore the pipelining fill and drain overhead in the above estimation.

Finally, the implementation of the access patterns may incur additional overhead, which can significantly affect the program's performance estimate. Both row ($slope = 0$), column ($slope = \infty$) and their reverses are easy to implement without any overhead, but diagonal access patterns require loop skewing, as shown in Figure 1. Therefore, the inner-loop iteration domains are a function of outer-loop index values. When $slope = 1$, we can use min and max operations to bound the inner-loop iteration domains, as shown in Figure 1 (c); but with $1 < slope < N$ or $0 < slope < 1$, we must use a combination of min, max, $ceil$ and $floor$ functions to bound the inner-loop iteration domains. As discussed previously, we apply intra-block pipelining (e.g., loop pipelining within a block) for all the blocks by default to improve the performance. However, to enable intra-block pipelining, the inner-loop bounds have to be constants for HLS tools. Thus, for diagonal access patterns, we use the maximum loop bound (N) for the inner loop but add extra if-else conditions to filter out the false loop iterations. The if-else statement compares the current inner-loop index with its domain and executes the loop body only if the condition is true. The if-else comparison to filter out false loop iterations takes extra cycle and the number of false iterations depends on the domain of the outer loop. When $slope > 1$, the domain of the outer loop increases linearly with the $slope$; when $slope < 1$, the domain of the outer loop increases linearly with $\frac{1}{slope}$. Hence, the performance overhead $cost_p$ is defined as follows

$$cost_p = \begin{cases} 0 & slope = 0 \text{ or } slope = \infty \\ C \times slope & 1 \leq slope < N \\ C \times \frac{1}{slope} & 0 < slope < 1 \end{cases}$$

where C is a constant.

In our formulation, the intra-block parallelization and inter-block pipelining are performed globally as a coarse-grained optimization. It is possible that a fine-grained optimization that selects different parallelization and pipelining parameters among different sets of blocks and/or communication paths could yield better performance. We expect that this could be implemented by applying this technique separately to subsets of program blocks with additional refinement to the performance model and buffers between the blocks; we leave these tasks for future work.

Pattern Selection. As shown previously, there are multiple data access patterns that include row, column and diagonal patterns with a variety of slopes and traversal directions. With high-dimension loop nests, this can be a large number of candidate patterns; however, we can easily prune the search space using the data dependencies to prune candidates that would violate dependencies. For stencil applications, the update of one data item depends on its neighbors in the surrounding stencil window, which is usually small compared to the size of the entire array. For diagonal access, we only need to focus on the data access pattern with $\frac{1}{n} \leq slope \leq n$, where n is the stencil window dimension. For all remaining slope values ($n < slope < N$ and $\frac{1}{N} < slope < \frac{1}{n}$), they are equivalent to $slope = n$ or $slope = \frac{1}{n}$ in terms of the traversal order of data items in the stencil window, except that they incur additional overhead in implementing the loop bounds. This traversal order is the only factor that affects the ability to apply intra-block parallelization and inter-block pipelining. Using these dependencies to prune the list, it is in practice feasible to estimate performance for each of the candidates and choose the best.

Note that we only explore a subset of legal polyhedral transformations in this work. In theory, it is also possible to select the optimal pattern following the polyhedral optimization flow in [20] where all the FPGA-specific features discussed above including resource modeling and performance metrics have to be encoded as constraints and cost functions for the optimization problem. In practice, our pattern selection solution runs very fast for all the tested benchmarks.

3.3 Implementation

Our framework is an automatic flow consisting of three logical steps: we automatically transform source code, use HLS tools to enable optimizations and synthesis, and generate FIFO interfaces between computation and communication blocks. If the communication block requires a large communication buffer and a complex multi-read communication interface, we automatically insert customized communication blocks.

For the first step, we integrate our framework into PoCC polyhedral framework [1]. Our framework defines data access patterns, estimates performance, selects desired patterns based on performance metrics and performs loop transformations. The PoCC framework is at the C-code source level; we have also modified the framework to automatically produce source code compatible with our chosen HLS tool, AutoPilot [25][3].

Next, we use the AutoPilot HLS tool to enable optimizations and synthesize the computation blocks. AutoPilot provides a set of directives for optimization, including $loop\_unrolling$ and $pipeline$ for the intra-block parallelization and pipelining, and the $dataflow$ and AP_FIFO directives for inter-block pipelining (with communication through a FIFO interface). AutoPilot will execute unrolled loop iterations in parallel if there are no data dependencies between the iterations. Similarly, AutoPilot will overlap execution of data-dependent blocks if they use the same access pattern and can thus communicate through a FIFO interface. However, for the blocks that have multiple accesses to an array, either due to algorithm (e.g., stencil codes read a pattern of neighboring data) or parallelization (e.g., multiple iterations execute and access data in parallel), memory bandwidth bottleneck often prevents the design from reaching the expected intra-block parallelism and inter-block pipelining. To alleviate the memory bottleneck, we implement the memory partition technique [16] that partitions the arrays into several banks and enables more array accesses per clock cycle.

[3] AutoESL was acquired by Xilinx; AutoPilot is now part of Vivado HLS.

It is important to emphasize again that although AutoPilot provides these directives for intra- and inter-block optimizations, these optimizations can not always be applied with the default data access patterns and AutoPilot is unable to identify the optimization opportunities to enable them through transformations. Thus, the importance of this work is not just that we generate the AutoPilot directives that are already available, but that we automatically improve the number of situations in which the directives can be used.

In the first two steps, we can handle applications with simple inter-block communication patterns that can be transformed to use the FIFO interface automatically. However, some communication patterns are more complex; a block may perform multiple data reads per inner-loop iteration. In these situations, we need an optimized communication block. Particularly, we automatically insert a reuse buffer [15] that stores data that will be temporally reused. The implementation of the reuse buffer can use multi-port BRAMs or registers depending on resource and buffer sizes, as shown in Figure 7.

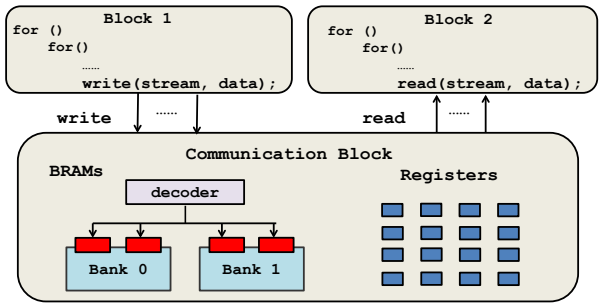


Figure 7: Communication block.

4. EXPERIMENTAL EVALUATION

In this section, we present our experimental results using a set of real-world applications that contain multiple data-dependent blocks and communicate through complex data access patterns. We first discuss the experiments setup and evaluated benchmarks. Then, we show the performance improvement of our proposed framework.

4.1 Experiments Setup

For our experiments, we use a set of benchmarks from PolyBench 3.0 [1] and some real-world applications from [11]. Table 2 describes the benchmark details.

Our framework is based on the polyhedral compiler infrastructure PoCC 1.1 [1]. PoCC is a source-to-source compiler that includes a set of tools for polyhedral compilation. It extracts the polyhedral intermediate representation at source code level. We modify PoCC to define data access patterns, evaluate FPGA architecture-specific performance, perform loop transformations and code generation. Finally, PoCC also provides libraries for program dependencies checking.

The output of our modified PoCC is transformed C code with AutoPilot pragmas to enable the HLS optimizations. Then, we use the AutoPilot HLS tool version 2011.3 to synthesize the transformed C code into Verilog RTL. As previously discussed, AutoPilot supports intra- and inter-block optimizations. We automatically insert directives into the configuration file to enable these optimizations. The target FPGA platform is Xilinx-Virtex-6 LX75T. We synthesize the RTL generated from AutoPilot using Xilinx ISE 13.1 and gather area and clock period data. To compute the operating frequency, we round down the operating frequency determined by the synthesis report's achievable clock period to an integer. The clock cycles are collected through simulation using Modelsim 6.1. Finally, we compute latency using the operating frequency and clock cycles and compute speedup using the latency value.

Table 2: Benchmarks

Benchmark	Description
Deconv	Image Rician Deconvolution [11]
Denoise	Image Rician Denoising [11]
Seg	Image Segmentation [11]
Seidel	Seidel stencil computation [1]
Jacobi	Jacobi stencil computation [1]

4.2 Performance Improvement

We compare the performance of three different implementations. The first implementation is the baseline — it is the original source code without using intra-block parallelization or inter-block pipelining optimization. The second implementation is an improved version — it applies the intra-block parallelization and inter-block pipelining optimizations to the original source code when supported without code transformation. The first and second implementations do not require source code transformation. Although the second implementation tries to use optimizations, they cannot be enabled if the default data access pattern does not support it. The third implementation is our proposed implementation — it transforms the source code and then enables the intra- and inter-block optimizations. Note that by default, we apply intra-block pipelining within each individual block for all the implementations and benchmarks.

Performance Comparison. Table 3 describes the clock cycles, resources and achieved frequencies of the three implementations for all the benchmarks. Table 4 presents the details of the chosen data access patterns and enabled optimizations with and without transformation for all the benchmarks.

Compared to the baseline design, the second and third implementations improve the latency in clock cycles at the expense of more resource usage using the pipelining and parallelization optimizations. This demonstrates that with greater opportunity for intra-block parallelization and inter-block pipelining, HLS tools can effectively use additional FPGA resources. The latency speedup is shown in Figure 8. The speedup is normalized to the baseline implementation. Compared to the baseline, the second implementation (without transformation, with optimization) achieves an average of 4.89X speedup. Our proposed implementation (with transformation, with optimization) achieves further speedup by enabling more optimizations through polyhedral transformations. Our proposed implementation achieves an average of 29.59X speedup over the non-optimized source code and 6.04X speedup over code that is optimized but not transformed to enable optimizations. The average speedup is computed using geometric mean.

The performance improvement is from three-fold: first, the intra-block parallelization improves computation latency within blocks; second, the inter-block pipelining improves computation latency by overlapping the execution of blocks; and third, our memory and communication block optimization improves the memory bandwidth. We again use the example of the *Denoise* benchmark to demonstrate these points; the second implementation employs inter-block pipelining in conjunction with memory partition. These two optimizations improve clock cycles by overlapping the execution of two blocks and allowing multiple memory accesses per clock cycle. Without code transformation, the intra-block parallelization is available for the first block but not the second block as shown in Figure 1. We do not perform partial parallelization (e.g., parallelizing the first block only) together with inter-block pipelining

Table 3: Performance and resource comparison of different implementations

Benchmark	Implementation	Cycles	LUT	FF	DSP	BRAM	Frequency(MHz)
Deconv	w/o trans, w/o opt	5408	3234	948	24	48	151
	w/o trans, w/ opt	1809	6433	2650	24	5	182
	w/ trans, w/ opt	257	13819	13826	108	17	182
Denoise	w/o trans, w/o opt	5408	3266	948	24	5	160
	w/o trans, w/ opt	1809	6503	2672	24	5	182
	w/ trans, w/ opt	250	13817	13824	108	17	230
Seg	w/o trans, w/o opt	9864	3735	1202	30	24	117
	w/o trans, w/ opt	9864	3735	1202	30	24	117
	w/ trans, w/ opt	500	48796	9560	216	34	156
Seidel	w/o trans, w/o opt	64803	1400	891	2	2	103
	w/o trans, w/ opt	1818	13375	6626	32	6	134
	w/ trans, w/ opt	1130	47402	20040	96	14	134
Jacobi	w/o trans, w/o opt	5373	5563	1890	3	16	101
	w/o trans, w/ opt	1439	39430	18832	64	10	134
	w/ trans, w/ opt	482	38877	18664	64	10	133

Table 4: Details of optimizations and data access pattern

Benchmark	Optimizations		Selected Data Access Pattern
	w/o trans	w/ trans	
Deconv	inter-block pipeline	inter-block pipeline & intra-block parallel	diagonal (slope = 1)
Denoise	inter-block pipeline	inter-block pipeline & intra-block parallel	diagonal (slope = 1)
Seg	none	inter-block pipeline & intra-block parallel	diagonal (slope = 1)
Seidel	inter-block pipeline	inter-block pipeline & intra-block parallel	diagonal (slope = 2)
Jacobi	intra-block parallel	inter-block pipeline & intra-block parallel	row

because if the throughputs are not matched then it would require a buffer size proportional to the total data size which is not feasible in general. The third implementation transforms the code to a diagonal access pattern. Then, in addition to inter-block pipelining and memory customization, it enables intra-block parallelization for both blocks that allows multiple iterations to execute simultaneously. The third implementation maximizes resource use such that each block is parallelized to the same degree and fits in the Virtex 6 LX75T, which has 46240 LUTs, 92480 FFs, 288 DSPs, and 312 18Kb BRAMs. For *Denoise*, the intra-block parallelization degree is 9, given the resources on FPGA. As a result, our implementation achieves 31.09X and 9.14X speedup over the first and second implementations, respectively.

In addition, our optimization produces a side-benefit of improved operating frequency. As stated earlier, we assume that the clock period is not affected by our transformations and make no specific effort to improve clock period. However, because our technique implements simplified memory and communication interfaces, we also have the benefit of reducing the complexity of addressing and communication structures. Therefore, we also improve the design's critical path and get a corresponding modest improvement in operating frequency as shown in Table 3.

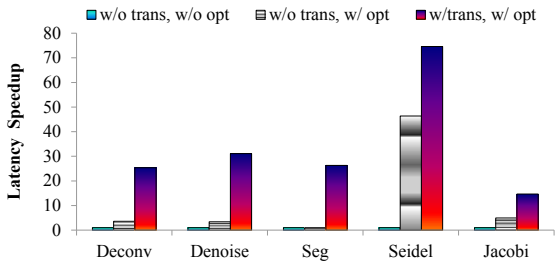


Figure 8: Latency speedup comparison.

Data Access Pattern and Optimization. For benchmarks *Denoise*, *Deconv* and *Seidel*, without transformations inter-block pipeline can be enabled as different blocks communicate in the same order; but intra-block parallelization cannot be enabled with the default data access order. Our implementation transforms them into diagonal access order with different slopes, which enables both intra- and inter-block optimization. The slope of the data access order is chosen based on the performance metric we developed in section 3.2. In particular, we choose $slope = 2$ for *Seidel* as it allows simultaneous execution of the iterations on the same diagonal line and retain inter-block pipelining. For benchmark *Seg*, without transformation neither intra-block parallelization nor inter-block pipelining can be enabled. Thus, the first and second implementation are the same as shown in Table 3. Our implementation transforms it into diagonal access pattern ($slope = 1$), which enables both intra- and inter-block optimization. For benchmarks *Jacobi*, without transformation intra-block parallelization is available as there are no dependencies among iterations, but inter-block pipelining cannot be enabled as two blocks produce and consume data in different orders. Our implementation transforms it to enable both intra- and inter-block optimization. For all the benchmarks, our implementation successfully enables both intra-block parallelization and inter-block pipelining optimizations.

5. RELATED WORK

High level synthesis has seen significant advances in recent years, improving the quality of input source languages. There are many currently active HLS tools in both industry and academia, such as [8, 25, 4, 7, 19, 14]. Leading HLS tools support various intra-block and inter-block optimizations, including pipelining and parallelization. With such powerful optimizations, HLS offers increased productivity with lower design effort; however, in practice these transformations are difficult to apply — only certain data access

patterns are supported, limiting the applicability of an important HLS feature. Recent studies show that there is still a significant performance gap between manual design and HLS-generated designs [23, 17, 10], and the inability to apply these optimizations is one of the causes of this gap.

Real-world applications contain multiple data-dependent blocks that communicate through complex data access patterns, with source code that was not originally intended for HLS. These applications commonly have original data access patterns that do not support the HLS optimizations; thus, it is critical to transform the source code to enable these optimizations. Prior work in optimization opportunities for data-dependent blocks individually worked on pipelining and communication techniques [26, 22, 9]. Ziegler et al. developed coarse-grained pipelining for data-dependent loops that find efficient communication schemes [26], but they assume that the data access order is already identical between the communicating loops. Cong et al. developed a resource constrained scheduling formulation for the communication problem that can find the optimal communication order [9], but the technique requires completely unrolled loops, and additional storage and computation overhead for communication reordering. Similarly, Rodrigues et al. presented a fine-grained synchronization technique with hardware inter-stage buffers to manage the communication [22]. Prior works retain the original execution order but apply techniques to reorder communication; however, these techniques may require large inter-stage buffers in order to handle the reordering, which limits the feasible problem sizes they can support. Furthermore, these works manually optimize communication interfaces, rather than automated optimization such as that performed for high level synthesis.

In our technique, we instead rely on loop transformations to convert the execution order so that the communication order is optimized, while simultaneously minimizing the need for inter-stage buffers. The polyhedral model based loop transformations are based on a mathematical model that can represent any composition of loop transformations using affine transformations [24, 20, 13]. In the past, polyhedral models have been used for maximizing parallelism while minimizing communcation for parallel computing [18, 2]. Recently, polyhedral models have been used in high level synthesis for FPGAs to optimize on-chip memory bandwidth [12, 21], or to optimize the SDRAM bandwidth [3]. In contrast, our approach is to optimize multiple data-dependent blocks simultaneously in order to match their data access patterns and thus simultaneously optimize both intra-block parallelism and inter-block pipelining.

6. CONCLUSION

High level synthesis is a critical technology to ease the adoption of hardware accelerator resources. However, although current high level synthesis offers a variety of powerful optimizations, the implementation constraints limit the applicability and thus impact of the optimizations. Input source codes commonly have complex data access patterns that do not inherently support important high level synthesis optimization techniques, but polyhedral models can model the data access patterns and find loop transformations that enable these important parallelization and pipelining optimizations.

We have presented an integrated technique using polyhedral models to model and enable both intra- and inter-block optimizations. This integrated technique substantially improves the opportunity to use HLS optimizations for parallelism, pipelining, and fine-grained communication. Our framework automatically identifies data access patterns and candidate loop transformations, evaluates the performance benefit of each candidate to select the best option, performs the code transformations, and inserts the HLS code directives and communication structures to implement the optimized hardware. Experimental evaluation demonstrates an average of 6.04X speedup over high level synthesis without our transformations to enable intra- and inter-block optimizations.

7. ACKNOWLEDGMENTS

This work was partially supported by National High Technology Research and Development Program of China 2012AA010902 and A*STAR Singapore under the Human Sixth Sense Project.

8. REFERENCES

[1] Pocc. The polyhedral compiler collection. `http://www.cse.ohio-state.edu/~pouchet/software/pocc/`.

[2] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. *Int. J. Parallel Program.*, 29(5):493–544, 2001.

[3] Samuel Bayliss and George A. Constantinides. Optimizing SDRAM bandwidth for custom FPGA loop accelerators. In *FPGA*, pages 195–204, 2012.

[4] Thomas Bollaert. *High-Level Synthesis: From Algorithm to Digital Circuit*, chapter Catapult synthesis: A practical introduction to interactive C syntheis. Springer, 2008.

[5] Uday Bondhugula et al. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *CC*, pages 132–146, 2008.

[6] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*, pages 101–113, 2008.

[7] Andrew Canis et al. Legup: high-level synthesis for FPGA-based processor/accelerator systems. In *FPGA*, pages 33–36, 2011.

[8] Jason Cong et al. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE TCAD*, pages 473–491, 2011.

[9] Jason Cong, Yiping Fan, Guoling Han, Wei Jiang, and Zhiru Zhang. Behavior and communication co-optimization for systems with sequential communication media. In *DAC*, pages 675–678, 2006.

[10] Jason Cong, Muhuan Huang, and Yi Zou. Accelerating fluid registration algorithm on multi-FPGA platforms. In *FPL*, pages 50–57, 2011.

[11] Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. Customizable domain-specific computing. *IEEE Des. Test*, 28(2):6–15.

[12] Jason Cong, Peng Zhang, and Yi Zou. Optimizing memory hierarchy allocation with loop transformations for high-level synthesis. In *DAC*, pages 1233–1238, 2012.

[13] Paul Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *Int. J. Parallel Program.*, 21(5):313–348, 1992.

[14] Swathi T. Gurumani et al. High-level synthesis of multiple dependent CUDA kernels on FPGA. In *ASPDAC*, 2013.

[15] Ilya Issenin, Erik Brockmeyer, Miguel Miranda, and Nikil Dutt. DRDU: A data reuse analysis technique for efficient scratch-pad memory management. *ACM Trans. Des. Autom. Electron. Syst.*, 12(2), 2007.

[16] Peng Li et al. Memory partitioning and scheduling co-optimization in behavioral synthesis. In *ICCAD*, pages 488–495, 2012.

[17] Yun Liang et al. High-level synthesis: Productivity, performance, and software constraints. *J. Electrical and Computer Engineering*, 2012.

[18] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *ICS*, pages 228–237, 1999.

[19] Alex Papakonstantinou et al. Multilevel granularity parallelism synthesis on FPGAs. In *FCCM*, pages 178–185. IEEE, 2011.

[20] Louis-Noël Pouchet et al. Loop transformations: convexity, pruning and optimization. In *POPL*, pages 549–562, 2011.

[21] Louis-Noël Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *FPGA*, 2013.

[22] Rui Rodrigues, Joao M. P. Cardoso, and Pedro C. Diniz. A data-driven approach for pipelining sequences of data-dependent loops. In *FCCM*, pages 219–228, 2007.

[23] Kyle Rupnow et al. High level synthesis of stereo matching: Productivity, performance, and software constraints. In *FPT*, pages 1–8, 2011.

[24] Michael. E. Wolf and Monica. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, 1991.

[25] Zhiru Zhang et al. *High-Level Synthesis: From Algorithm to Digital Circuit*, chapter AutoPilot: a platform-based ESL synthesis system. Springer, 2008.

[26] Heidi E. Ziegler, Mary W. Hall, and Pedro C. Diniz. Compiler-generated communication for pipelined FPGA applications. In *DAC*, pages 610–615, 2003.