Polyhedral-Based Data Reuse Optimization for Configurable Computing

Louis-Noël Pouchet,[1] Peng Zhang,[1] P. Sadayappan,[2] Jason Cong[1]
[1] University of California, Los Angeles {pouchet,pengzh,cong}@cs.ucla.edu
[2] Ohio State University saday@cse.ohio-state.edu

Abstract

Many applications, such as medical imaging, generate intensive data traffic between the FPGA and off-chip memory. Significant improvements in the execution time can be achieved with effective utilization of on-chip (scratchpad) memories, associated with careful software-based data reuse and communication scheduling techniques.

We present a fully automated C-to-FPGA framework to address this problem. Our framework effectively implements data reuse through aggressive loop transformation-based program restructuring. In addition, our proposed framework automatically implements critical optimizations for performance such as task-level parallelization, loop pipelining, and data prefetching. We leverage the power and expressiveness of the polyhedral compilation model to develop a multi-objective optimization system for off-chip communications management. Our technique can satisfy hardware resource constraints (scratchpad size) while still aggressively exploiting data reuse. Our approach can also be used to reduce the on-chip buffer size subject to bandwidth constraint. We also implement a fast design space exploration technique for effective optimization of program performance using the Xilinx high-level synthesis tool.

Categories and Subject Descriptors B.5.2 [*Hardware*]: Design Aids — optimization; D 3.4 [*Programming languages*]: Processor — Compilers; Optimization

Keywords Compilation; Program Transformations; High-Level Synthesis; Data Reuse

1. Introduction

High level synthesis (HLS) tools for synthesizing designs specified in a behavioral programming language like C/C++ can dramatically reduce the design time especially for embedded systems. While the state-of-art HLS tools have made it possible to achieve QoR close to hand coded RTL designs from designs specified completely in C/C++ [5], considerable manual design optimization is still often required from the designer [17]. To get a HLS friendly C/C++ specification, the user often needs to perform a number of explicit source-code transformations addressing several key issues such as on-chip buffer management, choice of degree of parallelism/pipelining, attention to prefetching, avoidance of memory port conflicts etc., before designs rivaling hand coded RTL can be synthesized by the HLS tool.

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

Our objective is to develop automated compiler support based on the latest advances in polyhedral frameworks (e.g., [12, 38]) to greatly reduce the human design effort currently required to create effectively synthesizable specification of designs using HLS tools. In particular, we develop compiler support for source-to-source transformations to optimize critical resources such as memory bandwidth to off-chip memory and on-chip buffer capacity. We present in this article algorithms and tools to automatically perform the needed loop/data transformations as well as effective design space exploration techniques. Specifically, we make the following contributions.

1. A full implementation of a complete and automated technique to optimize data reuse for a class of programs, for FPGAs. This includes complete technique for dedicated on-chip buffer management, that exploits the data reuse in the transformed program.
2. A compile-time technique to automatically find communication schedules and on-chip buffer allocations to minimize the communication volume under maximal buffer size constraint.
3. A framework for fast design space exploration of transformation candidates based on an available high-level synthesis tool and leveraging the specifics of the optimization framework we use.

The rest of the paper is organized as follows. Section 2 covers the related work. Section 3 presents our automated data reuse framework, and Section 4 our buffer size/communication volume optimization algorithm. Finally, Section 5 presents our fast design space exploration framework and experimental results.

2. Related Work

Design automation and optimization for data reuse have been studied for decades. The data transfer and storage exploration (DTSE) methodology [14, 15] is one of the milestones in this field. A data reuse graph is introduced to express the possible data reuse candidates between array references in the source program [28], where a polyhedral model is used to analyze the data dependence. Then, heuristics based on reuse buffer size and bandwidth reduction can be applied to decide the allocation of the reuse candidates and their memory hierarchy [13, 29]. A large body of previous work has also considered data locality optimization, but focuses exclusively on CPU and data caches optimization [8, 30, 39]. Loop transformations for data locality optimization are only an enabler for effective on-chip data reuse. Numerous previous work for FPGAs and GPUs, such as [9, 14, 15, 19, 24, 31], have considered automatic techniques to promote memory references on-chip. However, previous work had systemic limitations based either on the program representation used (which for instance only approximates the data accessed by a reference), and/or constrained to managing an on-chip buffer that corresponds precisely to a tile. Other work studied the power of the polyhedral transformation framework for FPGA design. For instance, DEFACTO combines a parallelizing compiler

technology (SUIF) with early hardware synthesis tools to propose a C-to-FPGA design flow [21, 36], and MMAlpha [25] focused on systolic designs. These works illustrated the benefit of using advanced compiler technologies for memory optimization and parallelization analysis. However, to the best of our knowledge, none of those frameworks consider a space of program transformation as large as ours, and/or have limited loop tiling capabilities. Bayliss et al. [11] used the polyhedral model to compute an address generator exploiting data reuse, however they do not consider any loop transformation and therefore do not restructure the program to better exploit its inherent data locality potential. In contrast, loop transformations for improving data locality is the starting point of our framework.

Tiling in particular is a crucial loop transformation for data locality improvement, and is one key transformation used in our framework. As an illustration, on a matrix-multiply example previous work by Cong. et al using only loop permutation, (limited) loop skewing, and loop fusion/distribution reduces the communication volume from $3.N^3$ to N^3, using a buffer of size $2.N$ [19]. Using a simple square tiling with tile size T reduces the communication volume to to roughly N^3/T^2, for a buffer of size T^2, and even better solutions can be achieved with rectangular tiles, as used in this paper. However in [19], finer-grain data reuse opportunities are explored in a combined problem searching for buffer allocation and loop transformation simultaneously, which can lead to even smaller buffer sizes than achievable by our present work. Loop tiling often requires a complex sequence of complementary loop transformations such as skewing, fusion/distribution, shifting, etc., to be applied [6,12,27,40], and finding such sequence is a challenging problem. In this work, we address it in an automatic fashion by using the Tiling Hyperplane scheduling method, which is geared towards maximizing data locality and program tilability [12].

Recently, the importance of considering platform-dependent cost modeling in optimizing the loop transformation has been emphasized [32]. Loop transformation and data reuse optimization are loosely coupled by introducing fast hierarchical memory size estimators [26, 33] to evaluate the promising transformations. But the search process lacks an analytic model for guidance, which makes it inefficient to search a large transformation space. Previous work tries to establish analytic optimization formulations for the combined problem, such as optimizations of loop tiling parameters and reuse buffer selections are formulated into quadratic programming [9] and geometric programming [31] respectively. Alias et al. uses tiling and prefetching to reduce the memory traffic [7], focusing on the Altera tool-chain. They proposed a formulation for the prefetching problem and the pipelining of communications, but their approach does not consider the balance between communication volume and scratchpad size/energy, nor any design-space exploration, contrary to the present work.

3. Automatic Data Reuse Framework

3.1 Overview of the Method

In our framework, we perform a multi-stage process to automatically optimize C programs for effective execution on a FPGA. Our approach uses design-space exploration to determine the best performing program variant. Specifically, we search for best performance through the evaluation of different *rectangular* tile sizes. Our framework is built so that different tile sizes lead to different program candidates, with distinct features in terms of the communication schedule, buffer size, loop to be tiled (e.g. when a tile size of 1 is used for this loop), etc. Each candidate is built as follows.

1. We first transform the input program, using polyhedral loop transformations. The objective is to restructure the program so that data locality is maximized (e.g., the "time" between two accesses to the same memory cell is minimized), and at the same time the number of loops that can be tiled is maximized. This is presented in Section 3.2. Tilable loops are tiled using a tile size given as input.
2. We then promote all memory accesses to on-chip buffers in the transformed program, and automatically generate off-chip/on-chip communication code. Data reuse between consecutive iterations of a loop is automatically exploited. This is presented in Section 3.3. The hardware constraints on the maximal buffer size are automatically satisfied, using a lightweight search algorithm that trades off communication volume for buffer size. This is presented in Section 4.
3. We conclude the code transformation process by performing a set of HLS-specific optimizations, such as coarse-grain and fine-grain/task-level parallelism extraction. This is presented in Section 3.4.

3.2 Polyhedral-Based Transformations for Data Reuse

Unlike the internal representation that uses abstract syntax trees (AST) found in conventional compilers, polyhedral compiler frameworks use an internal representation of imperfectly nested affine loop computations and their data dependence information as a collection of parametric polyhedra, this enables a powerful and expressive mathematical framework to be applied in performing various data flow analysis and code transformations.

Significant benefits over conventional AST representations of computations include the effective handling of symbolic loop bounds and array index function, the uniform treatment of perfectly nested versus imperfectly nested loops, the ability to view the selection of an arbitrarily complex sequence of loop transformations as a single optimization problem, the automatic generation of tiled code for non-rectangular imperfectly nested loops, etc.

3.2.1 Polyhedral Program Representation

The *polyhedral model* is a flexible and expressive representation for loop nests with statically predictable control flow. Loop nests amenable to this algebraic representation are called *static control parts* (SCoP) [22,23], roughly defined as a set of consecutive statements such that loop bounds and conditionals involved are affine functions of the enclosing loop iterators and variables that are constant during the SCoP execution (whose values are unknown at compile-time). Numerous scientific kernels exhibit those properties; they can be found typically in image processing filters (such as medical imaging algorithms) and dense linear algebra operations.

```
1  for (t = 0; t < T; ++t) {
2    for (i = 1; i < N-1; ++i)
3      for (j = 1; j < N-1; ++j)
4  R:    B[i][j] = 0.2*(A[i][j-1] + A[i][j] + A[i][j+1]
5                       + A[i-1][j] +A[i+1][j]);
6    for (i = 0; i < N; ++i)
7      for (j = 0; j < N; ++j)
8  S:    A[i][j] = B[i][j];
}
```

Figure 1. Jacobi2D example

First, a program is analyzed to extract its polyhedral representation, including iteration domain, access pattern and dependence information.

Iteration Domains For all textual statements in the program, for example R in Figure 1, the set of its dynamic instances is captured with a set of affine inequalities. When the statement is enclosed by loop(s), all iterations of the loop(s) are captured in the iteration domain of the statement. Considering the jacobi2D kernel

in Figure 1, the iteration domain of R is:

$$\mathcal{D}_R = \{(t,i,j) \in \mathbb{Z}^3 \mid 0 \leq t < \mathrm{T} \wedge 1 \leq i < \mathrm{N}-1 \wedge 1 \leq j < \mathrm{N}-1\}.$$

The iteration domain $\mathcal{D}_R$ contains only integer vectors (or, integer points if only one loop encloses the statement R). The *iteration vector* $\vec{x}_R$ is the vector of the surrounding loop iterators; for R it is (t,i,j) and takes values in $\mathcal{D}_R$. Each vector in $\mathcal{D}_R$ corresponds to a specific set of values taken by the surrounding loop iterators (starting from the outermost to the innermost enclosing loop iterator) when R is executed.

Access functions They represent the location of the data accessed by the statement. In SCoPs, memory accesses are performed through array references (a variable being a particular case of an array). We restrict ourselves to subscripts of the form of affine expressions which may depend on surrounding loop counters and global parameters. For instance, the subscript function for the read reference `A[i-1][j]` of statement R is simply $f_A(t,i,j) = (i-1,j)$.

The sets of statement instances between which there is a producer-consumer relationship are modeled as equalities and inequalities in a *dependence polyhedron*. This is defined at the granularity of the array cell. If two instances $\vec{x}_R$ and $\vec{x}_S$ refer to the same array cell and one of these references is a write, then they are said to be in dependence. Therefore to respect the program semantics, the transformed program must execute $\vec{x}_R$ before $\vec{x}_S$. Given two statements R and S, a dependence polyhedron, written $\mathcal{D}_{R,S}$, contains all pairs of dependent instances $\langle \vec{x}_R, \vec{x}_S \rangle$.

Multiple dependence polyhedra may be required to capture all dependent instances, at least one for each pair of array references accessing the same array cell (scalars being a particular case of array). It is possible to have several dependence polyhedra per pair of textual statements, as some may contain multiple array references.

3.2.2 Program Transformation for Locality/Parallelism

The next step in polyhedral program optimization is to compute a transformation for the program. Such a transformation captures, in a single step, what may typically correspond to a sequence of several tens of textbook loop transformations [23]. It takes the form of a carefully crafted affine multidimensional schedule, together with (optional) iteration domain or array subscript transformations.

In order to expose coarse-grain parallelization as well as data locality optimizations, we first compute a polyhedral transformation which is geared towards maximizing data locality while exposing coarse-grain parallelism when possible. This optimization is implemented via a possibly complex composition of multidimensional tiling, fusion, skewing, interchange, shifting, and peeling. It is known as the Tiling Hyperplanes method [12]. The Tiling Hyperplane method has proved to be very effective in integrating loop tiling into polyhedral transformation sequences [12, 27]. Bondhugula et al. proposed an integrated optimization scheme that seeks to maximally fuse a group of statements, while making the outer loops permutable (i.e., tilable) [12] when possible. A schedule is computed such that parallel loops are brought to the outer levels, if possible. This technique is applied on each SCoP of the program. When coarse-grain parallelism is exposed (such as pipelined tile parallelism), we automatically exploit it to support concurrent execution on the FPGA.

From a data reuse standpoint, the Tiling Hyperplane method schedules iterations that access the same data elements as close to each other as possible, maximizing temporal data locality under the framework constraints. We note that the Tiling Hyperplane method attempts to maximize the number of loops that *can* be tiled, but operates seamlessly on non-tilable (or non-fully tilable) programs, also maximizing locality in those cases. In our framework, loop tiling is applied on the set of loop nests that are made permutable after applying the Tiling Hyperplane method. Finally, Syntactically correct transformed code is generated back from the polyhedral representation, and this code scans the iteration spaces according to the schedule we have computed with the Tiling Hyperplane method. We use the CLooG, a state-of-the-art code generator [10] to perform this task.

3.3 Automatic On-Chip Buffer Management

Most ICs, especially embedded systems, use on-chip buffer memories for fast and energy-efficient access to the most frequently used data. For FPGAs, the total data for the application is much larger than on-chip memory capacity. In contrast to general-purpose processors that use hardware-managed caches to hold frequently accessed data, the use of on-chip buffers with explicit copy-in and copy-out of data is a key optimization for embedded systems [18]. By storing frequently accessed data in the on-chip buffer, the bandwidth contention is decreased, and the overall performance increases significantly as the latency of accessing on-chip data is significantly faster than off-chip accesses. In the following we present a fully automated approach for on-chip buffer management that consists of promoting to local memory (e.g., the on-chip buffer) all memory references in the program.

Promoting the entire data accessed by a program to local memory is often infeasible, in particular for FPGA design where the on-chip buffer resource is limited. Therefore, we want to enable the promotion all program references to an on-chip buffer, while still controlling its size. We chose to solve this problem by using the granularity of the loop iteration, for any of the loops in the program. That is, given an arbitrary loop in the program (which may very well be surrounded by other loops), our technique will compute the minimal on-chip buffer size requirement and associated communications to execute one iteration of this loop, while exploiting the reuse between consecutive iterations of said loop. This implicitly offers a lot of freedom for the on-chip buffer size. By considering the innermost loop, its size will be similar to the number of registers required to execute the computation. By considering the outermost loops it will be equivalent to the entire data space of the program. Any loop in-between will trade off communication count for on-chip buffer size (and its associated static energy).

For example, in Figure 1 if we put on-chip the data accessed by one full execution of the j loop in line 3 (that is exactly one iteration of the i loop), we need to store the i^{th} row of A and B, as well as the $(i-1)^{th}$ and $(i+1)^{th}$ rows of A, leading to a buffer requirement of $4.N$. This buffer must be filled for each iteration of the i loop, that is roughly $T.N$ times (total communication volume is roughly $4.N^2.T$). Putting on-chip the full computation (that is, along the t loop on line 1) leads to a $2.N^2$ buffer requirement, but to be filled only once (total communication volume is reduced to $2.N^2$). So, the trade-off here is between a buffer size N times smaller versus a communication volume increase of $2.T$. Below, we show how to build better solutions exploiting reuse across executions of a loop.

3.3.1 Parametric Data Spaces for On-Chip Buffer Support

We now present our formalization to effectively promote memory references for on-chip buffer usage. Our technique operates on each array individually, and promotes optimally (under the framework constraints) all references to this array into a dedicated on-chip buffer for this array. Our approach is based on the concept of *parametric polyhedral sets* to express the set of data elements being used at various specific points of the computation. Those sets correspond exactly to the data to be communicated, reused, or stored. We then use a polyhedral code generator to scan those sets, and properly modify the program by inserting the code that scans communications sets, and change main memory references in the modified source code to on-chip buffer references.

We first define the data space of an array A for a program. The data space is simply the set of all data elements accessed through the various access functions referencing this array, for each value of the surrounding loop iterators where the reference is done. We use the concept of the image of a polyhedron (e.g., the iteration domain) by an affine function (e.g., the access function). The image of a polyhedron $\mathcal{D}$ by an affine function F is defined as the set $\{\vec{y} \mid \forall \vec{x} \in \mathcal{D},\ F(\vec{x}) = \vec{y}\}$.

DEFINITION 1 (Data space). *Given an array A, a collection of statements $\mathcal{S}$, and the associated set of memory references F_S^A with $S \in \mathcal{S}$, the data space of A is the set of unique data elements accessed during the execution of the statements. It is the union of the image of the iteration domain by the various access functions:*

$$DS(A) = \bigcup_{S \in \mathcal{S}} Image(F_S^A, \mathcal{D}_S)$$

We remark that $DS(A)$ is not necessarily a convex set, but can still be manipulated with existing polyhedral libraries. For example, in Figure 1, the data space of $DS_R(B)$ for the first statement (R : line 4) is the 2-dimensional square set going from 1 to $N-2$ in each dimension. But for the second statement (S : line 7), $DS_S(B)$ is the 2D square set going from 0 to N. As we make the union, it means $DS(B)$ is the 2D square set going from 0 to N in each dimension.

In order to capture the data accessed at a particular loop level, we must fix the value of the surrounding loop iterators to a certain value in the data space expression, while preserving all inner loop iterations. For the data space computation to be valid for any execution of this loop (nest), we resort to using parametric constants (i.e., whose value is fixed but unknown) in the formulation. All sets and expressions computed will be parametric forms of those constants, and therefore valid for any value these constants can take; they will consequently be valid for any value the surrounding loop iterators can take during the computation.

We first define the parametric domain slice, that will be the enabler for defining the data space of a loop iteration.

DEFINITION 2 (Parametric domain slice). *Given a loop nest with a loop l of depth n surrounded by $k-1$ loops, and an integer constant α, the parametric domain slice (PDS) of loop l is a subset of $\mathbb{Z}^n$ defined as follows:*

$$P_{l,\alpha} = \{(x_1,\ldots,x_n) \in \mathbb{Z}^n | x_1 = p_1, \ldots, x_{k-1} = p_{k-1}, x_k = p_k + \alpha\}$$

where $p_1,\ldots,p_n$ are parametric constants unrestricted on $\mathbb{Z}$.

For example, for loop i in line 3 of Figure 1, we have:

$$P_{i,1} = \{(t,i,j) \in \mathbb{Z}^3 | t = p_1, i = p_2 + 1\}$$

This is a (parametric) set of 3D integer points with the first two components of each point always having the same (parametric) values. This set contains an infinite number of points, as the third component takes any value in $\mathbb{Z}$.

We can now adapt the definition of a data space to the subset of data which is accessed by a loop iteration.

DEFINITION 3 (Data space of a loop iteration). *Given an array A, a collection of statements $\mathcal{S}$ surrounded by a loop l and their associated set of memory references F_S^A with $S \in \mathcal{S}'$, and $\vec{P}_{l,0}$ a PDS for loop l, the data space of A is the set of unique data elements accessed during one iteration of l:*

$$DS(A, \vec{P}_{l,0}) = \bigcup_{S \in \mathcal{S}} Image\left(F_S^A, \left(\mathcal{D}_S \cap P_{l,0}\right)\right)$$

To illustrate the power and generality of this approach, in Figure 2 we show the sets $DS(A, \vec{P}_{j,0})$ (left) and $DS(A, \vec{P}_{j,-1})$ (center), the data space of the immediately preceding iteration, for the first j loop (line 3) in the Jacobi2D example. By computing the difference or intersection between those sets (right), we can capture naturally the data reused between two consecutive iterations, as well as the data that is not alive at the previous iteration.

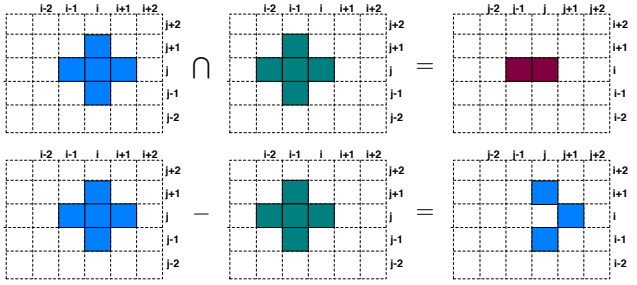

Figure 2. Computation of the *Reuse* (top) and *PerIterC* (bottom) sets for the `j` loop of jacobi2D

Formally, the reused data space between consecutive iterations of a loop is defined as follows. All data which is reused does not need to be communicated at the next iteration.

DEFINITION 4 (Reuse space). *Given an array A, and $\vec{P}_{l,0}$ and $\vec{P}_{l,-1}$ two PDS for loop l, the reused data space between two consecutive iterations of l is:*

$$Reuse(A, \vec{P}_{l,0}) = DS(A, \vec{P}_{l,-1}) \cap DS(A, \vec{P}_{l,0})$$

The communication required for each loop iteration is defined as follows. It consists in only the data elements that were not accessed by the previous iterations.

DEFINITION 5 (Per-iteration communication). *Given an array A, and $\vec{P}_{l,0}$ a PDS for loop l, then assuming the data reused between two consecutive iterations is still in local memory, the data space required to communicate in order to compute a given iteration of l is:*

$$PerIterC(A, \vec{P}_{l,0}) = DS(A, \vec{P}_{l,0}) - Reuse(A, \vec{P}_{l,0})$$

Finally, to ensure that for the first loop iteration, all data is ready in the on-chip buffer, we must communicate the per-iteration data set but also initialize the on-chip buffer with the reuse set at the first loop iteration, as no previous iteration has already loaded the data.

DEFINITION 6 (Initialization). *Given an array A, the data to be stored in the buffer before the loop starts is:*

$$Init(A, l) = Reuse(A, \vec{P}_{l,c})$$

with $c = lb(l) - p_k$, where $lb(l)$ is the lower bound expression of the loop l and p_k is the parameter associated with the loop l.

It is also required to store back to main memory any data element that is produced. This is captured by the copy-out set as defined below.

DEFINITION 7 (Per-iteration copy-out). *Given an array A and $\vec{P}_{l,0}$ a PDS for loop l, the copy-out set $CopyOut(A, \vec{P}_{l,0})$ storing written data in on-chip buffer back to main memory is the data space $DS^w(A, \vec{P}_{l,0})$ which considers only the access functions F_S^A that correspond to written references.*

We remark that as our framework computes the data reuse only between two consecutive iterations, it does not necessarily capture all the reuse potential in a loop nest. In particular, reuse between two non-consecutive iterations (e.g., `A[3i]` and `A[3i+6]`) is not exploited. We believe that is is not a strong limitation in practice, as those cases only rarely occur, especially after having applied the Tiling Hyperplane method to transform the original program.

3.3.2 Code Generation

Local buffer computation The polyhedral set defining the buffer requirement for a given loop l is $DS(A,\vec{P}_{l,0})$. We take a simple approach, which is based on computing the rectangular hull of *DS* (that is, the smallest rectangle that contains the set *DS*). It offers the advantage of generating a simple code with equally simple access functions. On the other hand, space can be wasted in particular when *DS* is not convex and contains holes, and/or when *DS* is a skewed parallelogram. In our experiments, buffer sizes of up to $2\times$ the minimal required size have been allocated, due to this over-approximation. Optimal allocation techniques do exist [20], and we plan to investigate their implementation as future work.

We compute the rectangular hull of $DS(A,\vec{P}_{l,0})$ dimension by dimension, using the following formula, for each dimension i:

$$\begin{aligned} dim_i &= project(rectangularHull(DS(A,\vec{P}_{l,0})),i) \\ bs_i &= lexmax(dim_i,1) - lexmin(dim_i,1) \end{aligned}$$

where $project(DS,i)$ projects the set *DS* onto the dimension i, and $lexmax(dim_i,1)$ returns the coordinate of the extremal (maximal) point of the one-dimensional set dim_i. We remark that this stage, this coordinate and the entire sets are parametric forms of the loop bounds.

Code generation algorithm Because of our generalized formalism above, the communication scheduling and final code generation has become straightforward. Our algorithm for the local memory promotion of an array at a given loop is shown in Figure 3. Function createLocalBufferDecl creates a declaration to a local buffer of same data type as the argument array, and its size is computed as the rectangular hull of the argument set. Function createScanningCodeIn creates an imperative C program that scans all elements of the polyhedral set given as an argument. For each point in this set, a statement `A_l[i%bsi].. = A[i]..` (or the opposite assignment for createScanningCodeOut) is executed, where bs_i is the buffer size along dimension i. All parameters p_i introduced by the PDS used are replaced with the loop iterator variable symbol they were assigned to. convertGlobalToLocalRef replaces all references to the original array with references to the local array, using modulo indexing similar to the copy statement above.

```
LocalMemoryPromotion:
Input:
  A: array to promote
  l: loop along which A is promoted
Output:
  l: in-place modification of l

1  A_l ← createLocalBufferDecl(A, DS(A,l))
2  pre ← createScanningCodeIn(A_l, Init(A,l))
3  pic ← createScanningCodeIn(A_l, PerIterC(A,l))
4  wout ← createScanningCodeOut(A_l, WriteOut(A,l))
5  convertGlobalToLocalRef(l, A, A_l)
6  insertFirstInLoopBody(l, pic)
7  insertLastInLoopBody(l, wout)
8  prependBlock(l, pre)
9  prependBlock(getEnclosingFunc(l), A_l)
```

Figure 3. Code generation algorithm for a loop l

Correctness of the algorithm Our algorithm is robust to any value taken by the loop iterator for which the buffer communication is computed, per virtue of the PDS mechanism. As a consequence, simply translating the various sets described above for each value of the surrounding and current loop iterators is sufficient to capture exactly the data accessed by each iteration. We also note that this algorithm is a data layout transformation only: the scheduling of the operations remains unchanged after application of the local memory promotion pass. Finally, to ensure correctness we write back all data elements written during a loop iteration at the end of the iteration. While there are occurrences where a lower amount of write communications may be performed, such as with reduction loops, this ensures that an element being written and then accessed at a much later iteration will hold the correct value.

3.4 HLS-Specific Optimizations

Despite very significant advances in HLS, a variety of complementary source-level transformations are often needed to produce the best result. In particular, fine-grain parallelism exposure and explicit overlapping of computation and communications dramatically impacts the performance. We leverage the power of the polyhedral model to precisely capture all sources of parallelism, reorganize the computation to pack sequential tasks together and recognize sets of parallel tasks, and produce a fine-grain task dependence graph that is used for task scheduling by the HLS tool.

3.4.1 Communication Prefetching and Overlapping

It is critical to properly overlap communication and computation, in order to minimize the penalty of data movements. A common technique is to prefetch in advance the data that will be required later by the computation. Following our paradigm of managing communications at the loop iteration level, the set of data elements that should be prefetched at the current iteration is defined as follows.

DEFINITION 8 (Communication prefetch). *Given an array A, and $\vec{P}_{l,0}$ and $\vec{P}_{l,1}$ two parametric iteration vectors for loop l, the data that needs to be communicated to execute the next iteration is:*

$$Prefetch(A,\vec{P}_{l,0}) = PerIterC(A,\vec{P}_{l,1})$$

When operating on tiled programs, we remark that if the loop along which we bufferize is a tile loop, this set corresponds to the data set being used by the next tile, minus elements which are reused between consecutive tiles. Therefore inter-tile reuse between consecutive tiles is automatically captured in our framework.

This set is scanned at the beginning of the current loop iteration, and the associated code segment is put in a dedicated function (i.e., a task) that can be executed in parallel with the rest of the loop iteration that has also been put in a dedicated function.[1] In terms of storage, one can implement a simple double-buffer for the *PerIterC* and *Prefetch* sets, with a buffer swap at the beginning of each loop iteration.

3.4.2 Loop Pipelining and Task Parallelism

One of the most important hardware optimizations for high-performance RTL design leverages the fine-grain parallelism available in the program. *Loop pipelining* pipelines consecutive iterations of the loop, and this amounts to executing loop iterations in parallel if the loop is synchronization-free. In this work we automatically apply post-transformations to expose parallelism at the innermost loop level, if possible. Previous work on SIMD vectorization for affine programs has proposed effective solutions to expose inner-loop-level parallelism [12, 37], and we seamlessly reuse those techniques to enable effective loop pipelining on the FPGA. This is achieved by using additional constraints during and after the tiling hyperplanes computation, to preserve one level of inner parallelism. As a result, we mark all innermost loops with a specific `#pragma AP pipeline II=1`, and let the HLS tool find the best II it can for this loop. And for all innermost parallel loops, we also insert `#pragma AP dependence inter false` pragmas

[1] AutoESL is able to exploit task-level parallelism in this case.

for all variables which are written in the loop iteration, to prevent conservative dependence analysis.

There exists a significant amount of parallelism *inside* loop iterations, in particular when prefetching is implemented. That is, some operations inside a loop iteration can be executed concurrently, and synchronization is (possibly) required at the end of the loop iteration. Such parallelism can be captured effectively in the polyhedral representation by focusing the analysis on the modeling of the possible order of statements within an iteration of a given loop. We implemented this analysis as a post-pass, after polyhedral code generation for the parallelism/locality extraction. Polyhedral transformations may indeed expose more task-level parallelism inside a loop iteration after code generation (precisely, after the separation step [10]) has been been performed. Focusing on a single loop at a time by using the appropriate PDS, we are able to automatically compute a graph of dependent operations within a given iteration of the loop body. This dependence graph is used to restructure the code as a collection of separate functions with distinct arguments for functions/tasks that can be run in parallel.

3.4.3 Additional Optimizations

Finally, we have implemented a series of complementary optimizations that have a significant impact on the performance of the final design. Specifically, we implemented a simple common subexpression elimination, loop bound normalization [8], simplification and hoisting, and a simplification of the circular buffer access functions. Indeed, affine loops generated increment by stride of 1, and modulo expressions can easily be replaced by a simple loop increment and a test against the modulo value at the end of each loop iteration.

4. Resource-Constrained On-Chip Buffer Management

The algorithm presented above to compute communications and the associated on-chip buffer size is designed to work seamlessly for any loop in the program. We now show how to leverage the generality of this approach to transform any affine program to use on-chip memory for all computations, while meeting any user-defined resource constraint on the maximal on-chip buffer size. We proceed in two steps: first we present how to build the solution space, and then we show how to find an optimal solution satisfying both bandwidth minimization and buffer size requirement.

4.1 Search Space Construction

The first step is to build the set of possible solutions for each array and each loop in the program. This is shown in Figure 4. In order to build the solution set, we simply apply the process described in the previous section for each arrays and each loop individually. That is, for the cross-product of all arrays and loops, we produce a tuple containing the buffer size requirement (*BS*) if bufferizing this specific array along this specific loop, and the bandwidth requirement (*BW*). *BS* is computed using the formula shown in Section 3.3.2, and *BW* is computed as the product of the cardinality of the *PerIterC* set by the number of executions of the loop, which can be exactly computed at compile-time for SCoPs, as later shown in Section 5.

By design of the algorithm, the number of possibilities is very tractable. It is the product of the number of loops in the program by the number of arrays; in practice most of the time for SCoPs this number is below 100. Our implementation is very fast, and computing all solutions is achieved in a matter of seconds for complex tiled 3D stencil programs.

We note that in order to guarantee correctness for the case of imperfectly nested loops, we enforce that, for a given array, all loops at a same nesting level are bufferized. This may lead to conservative solutions, but greatly simplifies the code generation process.

```
BuildSolutionSet:
Input:
  P: input SCoP
Output:
  solset: solution set for loop selection

  solset ← createEmptySetofTuples
  forall arrays a in P do
    forall loops l in P do
      ldup ← cloneAST(l)
      LocalMemoryPromotion(a,ldup)
      BS ← getBufferVolume(ldup)
      BW ← getCommVolume(ldup)
      insertTuple(solset,{a,l, (BW,BS)})
    end do
  end do
  return solset
```

Figure 4. Create solution space

4.2 Optimization Problem

We seek a solution that will systematically satisfy the objective that the sum of the buffer sizes required does not exceed the available on-chip resources (e.g., the number of BRAMs). That is, we want to find the loops l for all arrays a such that:

$$\sum_{a \in Arrays(P)} BS(a,l) < maxBuffSize$$

We note that as soon as the largest data space of a single iteration of the inner loops (that is usually in the order of the number of registers required to execute the iteration, assuming no spilling) is below *maxBuffSize*, our algorithm will find a solution. In practice, the number of BRAMs available on-chip is significantly larger than the number of registers required to execute a single iteration; hence our method will always find a valid solution.

Satisfying the above constraint let us find a solution that will meet hardware resource limitations. In order to find an (optimal) solution, we need to add the optimization objective. In this work we choose to minimize the bandwidth requirement (i.e., the communication volume). We note that this solution relates only to the communication scheduling, for a given fully specified program. We aim here at computing a valid generated code *for a given tiled program whose tile sizes have been already set*. The problem of finding a final solution that will maximize bandwidth usage is addressed in the next section, and is framed as a tile size selection problem, solved using fast design-space exploration techniques. The final constrained bandwidth minimization problem is stated as:

$$\begin{aligned} minimize \quad & \sum_{a \in Arrays(P)} BW(a,l) \\ s.t. \quad & \sum_{a \in Arrays(P)} BS(a,l) < maxBuffSize \end{aligned}$$

Solving this problem is achieved by repeatedly scanning the solution set obtained with `BuildSolutionSet`. For each array and each loop we compute the total bandwidth and buffer size requirement to find the solution with minimal bandwidth that meets the buffer size constraint. The complexity of finding the optimal solution is n^d, where n is the number of loops in the program and d the number of arrays. Despite being an exponential solver, in practice it is extremely fast. For instance a 3D image processing algorithm we tested has 12 loops and 4 arrays, leading to computing about 20,000 sums of 4 elements, which is done in a negligible time on modern processors. For too large spaces, one can implement approximate

solving heuristics based for instance on dynamic programming. No such case has been encountered in our test suite.

5. Fast DSE for Complete Optimization

5.1 Methods for Fast DSE

Our objective is to enable quality of result (QoR) computation by AutoESL without having to resort to complete synthesis/simulation, in order to speed up the design-space exploration phase. This is particularly important for our method as we employ DSE to search the space of possible tile sizes, for tilable programs.

5.1.1 Capturing the Exact Control Flow

In order to obtain accurate metrics from AutoESL RTL latency estimator, we must provide it with complete information about each loop in the program. The loop trip count (minimal, average and maximal) of a loop must be provided for each and every loop to be mapped to the FPGA. While for general programs it is impossible to accurately compute the loop trip count at compile-time, this is not an issue with SCoPs as addressed in this paper. As the control flow is static and therefore not data dependent, we can compute the number of times each statement in the loop body is executed, thus deducing the trip count of the surrounding loop. This is a critical benefit of manipulating SCoPs: *design-space exploration can be accurately achieved without synthesizing/running or simulating the application*. We use again our concept of PDS to capture the trip count of a statement in the following formula.

DEFINITION 9 (Statement trip count for a loop). *Given a statement S surrounded by a loop l, and $\vec{P}_{l,0}$ a parametric iteration vector, the trip count (noted STC) of l for S is:*

$$\begin{aligned} \mathcal{D}_{S,l} &= \mathcal{D}_S \cap P_{l,0} \\ STC(\mathcal{D}_S, l) &= lexmax(\mathcal{D}_{S,l}, k+1) - lexmin(\mathcal{D}_{S,l}, k+1) \end{aligned}$$

where k is the number of loops surrounding l dimension.

The computation of the minimal and maximal trip count of a loop follows naturally.

DEFINITION 10 (Loop trip count). *Given a loop l and a collection $\mathcal{S}_l$ of statements surrounded by l. The minimal and maximal trip count of l are given by:*

$$TC_{min}(l) = \min_{S \in \mathcal{S}_l}(STC(\mathcal{D}_S, l)) \qquad TC_{max}(l) = STC((\bigcup_{S \in \mathcal{S}_l} \mathcal{D}_S), l)$$

This trip count computation is performed for each loop of the program. We note that as we have described the communication sets in a purely polyhedral fashion, loops introduced to scan the various data sets are necessarily following the static control flow requirements, and can be exactly analyzed at compile-time. When entering the final design space exploration stage, only numerical values can be provided to AutoESL to represent the loop trip count. At this stage, parameters (such as problem size, etc.) are inlined to their numerical value, leading to simple scalar expressions for the loop trip count.

5.1.2 Accurate Memory Latency Estimation

Off-chip SDRAM memory access has a high latency and limited bandwidth. To fully utilize the memory bandwidth, we use two FIFOs as to bufferize (a) the memory requests and (b) the fetched data, and access the off-chip memory in bursts. AutoESL does not natively model the latency consumed by off-chip memory accesses. To model this latency and throughput of the off-chip accesses, we modify the functions scanning the various communication sets. We insert cycle-wasting operations to emulate the time spent doing the corresponding off-chip accesses.

A property of our approach is that communications are executed in bursts, for each array and set to be scanned. So, at the beginning of each chunk of off-chip accesses we insert a `burst_wait()` function call, which corresponds to the burst time overhead (startup latency). We also insert `data_wait()` function calls for each word access. We have implemented these functions to force the design of a p-cycle long operation in the critical path of the function, so that AutoESL will count the additional latency introduced by `burst_wait()` and `data_wait()` in the final execution latency report, so as to emulate the time taken by transferring data to/from off-chip memory. In our experiments, we have micro-benchmarking on the Convey HC-1ex, and obtained $p = 131$ cycles for the burst waiting time, and $p = 1.15$ cycles for the per-word access time. Using this mechanism we accurately capture the bandwidth throughput per communication FIFO in our target platform.

5.1.3 Communication/Computation Functions

Finally, in order to effectively exploit the FIFO communication mechanism, and in order to simplify the DSE method, we perform a final AST-based transformation to the generated program. First, we create a *communication prefetch* function by cloning the entire program generated by the previous algorithms and removing all loops/code segments that do not relate to off-chip/on-chip communication. We then replace all communications by non-blocking FIFO send requests. That is, all requests are sent as fast as possible until the request buffer of the FIFO module is full (the number of in-fly requests is limited by the implementation), which makes the prefetch function wait.

Second, we modify the transformed program (that contains the actual computations) such that all communications are blocking FIFO receive requests. That is, until the data is available on the FIFO, the program will wait. We finish by encapsulating the program in a *computation function*.

We conclude the transformation process by calling the communication prefetch and computation functions simultaneously in the main FPGA function, so that communication prefetch and computation are perfectly overlapped. We note that for the fast DSE approach, the cycle wasting functions are inserted in the prefetch function, thus emulating the time taken to transfer the data from off-chip memory to the FIFOs.

5.2 Experimental Results

5.2.1 Implementation Details

The entire tool-chain presented in this paper has been fully implemented as an open-source software, PolyOpt/HLS.[2] Specifically, we have based our work on the PolyOpt/C polyhedral compiler [34] we have implemented, which is itself based on the LLNL ROSE source-to-source compiler; and on PoCC, the Polyhedral Compiler Collection [4] we have implemented. All polyhedral operations are performed using Sven Verdoolaege's ISL library [38], and we use CLooG [10] for the polyhedral code generation part. Starting from an input sequential C program to be executed on the CPU, our tool-chain automatically extracts regions where the framework can be applied, performs data locality, parallelization and tiling loop transformations, local memory promotion and all the additional HLS-specific optimizations mentioned above. Each SCoP is mapped to the FPGA, using a custom FIFO data management module.

5.2.2 Experimental Setup

Target FPGA platform We design the optimized codes for a multi-FPGA platform Convey HC-1ex [2], which provides four Xilinx Virtex-6 FPGAs (xc6vlx760-1-ff1760) and total bandwidth up to 80GB/s. We use the Xilinx ISE toolchain, version 14.2, which

[2] Available at http://cadlab.cs.ucla.edu/PolyOptHLS.

has been validated by Convey for the HC-1. For HLS, we use AutoESL, version 2011.4 [3]. The RTL is connected to memory interfaces and control interfaces provided by Convey, which have been designed to operate at 150MHz. Hence the working frequency of our core design is set to 150MHz. Off-chip memory runs at 300HMz.

Benchmark Description We evaluate our framework using two core image processing algorithms for 3D MRI, denoise and segmentation. These algorithms are taken from the CPU implementation of the CDSC medical imaging pipeline [1,16,17]. We also evaluate 2 benchmarks from the PolyBench/C test suite, representative of compute-bound and memory-bound numerical kernels. They are described in Table 1. We report the total number of operations as $x \times y$ where x indicates the number of operations per loop iteration, and y the number of iterations. All benchmarks use single-precision floating point arithmetic in the input C code. is well-known that

Table 1. Description of the benchmarks used

Benchmark	Description	#fp ops
denoise	3D Jacobi+Seidel-like 7-point stencils	$61 \times 256^3 \times 15$
segmentation	3D Jacobi-like 7-point stencils	$67 \times 256^3 \times 150$
DGEMM	matrix-multiplication	$3 \times 2048^3 + 2048^2$
GEMVER	sequence of mv	11×2048^2

5.2.3 Details of DSE Results

The time taken by our framework is decomposed as follow. For each point in the design space (e.g., a different tile size), the end-to-end transformation from the original C program to the AutoESL-friendly input C file (this includes program transformations for locality, on-chip buffer management and optimizations, and HLS-specific optimizations) takes about one minute, for the most complex program. AutoESL transforms the input C program and generates RTL as well as complete latency/usage reports in at most two minutes in our experiments. So, testing 100 points takes at most five hours, and took usually around two hours in our experiments.

We have set a maximal buffer size limit to 1440 BRAMs, as it is the maximum for the Virtex-6 FPGA on the Convey HC-1. To capture multiple scenarios of bandwidth usage, we evaluated about 100 different rectangular tile sizes, using different power-of-two values in each of the three tile dimensions. Each tile size will have a different buffer requirement, and a different communication volume.

Figure 5 plots the results of the fast DSE framework that we implemented on three representative benchmarks, for a subset of the entire computation. In this figure, we compare side-by-side the off-chip communication volume on the y axis, in number of 32-bit elements communicated, with the total off-chip communication time on the x axis, as reported by AutoESL. DSE results are reported using a single PE per FPGA.

We observe significant variations in the communication volume that can be transferred in the same amount of time. For instance, the time to transfer the same amount of data can vary by more than $3\times$ for Segmentation. This is because the quality of the RTL generated by AutoESL depends on the source code generated by our framework. Multiple factors influence the performance of the generated code. First, loop bounds used to compute the data space to communicate may be significantly more complex between two tile sizes. This is an artifact of polyhedral code generation, where generated loop bounds may contain tens or more sequences of *min*, *max*, *ceil* and $floor$ functions. For some tile sizes not evenly dividing the data space to communicate, more complex code is generated and the QoR is lowered. Second, the benefit of loop pipelining depends on the loop trip count of the innermost loops. For loops with a too-small trip count (lower than the pipeline depth), the benefit of pipelining will be reduced. As a consequence, two tile sizes having a similar communication volume (e.g., $4 \times 8 \times 1$ and $1 \times 4 \times 8$) will see a different QoR. While analytical modeling of those specific factors may be achievable, it is important to note that AutoESL is a *production compiler*. As such, it is fragile and sensitive to the input program shape, as different source codes triggers different internal optimizations, therefore leading to different QoR. Such effect is a well-known artifact of compilers, and has been widely observed and discussed for production compilers [35].

All those factors confirm our claim that using fast DSE addresses important considerations for the final performance. It is very unlikely that all artifacts related to QoR obtained by the HLS tools can be modeled analytically, as it would be equivalent to building an analytical model of an optimizing compiler. In addition, because of the very fast speed of AutoESL, this modeling effort is non-necessary. So, RTL generation is used to capture those effects such as as how good a compiler (our framework or AutoESL) will be at optimizing different program variants.

Figure 6 shows, for the same benchmark and design space, the Pareto-optimal points for total time and buffer size requirement.

We observe a trade-off between the buffer size requirement and the total time, illustrative of tile size exploration results. Indeed, the majority of data reuse is achieved with tile sizes that fit in a few tens of kB; for instance for segmentation a tile size of $4 \times 8 \times 256$ uses only 73 BRAMs, and has achieved a communication volume reduction of $20\times$ with respect to a non-tiled variant. Using a larger tile size requires a significantly larger buffer size (typically holding a complete 2D slice of the image), but achieves only a small communication improvement ($21\times$ vs. $20\times$ above). In addition, as pointed out above, one key challenge in improving the total execution time is taking into account all compiler optimization effects. With our DSE approach, we can select the transformed variant that achieves the best estimated total time, this takes into account all high-level synthesis/RTL generation artifacts.

5.2.4 Complete Results

Table 2 summarizes the best version found by our framework, for each tested benchmark. We report #PEs the number of replications of the full computation we have been able to place on a single Virtex-6 FPGA as in the Convey HC-1, showing the level of coarse-grain parallelization we have achieved. BRAM and LUT are totals for the set of PEs placed on the chip.

Table 2. Characteristics of Best Found Versions

Benchmark	tile size	#PEs	#BRAM	#LUT
denoise	4×8×128	2	132	178544
segmentation	4×8×256	8	584	177288
DGEMM	8×256×32	16	320	112672
GEMVER	128×128	10	500	140710

Table 3 reports the performance, in GigaFlop per second, of numerous different implementations of the same benchmark. out-of-the-box reports the performance of a basic manual off-chip-only implementation of the benchmark, without our framework. PolyOpt/HLS-E reports the performance achieved with our automated framework. Those are AutoESL results obtained with our fast DSE framework. Hand-tuned reports the performance of a manually hand-tuned version serving as our performance reference, from Cong et al. [17]. It has been designed through time-consuming source code level manual refinements, specifically for the HC-1ex machine. It demonstrated that a 4-FPGA manual design for denoise and segmentation systematically outperforms a CPU-based implementation, both in terms of performance improvement (from $2\times$ to $20\times$) and energy-delay product (up to $2000\times$), therefore showing the great potential of implementing such 3D image processing algorithms on FPGAs [17].

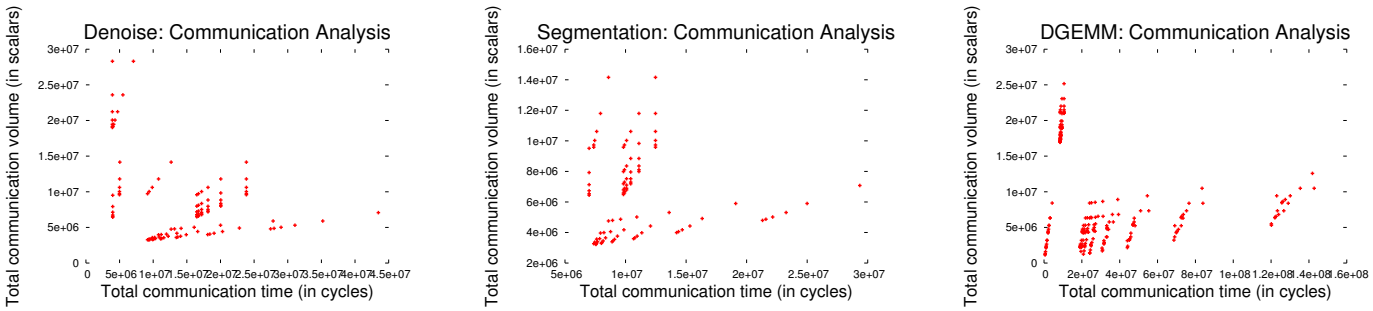

Figure 5. Communication time vs. Communication volume

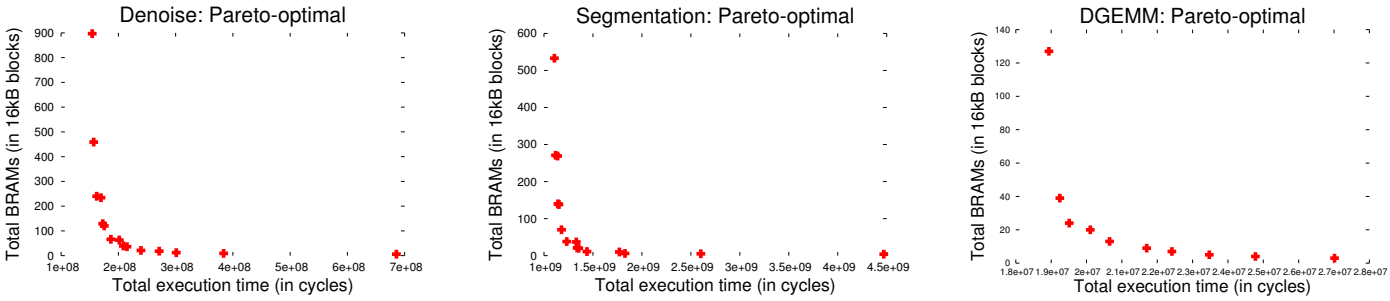

Figure 6. Total time vs. On-Chip Buffer Size Requirement, Pareto-optimal points

We observe that for denoise (only 2 PEs were generated by PolyOpt/HLS) the final performance, despite being significantly better than an off-chip-based solution, remains far from the manual design (which uses 4 PEs). On one hand, the code we generate, and especially the loop structures, are more complex for denoise than, e.g., segmentation. This leads to under-performing execution for our automatically generated code. On the other hand, the reference manual implementation uses numerous techniques not implemented in our automatic framework, such as in-register data reuse, fine-grain communication pipelining, and algorithmic modifications leading to near-optimal performance for this version.

For segmentation, we outperform the manual design, despite the clear remaining room for improvement our framework still has, as shown by the denoise number. We mention that semi-automated manual design can be performed on top of our framework, to address optimizations we do not support, such as array partitioning.

Table 3. Side-by-side comparison

Benchmark	out-of-the-box	PolyOpt/HLS-E	hand-tuned [17]
denoise	0.02 GF/s	4.58 GF/s	52.0 GF/s
segmentation	0.05 GF/s	24.91 GF/s	23.39 GF/s
dgemm	0.04 GF/s	22.72 GF/s	N/A
gemver	0.10 GF/s	1.07 GF/s	N/A

Finally Table 4 compares the latency as reported by AutoESL using our memory latency framework for fast DSE, against the wall-clock time observed on the machine after full synthesis of the generated RTL. We report the performance of a single PE call executing a subset (slice) of the full computation.

Table 4. AutoESL vs. full synthesis comparison (in cycles)

Benchmark	AutoESL only	full synthesis
denoise-1PE (1/32 slice)	23732704	25254164 (+6%)
segmentation-1PE (1/32 slice)	131984559	148878928 (+12%)
dgemm-1PE (1/64 slice)	5022287	5055335 (+1%)

6. Conclusion

High Level Synthesis (HLS) tools for synthesizing designs specified in a behavioral programming language like C/C++ can dramatically reduce the design time especially for embedded systems. HLS systems have now reached a level of advancement to be able to generate RTL that comes quite close to hand generated designs. However, the current state-of-the art is still very far from being able to take a simple high-level description of a system in C/C++ and derive an efficient FPGA implementation. Currently, an expert designer must perform a number of manual source-level transformations of the input C/C++ code to create an "HLS-friendly" C/C++ program before an effective hardware design can be synthesized by the HLS tool.

We have provided in this paper a complete and fully implemented compiler support to alleviate the burden of manually transforming an input sequential C program into a version that can be effectively mapped to FPGA using HLS tools. Our approach leverages the polyhedral compilation framework to automatically transform the input program for data reuse improvement, as well as for outer and inner parallelism extraction. We have designed and implemented a novel and powerful end-to-end solution for on-chip buffer optimization, that automatically implements the available data reuse in a loop nest. This approach is able to meet any hardware-based resource constraint on the maximal buffer size. In addition we presented a complete fast design space exploration technique, leveraging the specifics of polyhedral program. As a result, we have performed extensive design space exploration using the Xilinx ISE tool-chain on medical imaging algorithms. Experiments showed very significant performance improvements over purely out-of-the-box off-chip automatic solutions, and our automated framework even beats in one case a hand-tuned reference implementation of a segmentation algorithm.

Acknowledgment This work was supported by the Center for Domain-Specific Computing (CDSC) funded by NSF "Expeditions in Computing" award 0926127, and the Gigascale Systems Research Center (GSRC).

References

[1] Center for domain-specific computing. http://cdsc.ucla.edu.

[2] Convey. http://www.conveycomputer.com.

[3] http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm.

[4] Pocc 1.1. http://pocc.sourceforge.net.

[5] An independent evaluation of the autoesl autopilot high-level synthesis tool. Technical report, Berkeley Design Technology, Inc., 2010.

[6] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. In *ACM/IEEE Conf. on Supercomputing (SC'00)*, Dallas, TX, USA, Nov. 2000.

[7] C. Alias, A. Darte, and A. Plesco. Optimizing remote accesses for offloaded kernels: application to high-level synthesis for fpga. *SIGPLAN Not.*, 47(8):285–286, Feb. 2012.

[8] J. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.

[9] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *ACM Symposium on Principles and practice of parallel programming*, pages 1–10. ACM, 2008.

[10] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'04)*, pages 7–16, Sept. 2004.

[11] S. Bayliss and G. A. Constantinides. Optimizing sdram bandwidth for custom fpga loop accelerators. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, FPGA '12, pages 195–204, New York, NY, USA, 2012. ACM.

[12] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2008.

[13] E. Brockmeyer, M. Miranda, and F. Catthoor. Layer assignment techniques for low energy in multi-layered memory organisations. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 1070–1075, 2003. DATE.

[14] F. Catthoor, K. Danckaert, K. Kulkarni, E. Brockmeyer, P. Kjeldsberg, T. v. Achteren, and T. Omnes. *Data access and storage management for embedded programmable processors*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[15] F. Catthoor, E. d. Greef, and S. Suytack. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[16] J. Cong, K. Guruaj, M. Huang, S. Li, B. Xiao, and Y. Zou. Domain-specific processor with 3d integration for medical image processing. In *IEEE Intl. Conf. on Application-Specific Systems, Architectures and Processors*, pages 247 –250, sept. 2011.

[17] J. Cong, M. Huang, and Y. Zou. Accelerating fluid registration algorithm on multi-fpga platforms. In *Proceedings of International Conference on Field Programmable Logic and Applications*, FPL. IEEE, 2011.

[18] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473 –491, april 2011.

[19] J. Cong, P. Zhang, and Y. Zou. Optimizing memory hierarchy allocation with loop transformations for high-level synthesis. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1229 –1234, june 2012.

[20] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. *IEEE Trans. Comput.*, 54(10):1242–1257, 2005.

[21] P. Diniz, M. Hall, J. Park, B. So, and H. Ziegler. Bridging the gap between compilation and synthesis in the defacto system. In *LCPC'03*, pages 52–70. 2003.

[22] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Program.*, 21(5):389–420, 1992.

[23] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Intl. J. of Parallel Programming*, 34(3), 2006.

[24] A. Grosslinger. Precise Management of Scratchpad Memories for Localising Array Accesses in Scientific Codes. In *Compiler Construction*, pages 236–250, 2009.

[25] A.-C. Guillou, F. Quilleré, P. Quinton, S. Rajopadhye, and T. Risset. Hardware design methodology with the Alpha language. In *FDL'01*, Lyon, France, Sept. 2001.

[26] Q. Hu, P. G. Kjeldsberg, A. Vandecappelle, M. Palkovic, and F. Catthoor. Incremental hierarchical memory size estimation for steering of loop transformations. *ACM Trans. Des. Autom. Electron. Syst.*, 12, September 2007.

[27] F. Irigoin and R. Triolet. Supernode partitioning. In *ACM SIGPLAN Principles of Programming Languages*, pages 319–329, 1988.

[28] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt. Drdu: A data reuse analysis technique for efficient scratch-pad memory management. *ACM Trans. Des. Autom. Electron. Syst.*, 12, April 2007.

[29] M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *Design Automation Conference, 2002. Proceedings. 39th*, pages 628–633, 2002.

[30] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *ACM SIGPLAN'97 Conf. on Programming Language Design and Implementation*, pages 346–357, Las Vegas, June 1997.

[31] Q. Liu, G. A. Constantinides, K. Masselos, and P. Cheung. Combining data reuse with data-level parallelization for fpga-targeted hardware compilation: A geometric programming framework. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(3):305–315, 2009.

[32] M. Palkovic, F. Catthoor, and H. Corporaal. Trade-offs in loop transformations. *ACM Trans. Des. Autom. Electron. Syst.*, 14:22:1–22:30, April 2009.

[33] P. R. Panda, N. D. Dutt, and A. Nicolau. Local memory exploration and optimization in embedded systems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 18:3–13, January 1999.

[34] PolyOpt: A complete source-to-source Polyhedral Compiler, http://www.cse.ohio-state.edu/∼pouchet/polyopt.

[35] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO'07)*, pages 144–156, 2007.

[36] B. So, M. W. Hall, and P. C. Diniz. A compiler approach to fast hardware design space exploration in fpga-based systems. In *Proc. of the ACM SIGPLAN Conference on Programming language design and implementation (PLDI'02)*, pages 165–176. ACM, 2002.

[37] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 327–337, 2009.

[38] S. Verdoolaege. isl: An integer set library for the polyhedral model. In *Mathematical Software - ICMS 2010*, pages 299–302, 2010.

[39] M. Wolf and M. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN'91 Conf. on Programming Language Design and Implementation*, pages 30–44, New York, June 1991.

[40] M. Wolfe. Iteration space tiling for memory hierarchies. In *3rd SIAM Conf. on Parallel Processing for Scientific Computing*, pages 357–361, Dec. 1987.