Improving Polyhedral Code Generation for High-Level Synthesis

Wei Zuo[1,5] Peng Li[2,3] Deming Chen[5] Louis-Noël Pouchet[4,3] Shunan Zhong[1] Jason Cong[4,3]
[1]Beijing Institute of Technology
[2]Peking University
[3]PKU/UCLA Joint Research Institution for Science and Engineering
[4]University of California, Los Angeles
[5]University of Illinois at Urbana-Champaign

ABSTRACT

High-level synthesis (HLS) tools are now capable of generating high-quality RTL codes for a number of programs. Nevertheless, for best performance aggressive program transformations are still required to exploit data reuse and enable communication/computation overlap. The polyhedral compilation framework has shown great promise in this area with the development of HLS-specific polyhedral transformation techniques and tools.

However, all these techniques rely on polyhedral code generation to translate a schedule for the program's operations into an actual C code that is input to the HLS tool. In this work we study the changes to the state-of-the-art polyhedral code generator CLooG which are required to tailor it for HLS purposes. In particular, we develop various techniques to significantly improve resource utilization on the FPGA. We also develop a complete technique geared towards effective code generation of rectangularly tiled code, leading to further improvements in resource utilization. We demonstrate our techniques on a collection of affine benchmarks, reducing by 2x on average (up to 10x) the area used after high-level synthesis.

Categories and Subject Descriptors

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

Keywords

Polyhedral Compilation; High-Level Synthesis; Loop tiling

1. INTRODUCTION

High-level synthesis (HLS) software tools such as AutoESL and its successor Xilinx Vivado-HLS [2] are capable of taking an input C program and generating effective RTL with performance that can rival manual design [14]. However, even for a sub-class of programs with regular loop bounds and array accesses (namely, *affine programs* [18]), additional efforts are still required in complement to Vivado-HLS capabilities to get the best performance – in particular to exploit data reuse opportunities in the program, generate off-chip communications, and overlap communication and computations [13, 24, 28].

Recent work focusing on the polyhedral compilation model has shown great promise in automating those tasks. For instance, Alias et al. have shown how loop tiling can be exploited effectively to design a multi-buffer execution of affine programs [7]. Pouchet et al. recently presented an end-to-end system using the polyhedral model which automatically transforms the program for effective data reuse, including the handling of on-chip buffers [24]. Zuo et al. perform inter- and intra-block optimizations for high-level synthesis using polyhedral loop transformations [28]. These works make extensive use of the polyhedral compilation model, a powerful program representation that leverages strong mathematical foundations to model arbitrarily complex sequences of loop transformations in a single, well-designed optimization stage [18, 19].

One key component of the polyhedral compilation framework that needs to be adapted in the context of high-level synthesis is *polyhedral code generation*. This is the process where C code is generated from an optimized polyhedral representation, that implements the loop transformations set by the user. It has been the subject of intense research in the past two decades, but with a very strong emphasis on x86 CPU execution. When generating code for CPUs, we devote our attention to the code segments where the largest fraction of total time is spent, mostly ignoring segments which account for a small fraction of the computation time. On the other hand, for FPGAs these non-critical code segments could very well require more chip surface than the hot spots, and therefore need special attention too.

In this work we study how to tailor the process of polyhedral code generation for HLS and FPGA mapping purposes, with the objective of minimizing resource usage without any performance (e.g., latency) penalty. Our paper creates a solid basis for future work using the polyhedral optimization framework on top of HLS tools. We make the following contributions.

- We provide a comprehensive study of the area, performance, power and energy of various techniques for code generation in the polyhedral model. In particular, we study the tuning of arithmetic operators generated during complex loop tiling, and two alternative methods to generate loop bounds for tiled programs.
- We show that using our techniques, we can fine-tune the state-of-the-art polyhedral code generator CLooG for HLS purpose, reducing by 2x on average (up to 10x) the area used.

The paper is organized as follows. Sec. 2 provides background and motivation for our work. Sec. 3 recalls the principles of polyhedral program generation and high-level synthesis. Sec. 4 presents a collection of techniques to reduce resource usage for polyhedral codes mapped on FPGAs. Sec 5 presents an alternative loop tiling technique suited for HLS. Extensive experimental results on eleven benchmarks are presented in Sec. 6 before we conclude in Sec. 7.

2. BACKGROUND AND MOTIVATION

Optimizing programs using the polyhedral model is a three-stage process. First, a mathematical representation (made from polyhedra and matrices) of the input program is computed from its original abstract syntax tree. Second, a program transformation is computed in the form of a schedule for each *dynamic* instance of each syntactical statement in the program. Third, this schedule is applied to the polyhedral program representation, and a new AST that represents a program implementing this new execution order is generated. This last stage is called *polyhedral code generation*, and is the focus of the present work.

Polyhedral code generation has been the subject of much previous work in the past two decades [8, 23, 22, 25], culminating with the development of CLooG, a generic and scalable code generator [10, 1]. CLooG has been the de facto state-of-the-art code generator for almost a decade because of its effectiveness and generic support of arbitrary affine schedules. However, the focus for its development has been mostly about effective program execution on CPUs. That is, objectives such as reducing possible interference with branch predictors (through the avoidance of conditionals in innermost loops), or reducing the number of branches taken to execute a computation have been used to drive its development.

The advent of HLS systems has triggered more aggressive uses of the polyhedral model in the compilation process [7, 16, 11, 28, 24]. But numerous FPGA optimization metrics differ significantly from CPU optimization metrics with regard to polyhedral code generation quality. For FPGAs, controlling the resource usage (i.e., the number of LUTs, DSPs, etc.) is critical to reduce energy and/or to enable hardware replication of a functional block to reduce the total execution time. *Resource sharing* is key for FPGAs, and has to be achieved by using different program structures than those for CPUs. The impact of complex loop bounds as generated by polyhedral transformations can be very significant too.

All those factors drive the need for a dedicated polyhedral code generation strategy tailored for high-level synthesis purposes. Table 1 use two benchmarks to illustrate how all those aspects can be effectively tuned for HLS. We consider two stencil benchmarks, Seidel-2D and Jacobi-2D, which have been tiled for effective on-chip data reuse. In their untiled variant, as shown later in Fig. 1 for Seidel-2D, these codes use only two arithmetic operations in the loop bound expressions. Applying loop tiling and the required pre-transformations to make tiling legal dramatically increases the complexity of loop bounds, as shown by the # ops column in Table 1, which reports the number of $+$, $-$, $/$, $*$, *ceil*, $floor$, *min* and *max* operations used in loop bounds. The performance of the code obtained by current polyhedral tools using CLooG 0.18.0, out-of-the-shelf [1] is shown as Default. The area metric is a normalized expression of the resource used by the program (LUT, FF, DSP), the lower the better; the latency is reported in milliseconds (ignoring off-chip communication time). Xilinx Vivado-HLS was used to compute this data. With careful tuning of the code generation process proposed in this paper, significantly better metrics are achieved, as shown by the Tuned rows.

Table 1: Impact of tuning polyhedral code generation for HLS

	# Stmts	# `for`	# `if`	# ops	Area	Latency(ms)
Seidel-Default	1	6	0	115	0.699	1395.78
Seidel-Tuned	1	6	0	86	0.129	1377.65
J2D-Default	13	12	12	394	0.533	120.08
J2D-Tuned	2	7	2	137	0.087	103.51

Recent work on polyhedral code generation has focused primarily on parametric tiling, a code generation process where the tile sizes are not known [26, 21, 9]. However, those approaches still suffer from drawbacks when implementing the program on FPGAs, such as code size explosion and/or very complex loop bound expressions. Other work such as CLooG-VHDL focused on the generation of VHDL from a polyhedral representation [17], operating only on a subset of affine programs. In contrast, our work is applicable to any polyhedral program that can be input to CLooG. In the following we perform an extensive evaluation of several techniques to improve the resource usage and latency of polyhedral programs, leveraging key aspects of HLS such as resource sharing opportunities.

3. PRELIMINARIES

In the following, we outline the key features of polyhedral code generation and high-level synthesis.

3.1 Polyhedral Code Generation

To illustrate the underlying ideas behind program transformations in the polyhedral model and polyhedral code generation, we use the code in Fig. 1 as a driving example.

```
for (t = 0; t < TSTEPS; ++t)
  for (i = 1; i < N - 1; ++i)
    for (j = 1; j < N - 1; ++j)
R:     A[i][j] = (A[i-1][j-1]+A[i-1][j]+A[i-1][j+1]
                 +A[i][j-1]+A[i][j]+A[i][j+1]+A[i+1][j-1]
                 +A[i+1][j] + A[i+1][j+1])/9.0;
```

Figure 1: Seidel-2D

Iteration domain. The iteration domain of a syntactic statement (`R` in Fig. 1) precisely captures the set of dynamic instances of the statement during the program execution. It is modeled using a (parametric) polyhedron $\mathcal{D}_R$, whose bounding hyperplanes are an affine form of the surrounding loop iterators (`t`, `i`, and `j` for `R`) and constants whose value are unknown at compile-time (`TSTEPS` and `N` here). The iteration domain of `R` is:

$$\mathcal{D}_R : \{(t,i,j) \mid 0 \le t < TSTEPS \wedge 1 \le i,j < N-1\}$$

The set $\mathcal{D}_R$ contains exactly one integer point per executed instance of `R`, and this point has as coordinates the value taken by the surrounding loop iterators when the instance is executed. Visually, $\mathcal{D}_R$ is a 3D rectangle of size roughly $TSTEPS \times N \times N$. In the polyhedral model, each syntactic statement in the program is associated with its iteration domain. These are input to the polyhedral code generation process, together with an order in which instances (e.g., each point in each iteration domain) have to be executed.

Scheduling function. The order in which instances are executed is modeled with a (multidimensional) affine function. Each statement has its own scheduling function, which associates a (multi-dimensional) timestamp to each point in the iteration domain. For instance, in order to make loop tiling valid, it is first required to reorder the execution of the instances of `R`. This is achieved by the following schedule Θ_R:

$$\Theta_R(t,i,j) = (t, t+i, 2t+i+j)$$

This scheduling function assigns to each 3-dimensional point (t,i,j) in $\mathcal{D}_R$ another 3-dimensional point whose coordinates are computed by the function Θ_R. One may note that as Θ_R is restricted to be an affine form of t, i and j, then Θ_R can be written in the form of a matrix.

Code generation. For a given program, the iteration domain and scheduling function of each statement are input to the polyhedral code generator. The goal is to generate a new AST, made of `for` loops, `if` conditionals, and the statements, which scan each iteration domain in the order described by the schedule. Loop transformations are implicitly modeled during this process: for example, if instances from two statements are scheduled at the same time, then they will share a common surrounding loop scanning those instances in the generated code, thereby achieving loop fusion.

The process is as follows. First, the new schedule is embedded in the original iteration domain $\mathcal{D}_R$ to create a new iteration domain $\mathcal{D}_R^{'}$ such that the lexicographic order of points in $\mathcal{D}_R^{'}$ is strictly identical to the ordering specified by Θ_R. Then, a valid AST (using loops) is generated, scanning $\mathcal{D}_R^{'}$ in lexicographic order. This two-step approach allows the design of a single, schedule-independent code generation primitive that scans all points in each input polyhedra in lexicographic order, provided that the user-defined schedule of operations has been integrated in the original iteration domain first. For instance, using the example above, we get:

$$\begin{aligned}\mathcal{D}_R^{'} : \{(c0, c1, c2) \quad | \quad & 0 \leq t < TSTEPS \wedge 1 \leq i, j < N-1 \\ & \wedge c0 = t \wedge c1 = t+i \wedge c2 = 2t+i+j\}\end{aligned}$$

Visually, $\mathcal{D}_R^{'}$ is a 3D parallelogram obtained by "skewing" $\mathcal{D}_R$ along two dimensions. Syntactic code scanning each of the new domains is then generated. Starting from the outermost dimension (e.g., $c0$ in our example), each new iteration domain is projected onto this dimension. The range of this projection (e.g., $0 \leq c0 < TSTEPS$) determines the loop bound for this dimension. If there are multiple statements, and/or the projection is not contiguous, then possibly many loops are generated, each scanning a contiguous set of points in the projection. This process of creating multiple loops to cope with having possibly different statements covering different ranges in the projection is called *separation* [25, 10], and is key to effective CPU code generation.

The alternative is to generate a single loop that iterates on the hull of the projection, and use `if` conditionals inside this loop body to skip loop iterations that do not correspond to a dynamic statement instance to be executed. This leads to code with less loops being generated, at the cost of having inner conditionals that are frequently evaluated. In contrast, separation leads to a higher number of loop nests, but with a lower number of conditionals inside them. This has been shown to be a significantly more effective approach for modern CPUs [10]. However, we show in Sec. 4 that this approach is often detrimental for HLS.

Once the (list of) range(s) on the outermost dimension has been computed, the process is recursively applied on the next dimension (e.g., $c1$ in our example) [10]. That is, for each range of $c0$ obtained previously, we project out $\mathcal{D}_R^{'}$ along the $c1$ dimension and repeat the process of possibly separating the projections into multiple contiguous regions. There is only one range for $c0$ in our example, and we get $c0+1 \leq c1 \leq c0+N-2$ for the range of $c1$. Fig. 2 shows the result of this algorithm when applied to a simple, single-statement program.

```
for (c0 = 0; c0 <= (TSTEPS + -1); c0++)
  for (c1 = c0 + 1; c1 <= c0 +N-2; c1++)
    for (c2 = c0 + c1 + 1; c2 <= c0+c1 + N -2; c2++) {
      i = ((-1* c0) + c1);
      j = (((-1 * c0) + (-1 * c1)) + c2);
R:    A[i][j] = (A[i-1][j-1]+A[i-1][j]+A[i-1][j+1]
                 +A[i][j-1]+A[i][j]+A[i][j+1]+A[i+1][j-1]
                 +A[i+1][j] + A[i+1][j+1])/9.0;
     }
```

Figure 2: Seidel-2D after skewing to enable tilability

One may note that as output, we obtain a transformed program where loop skewing has been applied on two of the loops. As a result, now all data dependences have positive components, and all loops are permutable; therefore, tiling can now be applied on all loops, including the time loop `t` [12].

Artifacts of polyhedral code generation. Despite its elegant expression, polyhedral code generation can very quickly lead to the generation of extremely complex codes. This is illustrated with the Jacobi-2D example in Sec. 2 where a transformation that achieves loop fusion, skewing, and multidimensional tiling leads to a code containing 13 loops and 12 conditionals. This complexity mainly comes from the following points.

First, when the projection along a given dimension cX is not a simple inequality (i.e., $l \leq cX$), but a conjunction of inequalities (i.e., $l_1 \leq cX \wedge l_2 \leq cX$), then min/max expressions are generated to correctly capture that the loop bound is a conjunction of expressions (i.e., $min(l_1, l_2) \leq cX$). Such conjunctions arise quickly with tiled programs with more than one statement, and/or programs with parametric loop bounds.

Second, when applying loop tiling in the polyhedral model, one popular approach is to insert new dimensions in $\mathcal{D}^{'}$ to model the loops scanning the set of tiles created; these are called inter-tile loops, or tile loops. The loop bounds for the existing dimensions are adapted so that they scan only the body of a tile; they are called intra-tile loops, or point loops. Inter-tile loops execute a fraction of the loop's original iteration range. That is, if the loop originally iterates N times and the tile size is 32, then a loop with $N/32$ iterations (the tile loop) is created, and a loop with 32 iterations (the point loop) is created. This creates division expressions in the loop bounds, in conjunction with ceil and floor operations.

Third, when separation is applied, numerous loop nests are generated to capture all possible cases of statement combinations. Intuitively, for a tiled program, we will have cases where all statements are executed inside a given tile, or where only one of the statements is in the tile, where the tile is a full rectangle, or where it is only a partial rectangle, etc. Most of those cases are corner cases that arise very infrequently in the program execution, but for which resources on a FPGA will be required.

Our objective in this work is to control and tailor those artifacts so as to minimize their impact on the FPGA design, in particular in terms of resource usage and total latency.

3.2 High-Level Synthesis

High-level synthesis (HLS) plays a central role in boosting the productivity in hardware accelerator design by automatically transforming the high-level untimed algorithmic description to low-level cycle-accurate RTL specifications. First, the input program is translated into a control/dataflow graph (CDFG) by a compiler front-end. Some common optimization techniques, including dead-code elimination, constant propagation and common subexpression elimination, could be performed on the intermediate representation.

Operation scheduling and resource binding are the key steps at the heart of high-level synthesis. During the operation scheduling step, HLS tools will determine in which cycle each operation will occur. The scheduling objective is optimized considering various design constraints including dependency constraints, timing constraints and resource constraints. The resource allocation step determines the amount and type of resources, and binding determines which operations use which resources. For example, HLS tools will automatically determine if two add operations will share the same adder by time-multiplexing or use two separated adders. Since the decisions in the resource allocation step can influence the scheduling of operations, sometimes the two steps are iterated [15].

Due to the inherently parallel nature of synthesized hardware architecture, HLS tools have different design trade-offs than software compilers. For example, divergent branches in the innermost loop, which could possibly flush the pipeline of a processor, will not be a problem for a system generated by the HLS tool.

4. FINE-TUNING CLOOG FOR HLS

We now present a collection of alternative syntactic code generation schemes, each targeting a particular artifact of polyhedral code generation. We focus our work on CLooG, and our reference point uses the default setting of CLooG [1]. All developments proposed are tuned for this software. To evaluate the impact of our techniques, we use as input programs a collection of numerical benchmarks from the PolyBench/C 3.2 suite [5], which are tiled using the tiling hyperplane method implemented in the Pluto software [12]. We particularly focus on five benchmarks for the detailed analysis of each individual optimization presented in this section before reporting their combined impact for eleven benchmarks in Sec. 6.

4.1 Optimization Metrics

In order to quantify the quality of a design, we use a set of FPGA-specific metrics that will be collected for each program variant we evaluate. Those metrics are as follows.

For the area, we consider the number of LUTs, FFs, slices, BRAMs and DSPs. Modern FPGA devices contain a heterogeneous mixture of different kinds of hardware blocks. Lookup table (LUT) elements give the FPGA devices the flexibility to implement arbitrary digital logic using truth tables. Most commonly used hardware elements, such as registers, memory blocks and arithmetic function units, are integrated as dedicated hardware blocks such as flip-flop (FF), block ram (BRAM) and digital signal processor (DSP) units in modern FPGA devices. For the performance, we consider the critical path and execution cycles. The frequency of a FPGA design is determined by the delay of its longest path, usually referred to as the critical path (CP). The overall latency is the product of CP and the number of execution cycles. For power, both dynamic power and static power are considered. Experimental results show that the static power of Xilinx Virtex-6 VLX75T varies very little (from 1290.24 mW to 1295.23 mW) while the dynamic power can change from 1.97 mW to 309.93 mW. The total energy consumption is also reported for our complete experiments in Sec. 6.

4.2 Experimental Setup

Generating program variants. Our framework is based on PoCC, the Polyhedral Compiler Collection [4], which includes both CLooG and Pluto. We particularly consider four stencil computations from PolyBench: Jacobi-1D (J1D) is a 1D 3-point iterative Jacobi process, Jacobi-2D (J2D) is a 2D 5-point iterative Jacobi process, Seidel-2D (Seid) is a 2D 9-point iterative Seidel process, and FDTD-2D (FDTD) is a finite-domain time difference discrete solver. In addition, we use a matrix-multiplication kernel GEMM. These five codes each benefit from aggressive tiling and have a large data reuse potential; and the four stencils pose numerous code complexity challenges when the required transformations for tiling have been applied. The loop bounds of GEMM remain constant numbers with simple rectangular shape after loop tiling, and we use this benchmark to validate that our techniques are not detrimental in the case of simple, regular benchmarks. For all we chose a tile size of $5 \times 20 \times 20$, enough to exploit most of the available data reuse. Additional benchmarks are used in later Sec. 6. For each tiled program, we perform code generation using different techniques described later in this section. Each variant obtained (possibly through combining several techniques) is synthesized as described below.

We use problem sizes of 20 for the time loop of stencils and 500 for each remaining loop in the programs. We note that the final FPGA design after HLS would not differ much if we used larger problem sizes: the core computation is a tile (of size $5 \times 20 \times 20$). Increasing the problem size only changes the number of iterations of the outer loops in the tiled program (those which iterate on the set of tiles), therefore it has little to no impact on the design.

Program synthesis. The optimized code segments are then synthesized and implemented using high-level synthesis, logic synthesis and physical implementation tools. The Xilinx Virtex-6 FPGA device, Xilinx Vivado 2012.3, and Xilinx ISE 14.3 tools are used in our experiments. Various optimization techniques such as constant propagation, common sub-expression elimination and global value numbering are automatically applied by the LLVM compiler used in Vivado-HLS.

Area utilization (represented by the number of LUTs, FFs and DSPs used) and the critical path delay are reported by ISE after place-and-route. Execution cycles are reported by a cycle-accurate SystemC simulator with the target circuit and test bench as the input. During the simulation, switching activities of each wire are traced by the simulator using value change dump (VCD) files.[1] Power data is reported by the Xilinx XPower tool with the place-and-routed circuits and the circuit simulation traces as the input.

4.3 Polyhedra Separation

The first technique we investigate relates to the issue of choosing a code structure with more loop nests and less conditionals – that is, using *polyhedral separation* – versus choosing a code structure with less loop nests but more (inner) conditionals – that is, no separation is used. CLooG offers options to control separation, and we evaluate the impact of turning it on (e.g., separation happens at all loop levels, which is the default setting) or off.

The benefits of turning off separation are: (1) it reduces the code size, and reduces the number of syntactic statements; (2) as statements are syntactically closer, typically under the same loop nest, the HLS tool has a better opportunity for resource sharing between the computing elements of the statements (they are typically made of numerous fixed/floating point arithmetic operations); (3) fewer loop nests will possibly reduce the area consumed by complex loop bound calculation. On the other hand, the drawbacks of turning off separation are: (1) the creation of conditionals that have to be evaluated for each dynamic instance of a statement (i.e., $T \times N^2$ for Seidel-2D); (2) loops with possibly more iterations than strictly needed; and (3) more pipeline stages added to the loop body. Table 2 shows the impact of separation on four representative benchmarks.

Table 2: Impact of separation

	LUT	FF	DSP	CP(ns)	Cycle	Pwr(mW)
FDTD-sep	39803	24532	56	11.313	12622094	1378.49
FDTD-nosep	25595	16692	40	8.965	11500269	1452.56
Gemm-sep	1822	1532	14	7.609	14567698	1302.43
Gemm-nosep	1822	1532	14	7.609	14567698	1302.43
J1D-sep	11926	7586	14	8.435	7638461	1384.48
J1D-nosep	11327	7350	14	8.100	5724101	1411.92
J2D-sep	24818	15351	35	8.990	13356949	1435.77
J2D-nosep	14216	10103	27	8.582	21800977	1427.02
Seid-sep	32561	19599	9	8.763	159281806	1459.67
Seid-nosep	32561	19599	9	8.763	159281806	1459.67

First, for the case of Seidel-2D and GEMM, turning on or off separation has no impact: a single loop nest is being generated, identical with or without separation. For all other cases, we observe

[1]To save the simulation time and space, only the first 10ms of the entire trace is recorded.

a strong benefit in terms of resource savings. However, the impact on the latency varies from benchmark to benchmark. Despite both being stencils, opposite results are observed for Jacobi-2D and FDTD-2D. This is due to the two contrary effects introduced by turning off polyhedra separation. On the one hand, turning off separation will reduce the number of loop nests, which could reduce the cost of starting and ending outer loop nests. On the other hand, turning off separation will introduce branch conditions in the innermost loop nest, which could possibly increase the pipeline stages. The extra cycles to fill and drain longer pipelines is non-trivial considering the small innermost loop trip counts after loop tiling. For Jacobi-2D, complex conditions with division in the innermost loop increases the pipeline stages from 33 to 77. Techniques described in Sec. 4.4 will greatly bridge this gap.

4.4 Loop Bounds Tuning

We now study a collection of alternatives to generate the loop bound expressions. In particular, we study 1) the impact of hoisting common expressions in loop bound computations as early as possible, 2) the impact of using integer or bit operations in place of the default floating-point operations generated by CLooG, and 3) alternatives to compute conjunctions of constraints.

Hoisting and CSE. We first conducted experiments to hoist the loop bound computations as early as possible, and perform basic common subexpression elimination. Production compilers using SSA representation like LLVM use global value numbering (GVN) [3] to eliminate obviously redundant expression computations, such as those generated by polyhedral code generation. Our experiments were aimed at validating that Vivado-HLS was able to hoist and simplify loop bounds as expected through hoisting and GVN passes. We observed that in all cases, performing hoisting and common subexpression elimination in the input program did not result in any improvement in any of the metric. We therefore concluded that Vivado-HLS was already able to perform these optimizations effectively.

We also point out that Vivado-HLS takes as input a C program where all loop bounds are known at compile-time: they are expressions made of only compile-time constants and loop iterators (for instance `TSTEPS` is replaced by 20, `N` by 500, etc.). The required complex loop bound expressions using ceil/floor/min/max cannot be simplified further by Vivado-HLS compiler passes, thus motivating the techniques developed in this paper.

Division Optimization (dopt). In polyhedral code generation, four operations are heavily used, particularly for tiled codes. Those operations relate to the division of an expression by a constant integer value (taking either the ceil or floor of the result), typically the tile size, and conjunctions of expressions (taking either the min or max of various expressions). The default implementation of those functions in CLooG is shown below, with the mathematical operation on the left, and its C implementation on the right.

$\lceil \frac{x}{y} \rceil$	`ceil(((double)(x))/((double)(y)))`
$\lfloor \frac{x}{y} \rfloor$	`floor(((double)(x))/((double)(y)))`
$min(x,y)$	`((x) < (y)? (x) : (y))`
$max(x,y)$	`((x) > (y)? (x) : (y))`

These operator implementations are known to perform very effectively for CPUs. However, the use of floating point operations in loop bound computations is, of course, to be avoided as much as possible on FPGAs. We have evaluated a series of alternative implementations of these operators. First, IntDiv replaces the floating-point division by the appropriate integer division and offset (`x` and `y` are integer expressions, so the C semantics of `x/y` is an integer division):

$\lceil \frac{x}{y} \rceil$	`((x > 0)? (1 + (x - 1)/y): (x / y))`
$\lfloor \frac{x}{y} \rfloor$	`((x > 0)? (x / y): (1 + (x -1)/ y))`

Second, MulUB is a technique to scale the upper bound constraints to eliminate division whenever possible. That is, $cX \le \left\lfloor \frac{x}{y} \right\rfloor$ is replaced by $y \cdot cX \le x$. In the case of conjunction of constraints, the largest common multiple is used as the scaling factor.

Third, LcmLB is a technique to scale the lower bound constraints, to eliminate division whenever possible. That is,

$$\left\lceil \frac{x}{y} \right\rceil \le cX \ \wedge\ \left\lceil \frac{u}{v} \right\rceil \le cX$$

is replaced by

$$\left\lceil \frac{\max(\frac{lcm(y,v)}{y}x, \frac{lcm(y,v)}{v}u)}{lcm(y,v)} \right\rceil \le cX$$

$\frac{lcm(y,v)}{y}$, $\frac{lcm(y,v)}{v}$ and the least common multiple lcm(y,v) can be computed at compile time if y and v are constant numbers. We report the cumulative impact of these three techniques in Table 3.

Table 3: Impact of division optimization

	LUT	FF	DSP	CP(ns)	Cycle	Pwr(mW)
FDTD-nosep	25595	16692	40	8.965	11500269	1452.56
+ IntDiv	11319	8864	100	8.948	11487327	1365.44
+ MulUB	9210	7456	62	8.722	11486905	1366.24
+ LcmLB	9988	7572	50	8.668	11488085	1364.21
Gemm-nosep	1822	1532	14	7.609	14567698	1302.43
+ IntDiv	1822	1532	14	7.609	14567698	1302.43
+ MulUB	1822	1532	14	7.609	14567698	1302.43
+ LcmLB	1822	1532	14	7.609	14567698	1302.43
J1D-nosep	11327	7350	14	8.100	5724101	1411.92
+ IntDiv	2759	1897	17	8.148	3194601	1312.37
+ MulUB	2495	1787	16	8.853	3094421	1312.42
+ LcmLB	2496	1835	15	7.946	3094951	1312.11
J2D-nosep	14216	10103	27	8.582	21800977	1427.02
+ IntDiv	6312	4749	39	8.757	12408718	1318.99
+ MulUB	5221	4164	32	8.596	12150238	1319.23
+ LcmLB	4968	4148	31	8.428	12148886	1320.37
Seid-nosep	32561	19599	9	8.763	159281806	1459.67
+ IntDiv	8844	5909	33	8.460	159265028	1369.15
+ MulUB	7934	5227	18	7.908	159265296	1366.94
+ LcmLB	7573	5125	16	8.039	159265212	1368.42

A floating point division requires more than 3K LUTs and more than 3K FFs, while an integer point division will use 4 DSPs (with very little LUT and FF elements) after transforming the division to multiplication [6]. As expected, we observe a dramatic improvement in terms of LUT and FF required when floating-point operations are replaced by equivalent integer operations. However, this leads to a significant increase in DSP consumption.

The increase in the number of DSP required is largely compensated by the reduction in all other resources, reducing the total energy consumed by the system. Without any penalty on latency, these techniques bring consistent improvements on resource usage. While it may happen that optimizing lower bounds slightly increases resource usage, we nevertheless choose to systematically apply all these techniques because of their potential benefit.

Balanced min/max tree. The third technique we evaluated is a simple reorganization of conjunctions in a tree, versus a sequence. Currently, for a bound of the form

$$cX \le A \ \wedge\ cX \le B \ \wedge\ cX \le C \ \wedge\ cX \le D$$

CLooG generates $cX \le min(min(min(A,B),C),D)$. We instead generate $cX \le min(min(A,B),min(C,D))$. Table 4 summarizes the results. These improvements, albeit marginal, further reduce resource usage in all cases.

Table 4: Impact of balanced min/max tree (bmt)

	LUT	FF	DSP	CP(ns)	Cycle	Pwr(mW)
FDTD-ns-dopt	9988	7572	50	8.668	11488085	1364.21
+ bmt	9746	7542	50	8.273	11486733	1365.45
Gemm-ns-dopt	1822	1532	14	7.609	14567698	1302.43
+ bmt	1822	1532	14	7.609	14567698	1302.43
J1D-ns-dopt	2496	1835	15	7.946	3094951	1312.11
+ bmt	2496	1835	15	7.946	3094951	1312.11
J2D-ns-dopt	4968	4148	31	8.428	12148886	1320.37
+ bmt	4968	4148	31	8.428	12148886	1320.37
Seid-ns-dopt	7573	5125	16	8.039	159265212	1368.42
+ bmt	7495	5097	16	8.348	159265212	1370.78

16-bit iterators. The last technique leverages the fact that we can determine, at compile time, precisely the integer range of each loop iterators. In all the benchmarks we tested, the range of all iterators will not exceed $2^{16} = 65536$. Therefore, we can use fewer bits to represent iterators, e.g., changing iterators from 32-bit integers to 16-bit integers. This modification is useless for modern x86 CPUs, yet quite useful for HLS-generated code, especially on FPGA platforms. Typical hardware multiplier blocks in modern FPGA devices are a 25×18 multiplier. Our experiments show that 16-bit integer multiplication will use two less DSPs and two less pipeline stages than 32-bit integer multiplication. The range of each iterator can be precisely analyzed given all parameter values at compile-time using standard polyhedral analysis. The experimental results are shown in Table 5.

Table 5: Impact of short iterators

	LUT	FF	DSP	CP(ns)	Cycle	Pwr(mW)
FDTD-dopt-bmt	9746	7542	50	8.273	11486733	1365.45
+ short iterator	7259	5780	51	8.986	11357291	1369.41
Gemm-dopt-bmt	1822	1532	14	7.609	14567698	1302.43
+ short iterator	1731	1464	14	7.603	14568594	1302.78
J1D-dopt-bmt	2496	1835	15	7.946	3094951	1312.11
+ short iterator	1882	1493	15	7.727	3094421	1312.78
J2D-dopt-bmt	4968	4148	31	8.428	12148886	1320.37
+ short iterator	3698	2913	25	8.453	11760982	1320.35
Seid-dopt-bmt	7495	5097	16	8.348	159265212	1370.78
+ short iterator	6094	3922	10	8.268	159264736	1372.26

From the results, we can see that using short iterators will benefit both area and performance (as well as total energy). One exception is the extra DSP consumed by FDTD after applying the technique. After comparing the RTL code generated, we found that the reason for this is that the same expression $5 * i$ is synthesized into a shifter-and-adder in one version and a multiplier in the other version. We believe that design trade-offs by HLS tool will change according to the scarcity of various resources.

5. TILING EXPERIMENTS

We now investigate the impact of two different tiling methods. First we recall the basics of loop tiling as it is implemented in PoCC (that is, what we used to generate the input codes in the previous section). Then, we present an alternative strategy to simplify the loop bounds.

Loop tiling (a.k.a. blocking) is a powerful loop transformation that partitions the iteration space into regular blocks (the tiles), so that each tile can be executed in isolation from the rest of the computation [27]. Polyhedral loop tiling is performed by extending the iteration domain of the statements to be tiled with additional dimensions (e.g., loops) to model the strip-mining and interchange of loops to be tiled [19]. *Tile loops* are the loops iterating on the set of tiles. *Point loops* are the loops iterating on each point of the iteration space inside a given tile.

Previous work has focused on improving the quality of tiled code, especially in the context of parametric tiling where the tile sizes are not known at compile-time. One way is to produce a *bounding box* of the iteration space and use guards to test whether the iteration to be executed actually belongs to the original iteration space. This method can introduce many "empty tiles" – particularly for complex iteration domain shapes. Other work uses approaches based on syntactic separation [21] and scanning only non-empty tile origins [26]. However, none of these fit the requirements of our problem: the code size quickly grows out of control with syntactic separation [21], and the inset/outset method does not exhibit a parallel tile schedule [26]. Goumas et al. developed a rectangular tiling technique, however it is restricted to perfectly nested loops [20].

5.1 Sub-Bounding-Box Tiling

5.1.1 Overview

Our strategy takes the advantage of the bounding box idea, and tailors it to the properties of polyhedral rectangular tiling. Our method separately considers the tile loops (inter-tile) and point loops (intra-tile). For tile loops, the key idea of our method is to use a rectangular grid superimposed on the iteration domain to be tiled as the tiling structure. Each point in this grid represents a tile origin, and we aim for the appropriate *parallelogram* (or hyper-parallelogram, for domains of more than two dimensions) that contains the set of all tile origins to be executed. This idea is reminiscent of parametric tiling [9], but we tailor it to fixed-size tiling. Then the planes bounding this parallelogram become simple affine expressions of other dimensions and constants. Therefore min/max operations are not needed and loop bounds are simplified. This may introduce some extra points in the tile loop domain in the form of empty tiles (tiles that are scanned but do not contain any point of the original iteration domain), but for the benchmarks we evaluated carefully, choosing the parallelogram shape leads to very few empty tiles. Fig. 3 is the original tile iteration domain and its parallelogram approximation of Seidel-2D when looking at dimensions $c1$ and $c2$.

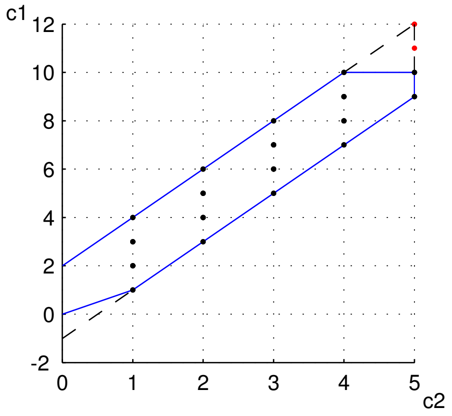

Figure 3: Tile origin iteration space and its approximation

The points inside the solid lines are the actual tile origins. The dashed lines show the new parallelogram bounds found by our method. By adding three more tile origins (top right and bottom left), the shape is now much simpler to describe.

For the point loops, we simplify the boundary by using a rectangular grid to support each tile. This is possible since the loops tiled are permutable. To make sure that the generated point loops only scan points in the original iteration domain, we construct the new boundary with the intersection of the tile boundaries and the original iteration domain boundaries.

5.1.2 *Sub-bounding Box Algorithm*

Our approach works by taking as input the polyhedral representation using standard polyhedral tiling (which will be altered by our algorithm) that is fed to CLooG; this includes the polyhedral schedule for all loops/dimensions. The algorithm is divided into two parts. First, generate the tile loop boundaries. At this step, we find the parallelogram that approximates the tile origin space, and combine the new parallelogram domain with the scheduling functions given as input. Second, generate the point loop boundaries to scan the iteration points within the tile in the order specified by the (arbitrary, user-specified) schedule. At this step, point loop boundaries are made of the intersection of the tile bounds and the iteration domain bounds.

New tile origin iteration domain. Vector $\vec{s}$ represents the tile size, where s_i is the tile size along dimension i. We construct the diagonal matrix $L = diag(\vec{s})$. We use $tile(L\vec{x})$ to represent the set of points in the tile whose origin is $L\vec{x}$.

$$tile(L\vec{x}) = \{\vec{z} | L\vec{x} \leq \vec{z} < L\vec{x} + \vec{s}\}$$

where $\vec{z}$ represents the points within a tile. We define the set of tile origins as the set of points $L\vec{x}$ such that $L\vec{x}$ is the origin of the tile which is either partial tile or full tile.

$$D_{tile} = \{L\vec{x} | tile(L\vec{x}) \cap D_R\}$$

where D_R is the iteration domain.

Parallelogram hull of the tile origin domain. The key idea is to find the smallest parallelogram that contains D_{tile}, that is, the parallelogram hull of D_{tile}. It may lead to the inclusion of tile origins for which the corresponding tile does not intersect with the program iteration domain (empty tiles). To compute this parallelogram hull, we first generate the code scanning D_{tile} using CLooG. For each loop bound using more than one expression (that is, there is a min/max expression in the associated loop bound), we select as the unique lower (resp. outer) loop bound the expression that has the largest (resp. smallest) value when the variables involved in the expression (that is, the surrounding loop iterators) are set to a value in the middle of their range. This computation is possible by operating from outermost to innermost loops, and because the value of parameters such as `TSTEPS` or `N` is known at compile-time.

The method is implemented as follows. First, we construct the tile origin iteration domain, and generate the code scanning it in lexicographic order using CLooG. Then, going from the outermost loop level to innermost, we calculate the new loop bounds. These loop bounds form the parallelogram hull of D_{tile}, $phull(D_{tile})$. Finally, we assign the scheduling function for tile loops initially provided as input to CLooG for this program, and generate the code scanning $phull(D_{tile})$ with CLooG.

Generating the point loops. Once the tile origins are computed, we need to generate the code scanning the iteration points within one tile. To ensure that it only scans points in the original iteration space, the bounds are made up of the intersection of the tile bounds and the iteration space bounds.

Since we are using rectangular tiling, the code generation for point loops is simple, and also reminiscent of parametric tiling [9]. We use a rectangular grid to model tiling, where the grid spacing equals the tile size. First we generate the code scanning the input iteration domain without tiling (that is, it implements the schedule given to CLooG). Then we post-process the generated code such that, for each lower bound lb_i and upper bound ub_i, we add the tile lower bound $s_i * t_i$ and upper bound $s_i * t_i + s_i - 1$, where s_i is the tile size of the i-th dimension, and t_i is the tile origin iterator for dimension i, generated at the previous stage. Finally, the lower bound and upper bound are written in the format of $max(lb_i, s_i * t_i)$ and $min(ub_i, s_i * t_i + s_i - 1)$. The complete code is created by making this modified code the body of the innermost loop generated during the tile loop code generation mentioned above.

Fig. 4 illustrates this process: dots on the lower left corner of each tile represent tile origins, other dots represent iterations inside each tile. The dashed lines bounds the iteration domain. The boundary is the intersection of the iteration domain bounds and the tile grid.

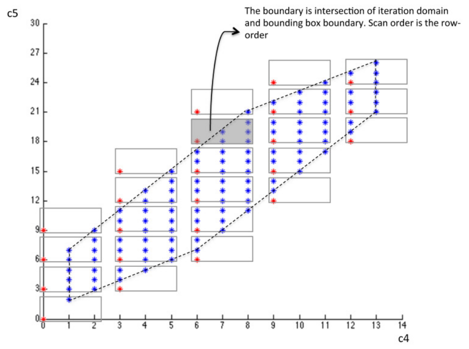
The boundary is intersection of iteration domain and bounding box boundary. Scan order is the row-order
c5
c4

Figure 4: Optimization for point loop

5.2 Evaluation

To evaluate the quality of the generated code using the proposed sub-bounding box algorithm, we performed two sets of experiments. We integrated the sub-bounding box algorithm on top of the short iterator optimization result, which is in Table 6. We can see that based on top of the other optimizations presented in Sec. 4, the sub-bounding box algorithm achieves about 15% reduction in LUTs and FFs required on average, and 22% reduction for DSPs, for a slight decrease in power (about 1%) without performance loss.

Table 6: Impact of tiling bounding-box method

	LUT	FF	DSP	CP(ns)	Cycle	Pwr(mW)
Seid-s-itr	7133	4366	9	7.704	159135759	1361.17
+ BB	5882	3637	5	7.874	159143274	1352.77
J2D-s-itr	4037	3237	24	8.244	11761063	1320.35
+ BB	3236	2514	19	7.943	11752537	1300.22
J1D-s-itr	2004	1558	15	8.068	3094952	1294.78
+ BB	1808	1475	15	8.12	3074938	1295.32
FDTD-s-itr	7259	5780	51	8.986	11357291	1369.41
+ BB	6956	5549	50	8.768	11354663	1368.06

As there are similarities between the proposed sub-bounding-box method and parametric tiling code generation, we also make a comparison with PTile, a tool included in PoCC which generates parametric tiling [9]. Parametric tiling is designed to allow the runtime selection of the tile size, a feature which requires heavy loop structures to be generated. We note that this feature is of no obvious interest for HLS, as the loop bounds need to be determined at compile-time. Nevertheless, PTile does use the idea of a rectangular supporting grid for the tiling, and a convex hull approximation of the tile origin domain, motivating the comparison with the sub-bounding box method.

Parametric tiled code was generated with PTile and we set the symbolic tile sizes to be constants ($5 \times 20 \times 20$). For fair comparison, as parametric tiling requires even more complex loop bound expressions that could not be optimized with the techniques presented in Sec. 4, we did not apply any of these optimizations anywhere. The results are in Table 7.

Table 7: Bounding-Box tiling versus PTile

	LUT	FF	DSP	CP(ns)	Cycle	Pwr(mW)
Seid+BB	6653	4310	19	8.12	159016038	1354.29
Seid+PTile	10787	7140	23	8.992	162090843	1361.80
J2D+BB	5134	3995	32	8.212	12011882	1298.89
J2D+PTile	8626	5919	23	8.938	21647016	1301.98
J1D + BB	2706	1959	18	7.909	3125028	1296.98
J1D + PTile	5337	3814	17	8.625	8512131	1296.18
FDTD + BB	8386	6571	51	8.949	11225174	1297.99
FDTD + PTile	10728	8123	22	8.996	35509389	1303.84

The results show that compared to parametric tiling, the sub-bound-ing box method uses less resources and is much faster. This is because PTile uses more complex loop bounds for symbolic tile sizes, including numerous floating point operations. Not only do they increase the resource cost, but they also make the dependence analysis in the HLS tool very conservative, this leads to an increased initiation interval (II) for the pipeline. In our experiment, the II for the sub-bounding-box is 3, while for parametric tiling it is 6. We conclude that a direct use of the PTile software out-of-the-box would not be an effective method for tiled code generation.

6. EXPERIMENTAL RESULTS

Based on the analysis in previous sections, we learned that turning off polyhedra separation, optimizing division operations, using hierarchical min/max operations, short iterations, and simplifying loop bounds using sub-bounding box tiling, could benefit polyhedral code generation for high-level synthesis. In this section, all these optimization techniques are integrated and applied on a set of eleven computation kernels and applications.

6.1 Experiments Setup

We use a set of benchmarks from PolyBench/C 3.2 [5], with small datasets (array sizes are typically 500 in each dimension) and a tile size of $5 \times 20 \times 20$, which is large enough to exploit the vast majority of the data reuse in all benchmarks we evaluated. As programs are tiled, larger problem sizes would simply lead to increasing the number of tiles visited, by increasing the iteration count of the outer (tile) loops; this is not expected to have any significant impact on the computed design. In the experiments, only the computation module is evaluated. Off-chip communication time is ignored, and memory utilization is omitted in the experimental results. Indeed, effective techniques to reduce off-chip communication while reducing the on-chip buffer size have been developed in previous work [24], and these considerations are orthogonal to the present work. Double precision floating point is used to be the main data type in computations as in the original code. A description of each benchmark can be found in Table 8. In all versions, the innermost loops are pipelined. Macro `USE_SCALAR_LB` is defined, and Vivado-HLS inter-loop dependence pragmas are added.

6.2 Complete Results

Table 8 describes all the raw data reported by the tool-chain including the various resource usage, critical path, execution cycles and power consumption. We generated codes 1) using CLooG with default parameters, 2) using CLooG but turning off polyhedra separation option, and 3) using CLooG with all optimization techniques described in this paper (mentioned as Default, NoSep and HLSOpt versions thereafter).

In addition to these raw metrics, we also use Eq. (1) and Eq. (2) to measure the total latency and energy consumed by the target circuit with the given inputs. Experimental results show that for the benchmark selected in this paper, a large fraction of power consumed is static power, ranging from 76.4 % to 99.8%. If only a portion of the target FPGA device is used, the rest of the device could possibly be used by some other modules. Considering this, the static power of the whole FPGA device can be conceptually divided between multiple modules according to the area utilization. Here $\max\left(\frac{LUT_{used}}{LUT_{Avail.}}, \frac{FF_{used}}{FF_{Avail.}}, \frac{DSP_{used}}{DSP_{Avail.}}\right)$ is used as the metrics to represent the proportion of the FPGA device used by a certain module. The area utilization is overestimated as we select the largest ratio between used and available hardware resources, computed individually for LUTs, FFs and DSPs. Using this notion, we use a *Normalized Energy* metric in this paper to reflect the quality of the generated circuit with area, performance and power information included (Eq. (4)). These metrics are computed as follows.

$$\begin{aligned}
Latency &= Critical\_Path * Execution\_Cycles & (1)\\
Energy &= (Pwr_{Static} + Pwr_{Dyn.}) * Latency & (2)\\
Area_{ratio} &= \max\left(\frac{LUT_{used}}{LUT_{Avail.}}, \frac{FF_{used}}{FF_{Avail.}}, \frac{DSP_{used}}{DSP_{Avail.}}\right) & (3)\\
Energy_{norm} &= (Area_{ratio} * Pwr_{Static} + Pwr_{Dyn.}) * Lat. & (4)
\end{aligned}$$

Latency, energy, area Ratio and normalized Energy calculated for the three implementations for all the benchmarks are shown in Fig. 5-8.

Latency. As explained in Sec. 4.3, turning off polyhedra separation could possibly increase or decrease the execution latency for different benchmarks. The latency of the HLSOpt version is consistently shorter than the NoSep version. For LU, HLSOpt is slightly slower than the Default version by 3.2% due to longer critical path. For other benchmarks, HLSOpt can reduce the execution latency by 0.1%(GEMM) to 61.8%(J1D) when compared to Default. The average latency reduction over the 11 benchmarks is 17.5%.

Energy. Since energy is proportional to both latency and power, the NoSep version could increase or decrease the total energy. However, HLSOpt is consistently more energy efficient. Total energy reduction of HLSOpt over Default ranges from 0.1% (GEMM and Lug) to 63.8% (J1D), with an average of 21%.

Area Ratio. Experiment results in Fig. 7 show that the techniques proposed in this paper are quite efficient for area reduction. The area reduction of HLSOpt vs. Default is 0(GEMM) to 91.4%(Dynp) with an average of 55.1%. NoSep uses consistently less area than Default, with an average of 17.2% area reduction.

Normalized Energy. Considering the significant area reduction, energy reduction is much more noticeable after separating the static power according to the area consumption. Results in Fig. 8 show a maximum reduction in normalized energy consumption of 93.6% (Dynp), for an average reduction of 59.8%.

7. CONCLUSION

Despite significant progress achieved by high-level synthesis tools, numerous subsequent transformations of the input C code are still

Table 8: Experimental results

Benchmark	Description	Version	LUT	FF	DSP	CP(ns)	Cycle	Static Pwr	Dyn. Pwr
		Default	7542	7279	34	8.993	174425051	1292.34	96.76
ADI	Alternating Direction Implicit solver	NoSep	7542	7279	34	8.993	174425051	1292.34	96.76
		HLSOpt	6843	6167	24	8.484	170436051	1292.31	95.39
		Default	14485	9636	14	8.892	8410401	1292.47	102.60
dynprog	Dynamic programming (2D)	NoSep	11326	7299	7	7.904	7494801	1292.40	99.38
		HLSOpt	1399	1033	8	8.271	6048301	1290.24	1.97
		Default	39803	24532	56	11.313	12622094	1292.11	86.38
FDTD-2D	2-D Finite Different Time Domain Kernel	NoSep	25595	16692	40	8.965	11500269	1293.73	158.84
		HLSOpt	6956	5549	50	8.768	11354663	1291.88	76.18
		Default	1822	1532	14	7.609	14567698	1290.46	11.97
GEMM	Matrix-multiply C = α.A.B + β.C	NoSep	1822	1532	14	7.609	14567698	1290.46	11.97
		HLSOpt	1731	1464	14	7.603	14568594	1290.46	11.97
		Default	30096	20175	124	8.000	20565307	1295.23	226.26
Gradient-2D	Jacobi-like 2D stencil	NoSep	17798	13730	114	7.918	29568426	1294.59	197.61
		HLSOpt	7164	6559	117	7.917	18997282	1292.28	94.01
		Default	11926	7586	14	8.435	7638461	1292.24	92.24
Jacobi-1D	Jacobi 1D stencil	NoSep	11327	7350	14	8.1	5724101	1292.84	119.08
		HLSOpt	1790	1461	15	7.988	3084421	1290.65	20.79
		Default	24818	15351	35	8.990	13356949	1293.36	142.41
Jacobi-2D	Jacobi 2D stencil	NoSep	14216	10103	27	8.582	21800977	1293.17	133.85
		HLSOpt	3236	2514	19	7.943	11752537	1290.86	29.94
		Default	26288	17141	46	7.975	13773376	1293.70	157.71
Laplacian-2D	Jacobi-like 2D stencil	NoSep	14908	11087	25	7.894	22460226	1293.64	155.13
		HLSOpt	3371	2779	25	7.661	11753289	1291.16	43.79
		Default	12086	9177	14	8.553	1990925	1292.84	118.92
LU	LU decomposition	NoSep	11718	8794	14	8.814	2102483	1292.88	120.73
		HLSOpt	5622	5151	14	8.854	1984462	1291.85	74.52
		Default	5918	5135	14	8.932	4073294	1292.74	114.62
ludcmp	LU decomposition	NoSep	5564	4964	14	8.140	4079524	1292.71	113.09
		HLSOpt	5525	5011	14	7.741	4079524	1292.53	105.08
		Default	32561	19599	9	8.763	159281806	1293.88	165.79
Seidel-2D	Seidel 2D stencil	NoSep	32561	19599	9	8.763	159281806	1293.88	165.79
		HLSOpt	5882	3637	5	7.874	159143274	1291.93	78.02

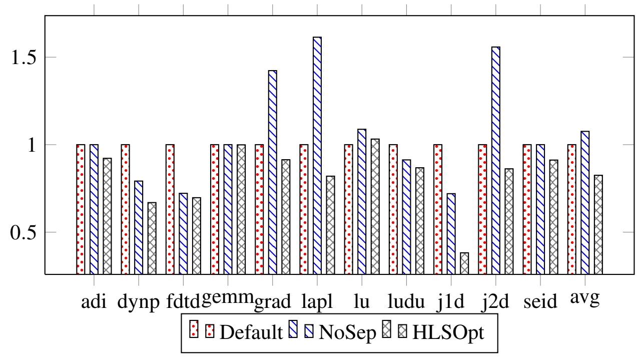

Figure 5: Latency (normalized to default)

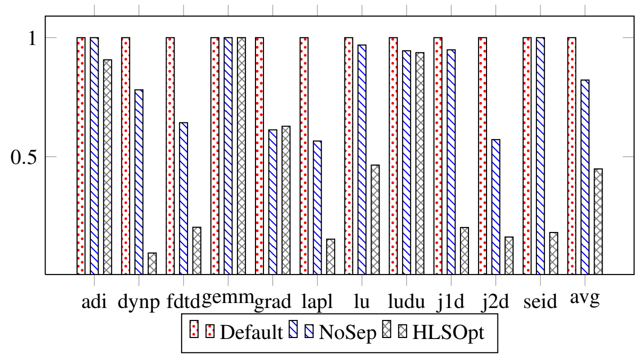

Figure 7: Area ratio (normalized to default)

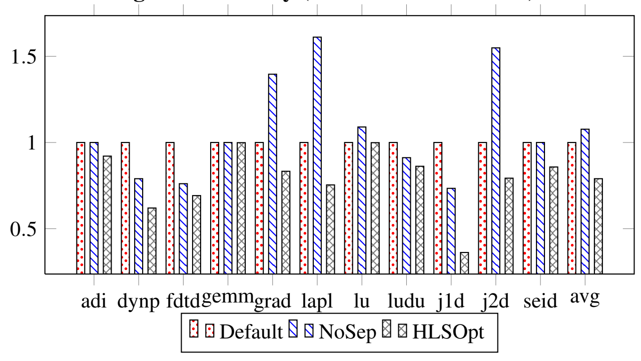

Figure 6: Energy (normalized to default)

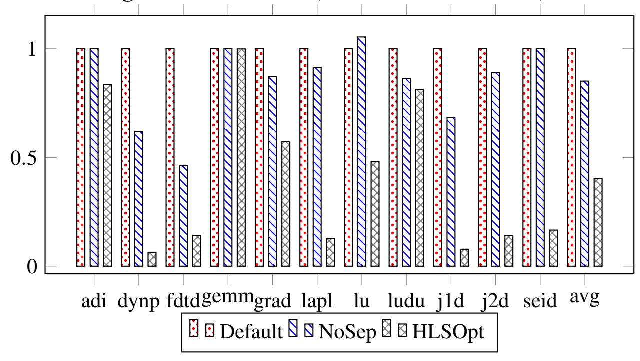

Figure 8: Normalized energy (normalized to default)

required to effectively exploit data reuse potential in the applications, and coarse- and fine-grain parallelism. The polyhedral compilation model has shown great promise in automating those tasks. However, the expressiveness and flexibility of this compilation framework comes at a cost: the programs obtained after polyhedral code generation usually contain very complex loop bounds. The code structure typically contain a hot spot loop nest and numerous other loop nests that correspond to infrequently executed cases. This is not an issue when operating on modern x86 CPUs, but is a major issue for FPGAs: extra resources are consumed for each case, including the ones with an extremely low contribution to the total execution time. Therefore, it is necessary to tailor polyhedral code generation to the specifics of FPGA execution for best results.

In this paper we investigated several techniques to reduce the resource usage for codes that are automatically generated by a polyhedral compilation framework. Focusing on Vivado-HLS and CLooG, two state-of-the-art tools, we extensively studied the impact of alternative loop bound computation techniques, and ended with systematic and significant resource savings when compared to off-the-shelf use of CLooG.

Acknowledgments. This work was supported in part by the Center for Domain-Specific Computing (CDSC) funded by NSF "Expeditions in Computing" award 0926127; China Postdoctoral Science Foundation (2012M520115); and the SRC/DARPA CFAR center. We are grateful to Xilinx, Inc. for equipment donations and financial contributions.

8. REFERENCES

[1] CLooG 0.18 .0.http: //www.cloog.org.

[2] http://www.xilinx.com/products/design-tools/vivado/index.htm.

[3] LLVM project. http://www.llvm.org.

[4] Pocc 1.2. http://pocc.sourceforge.net.

[5] Polybench 3.2. http://polybench.sourceforge.net.

[6] Xilinx dsp: Designing for optimal results. http://www.xilinx.com/publications/archives/books/dsp.pdf.

[7] C. Alias, A. Darte, and A. Plesco. Optimizing DDR-SDRAM communications at C-level for automatically-generated hardware accelerators an experience with the Altera C2H HLS tool. In *Intl. Conf. on Application-specific Systems Architectures and Processors (ASAP'10)*, pages 329 –332, July 2010.

[8] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *3rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 39–50, June 1991.

[9] M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *CGO*, April 2010.

[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] J. Cong, M. Huang, and Y. Zou. Accelerating fluid registration algorithm on multi-fpga platforms. In *Proc. of Intl. Conf. on Field Programmable Logic and Applications (FPL'11)*. IEEE, 2011.

[14] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 30(4):473–491, Apr. 2011.

[15] J. Cong, B. Liu, and J. Xu. Coordinated resource optimization in behavioral synthesis. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE'10)*, pages 1267–1272, 2010.

[16] J. Cong, P. Zhang, and Y. Zou. Combined loop transformation and hierarchy allocation for data reuse optimization. In *Proceedings of the 2011 IEEE/ACM International conference on Computer-aided design*, ICCAD. IEEE, 2011.

[17] H. Devos, K. Beyls, M. Christiaens, J. Campenhout, E. H. D'Hollander, and D. Stroobandt. Transactions on high-performance embedded architectures and compilers i. chapter Finding and Applying Loop Transformations for Generating Optimized FPGA Implementations, pages 159–178. Springer-Verlag, 2007.

[18] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *Intl. J. of Parallel Programming*, 21(6):389–420, Dec. 1992.

[19] 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.

[20] G. Goumas, M. Athanasaki, and N. Koziris. An efficient code generation technique for tiled iteration spaces. *Parallel and Distributed Systems, IEEE Transactions on*, 14(10):1021–1034, 2003.

[21] A. Hartono, M. Baskaran, J. Ramanujam, and P. Sadayappan. Parametric tiled loop generation for effective parallel execution on multicore processors. In *IPDPS*, 2010.

[22] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Intl. Symp. on the frontiers of massively parallel computation*, pages 332–341, McLean, VA, USA, Feb. 1995.

[23] M. Le Fur. Scanning parameterized polyhedron using Fourier-Motzkin elimination. *Concurrency - Practice and Experience*, 8(6):445–460, 1996.

[24] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong. Polyhedral-based data reuse optimization for configurable computing. In *21st ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'13)*, Monterey, California, Feb. 2013. ACM Press.

[25] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Intl. J. of Parallel Programming*, 28(5):469–498, Oct. 2000.

[26] L. Renganarayanan, D. Kim, S. Rajopadhye, and M. M. Strout. Parameterized tiled loops for free. *SIGPLAN Notices, Proc. of the 2007 PLDI Conf.*, 42(6):405–414, 2007.

[27] M. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley Publishing Company, 1995.

[28] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong. Improving High Level Synthesis Optimization Opportunity Through Polyhedral Transformations. In *Proc. of the ACM/SIGDA Intl. Symp. on Field Programmable Gate Arrays (FPGA'13)*, 2013.