Making Domain-Specific Hardware Synthesis Tools Cost-Efficient

Nithin George* David Novo* Tiark Rompf†‡ Martin Odersky† Paolo Ienne*

*Processor Architecture Laboratory, École Polytechnique Fédérale de Lausanne, Email: first.last@epfl.ch

†Programming Methods Laboratory, École Polytechnique Fédérale de Lausanne, Email: first.last@epfl.ch

‡Oracle Labs, Email: first.last@oracle.com

***Abstract*—Tools to design hardware at a high level of abstraction promise software-like productivity for hardware designs. Among them, tools like Spiral, HDL Coder, Optimus and MMAlpha target specific application domains and produce highly efficient implementations from high-level input specifications in a *Domain Specific Language* (DSL). But, developing similar domain-specific *High-Level Synthesis* (HLS) tools need enormous effort, which might offset their many advantages. In this paper, we propose a novel, cost-effective approach to develop domain-specific HLS tools. We develop the HLS tool by embedding its input DSL in Scala and using *Lightweight Modular Staging* (LMS), a compiler framework written in Scala, to perform optimizations at different abstraction levels. For example, to optimize computation on matrices, some optimizations are more effective when the program is represented at the level of matrices while others are better applied at the level of individual matrix elements. To illustrate the proposed approach, we create an HLS flow to automatically generate efficient hardware implementations of matrix expressions described in our own high-level specification language. Although a simple example, it shows how easy it is to reuse modules across different HLS flows and to integrate our flow with existing tools like LegUp, a C-to-RTL compiler, and FloPoCo, an arithmetic core generator. The results reveal that our approach can simultaneously achieve high productivity and design quality with a very reasonable tool development effort.**

I. Introduction

Tools to design hardware at a high-level of abstraction can enhance the productivity of FPGA designers. Among them, tools such as Spiral [1], HDL Coder [2], Optimus [3] and MMAlpha [4] target specific application domains in which they make designing hardware more accessible to their users. To achieve this, they utilize a high-level *Domain-Specific Language* (DSL) for the input specifications, like SPL for Spiral, Matlab/Simulink design for HDL Coder, StreamIt for Optimus and Alpha for MMAlpha, which is more natural to express the applications they target. Additionally, many of these tools also leverage on the domain information to optimize the hardware designs they generate. A simple tool for synthesizing matrix expressions into hardware can help us to understand these advantages: Using a DSL that uses concepts like matrices and operations on matrices makes expressing the input design easy. Additionally, by knowing that the input is a matrix expression, the tool can apply well known rules of matrix algebra to optimize the design and achieve better results compared to general-purpose tools. Application domain experts, who often have little knowledge of designing hardware, can use these domain-specific tools effectively to create designs targeting FPGAs. These tools can, hence, make the benefits of reconfigurable technology more accessible to users within different domains.

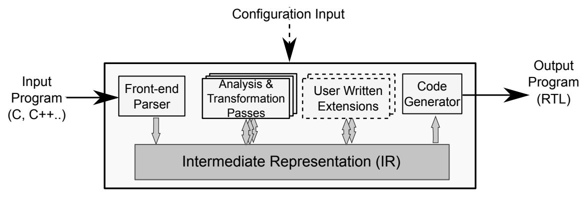
However, developing a new domain-specific HLS tool incurs enormous effort since one would need to design a new DSL, a custom compiler and, sometimes, even a development environment; developing just the compiler itself would involve writing a parser, multiple analysis and optimization steps and the output code generators. This high design effort makes such tools impractical in many application areas, despite their advantages. To reduce this design effort, we propose developing the domain-specific hardware synthesis flow in Scala [5] by embedding the input DSL in it and using *Lightweight Modular Staging* (LMS) [6], a compiler framework written in Scala, to perform analysis, optimizations and code generation. To reduce the effort needed to build the compiler, we propose building it in a modular fashion by employing multiple reusable optimization modules. These optimization modules can then be easily reused in a completely different HLS flow with very little effort. To further reduce the effort, we also build the HLS tool incrementally by integrating closely with external tools when possible.

As an illustration of this approach, we create an HLS flow to synthesize matrix expressions into hardware. This flow is composed of two optimization modules, one performing optimizations at the matrix-level and the other at the level of matrix elements (scala-level). Using this flow, we demonstrate the flexibility of the approach by showing how we can easily reuse the optimization modules and by integrating it with external tools like LegUp [7], a C-to-RTL compiler, and FloPoCo [8], an arithmetic core generator. While we present the concepts using a specific HLS flow, they are indeed quite general and can be used to create domain-specific tool-chains and/or to augment existing tool-chains with domain-specific optimizations with a very reasonable development effort.

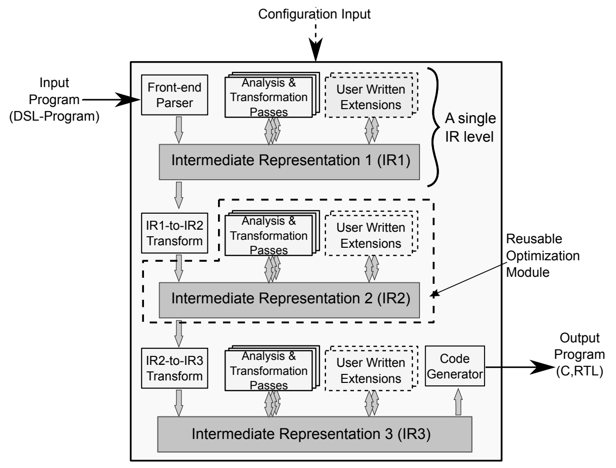
The remainder of this paper is organized as follows. Section II describes our modular design approach and Section III provides a brief introduction to LMS and our work-flow. In Section IV, we detail the development of the optimization modules. We evaluate our approach in Section V using a case study of implementing multiple matrix expressions in hardware. The results show that we can leverage high-level optimizations to considerably reduce the implementation area without significant performance loss. We then discuss related work in Section VI and Section VII concludes the paper.

This work was partially supported by the European Union program FP7-PEOPLE-2011-IEF, under Project 299405.

978-1-4799-2198-0/13/$31.00 ©2013 IEEE



(a) HLS Tool Design Using Traditional Compiler Frameworks


Configuration Input
Input Program (DSL-Program)
Front-end Parser
Analysis & Transformation Passes
User Written Extensions
A single IR level
Intermediate Representation 1 (IR1)
IR1-to-IR2 Transform
Reusable Optimization Module
Intermediate Representation 2 (IR2)
IR2-to-IR3 Transform
Code Generator
Output Program (C,RTL)
Intermediate Representation 3 (IR3)

(b) HLS Tool Design Using Our Approach

Fig. 1. Single IR vs multiple IR flows. Typical HLS tools use traditional compiler frameworks like LLVM or GCC which have a single IR-format (a). Instead, we propose a different architecture based on multiple IR-formats, each used by a unique optimization module, to enable a more efficient hardware synthesis (b).

II. Compiler Framework for Domain-Specific Hardware Synthesis

In this section, we discuss the advantages of LMS over popular open-source compiler frameworks, such as LLVM [9] and GCC [10], as a common platform for implementing multiple domain-specific synthesis tools.

Traditional Compiler Frameworks. Open-source compiler frameworks like LLVM and GCC have been used to build HLS tools like Vivado HLS [11], OpenCL to FPGA [12], LegUp [7] and Trident [13]. In these HLS tools, as shown in Figure 1(a), the input program is translated into an *Intermediate Representation* (IR) format by the front-end parser. Various analysis and transformation passes then optimize this IR, maintaining its original format, until the code generator finally uses it to produce the output RTL code. While HLS tool designers often extend the framework by adding new front-end parsers, analysis and transformation passes, and code generators, they usually use the same low-level IR-format used by the framework since changing it is not easy. By retaining the IR-format, they can reuse some of the standard optimizations that are already available in the framework, but it also introduces limitations.

For instance, many domain-specific tools, like Spiral [1] and MMAlpha [4], use multiple IR-formats to perform high-level, domain-specific optimizations at different levels of abstraction. One way to achieve this with a single IR is to perform the high-level, domain-specific optimizations directly at the front-end parser, before the low-level IR is generated. But, this can be quite tedious and since these optimizations are now a part of the parser, it will be hard to reuse some of them for a different HLS flow. An alternative is to perform the high-level optimizations using a sequence of optimizations on the low-level IR. But, this can be extremely difficult due to the low-level nature of the IR; for instance, performing simple optimizations using rules of matrix algebra is quite easy when the IR represents operations on matrices, but it becomes hard or just not feasible when the higher-level information is lost. Furthermore, with a single IR-format, there can also be complex interactions between the different optimization steps, making it harder to add new optimizations or reuse the existing ones selectively for an entirely new flow.

Proposed Approach. We use the extensible LMS compiler framework to define multiple IR-formats, as shown in Figure 1(b), and organize them into standalone *optimization modules*. Each optimization module uses a unique IR-format. It contains multiple analysis and transformation passes; an optional lightweight, front-end parser for the input DSL; and optional code-generators for the different output formats. Since all these components operate on its unique IR-format, we avoid the possibility of optimizations in different modules affecting each other, thereby, making it easier to make changes or add/remove optimizations. Additionally, as shown in the figure, we can develop complex HLS flows by connecting multiple optimization modules to each other using a special variant of a transformation pass called *IR-to-IR transform* [14]. Often, the IR-formats are common across different HLS flows, enabling a natural reuse of optimization modules.

In the software domain, the Delite project [15] employs a similar strategy to target code-generation for multi-core CPUs and GPUs starting from a DSL. In this paper, however, we demonstrate that our approach can dramatically lower the cost for developing domain-specific HLS flows to achieve efficient hardware generation. In particular, we show that this method of building domain-specific flows fit naturally into the current EDA ecosystem by enabling an easy integration of existing IP-cores and the extension of matured HLS flows by providing domain-specific optimizations.

Using the proposed approach, we can easily develop domain-specific synthesis tools like Spiral or MMAlpha by creating different optimization modules for each level of abstraction at which we want to apply optimizations. To illustrate this point, we use as an example a simple tool to synthesize matrix expressions into efficient hardware. We implement this flow using two optimization modules, one for performing optimizations at the matrix-level and the other for applying optimizations at the scalar-level. As done in many domain-specific tools, we also create a simple DSL to the make it easy for the end-user to express design specifications. Our matrix-level optimization module uses an IR at the matrix-level to reorder multiplications, apply the distributive property, and eliminate common subexpressions at the matrix-level. After that we use an IR-to-IR transform to decompose this matrix-level IR into a scalar-level one that represents operations on individual matrix elements. We then perform standard scalar-level optimizations, like strength reduction and common subexpression elimination, as done in many HLS tools before generating the output design. Although only a well understood example, it is general enough to show the potential of the proposed methodology.

III. Hardware Synthesis Using LMS

In this section, we first introduce the LMS framework and discuss how we use it to efficiently implement the optimization modules. We then describe our LMS-based workflow for hardware synthesis.

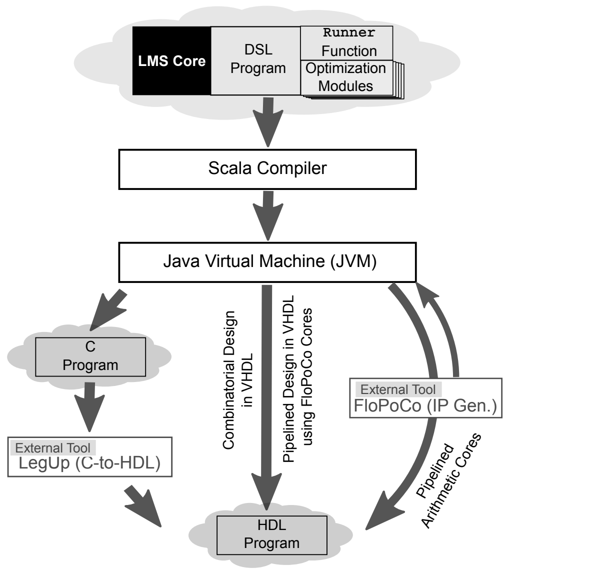


Fig. 2. An LMS-based workflow for hardware synthesis. The input program to the flow consists of the LMS library, the DSL-program, the custom optimization modules and the `Runner` function. This input is compiled using the *Scala* compiler to generate a Java byte-code which is then executed by the *Java Virtual Machine* to produce the optimized, low-level program in the user specified output format. Other tools can be integrated into this flow by interacting with them during this byte-code execution. We demonstrate this by integrating both LegUp and FloPoCo into our workflow.

A. The LMS Framework

Scala is a novel programming language that brings together functional and object-oriented programming concepts. More importantly, one can leverage Scala's syntax, rich type system and module system to embed other languages in it, giving us a very cost-effective way to provide a DSL for our HLS flow. LMS is a library and compiler framework written in Scala that lends the ability to optimize programs written using such an embedded DSL. More specifically, LMS enables one to represent the DSL-program in an *Intermediate Representation* (IR) format of our choice, a process referred to as *staging*, and by using multiple IR-formats that represent the DSL-program at different levels of abstraction we can progressively optimize it to achieve high quality results. To organize these optimizations into reusable modules, we leverage the modularity of the LMS framework and group them based on the IR-format they operate on. The framework then enables us to easily compose together these optimization modules and to connect them as required to implement custom HLS flows similar to the one shown in Figure 1(b).

To build a new HLS flow, the tool designer develops custom optimization modules and defines a `Runner` function that controls the application of optimizations in these modules to generate the optimized hardware design. While creating the optimization modules, the tool designer can specify the following:

- The DSL used to write the input specifications.
- The IR-format for representing the DSL-program.
- The transformation rules and how they are applied to optimize the IR for the DSL-program.
- The format for the output produced by the tool.

To keep the development effort low, the tool designer can reuse optimization modules from other flows and interface to external tools when it is both beneficial and possible.

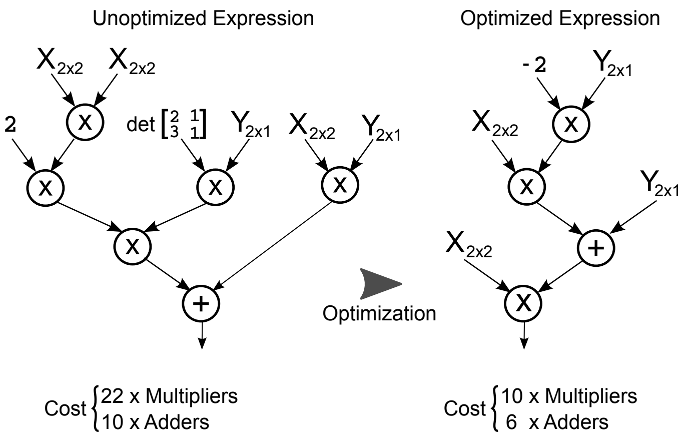


Fig. 3. The graph on the left shows the matrix expression as specified by the user. To implement this directly in hardware, it would need 22 multipliers and 10 adders. The graph on the right is the same expression after folding away constants and applying some matrix-level optimizations. Implementing this design needs only 10 multipliers and 6 adders.

B. Our Hardware Synthesis Workflow

Figure 2 shows the workflow we use for synthesizing hardware using LMS. The input to this flow is composed of the LMS library, the DSL program, a set of optimization modules that are needed for the specific HLS flow and the `Runner` function. This input is compiled using the Scala compiler into a byte-code format which is then executed on a standard *Java Virtual Machine* (JVM) to produce the output design.

When the JVM executes this byte-code, the execution control passes to the `Runner` function which then orchestrates the process of translating the high-level DSL-program into optimized hardware. In the flow we present, this function first uses the facilities in LMS to *stage* the DSL-program (i.e., converts it into the IR); coordinates all the optimization steps, first at the matrix-level and then at the scalar-level; and finally generates the optimized hardware design as output. The output design is generated in formats specified by the tool designer. To demonstrate this, we generate our outputs as a C program, as a combinatorial design in VHDL and as a pipelined design in VHDL that uses arithmetic cores from FloPoCo. The arithmetic cores used in the latter are automatically pipelined depending on the desired operating frequency.

IV. From Matrix Expressions to Hardware Designs

In this section, we highlight the features of LMS and how they are used to design our optimization modules. To serve as an example, we use a matrix-level optimization module that tries to reduce the resources needed to implement a matrix expression in hardware. But, the features presented here can be used to develop optimization modules for other purposes, like optimizing state-machines, DSP algorithms or streaming computation.

Figure 3 shows an example of the optimization performed by the matrix-level optimization module. The graph on the left computes $2 \cdot det(\left[\begin{smallmatrix} 2 & 1 \\ 3 & 1 \end{smallmatrix}\right]) \cdot X_{2\times2}^2 \cdot Y_{2\times1} + X_{2\times2} \cdot Y_{2\times1})$, where $X_{2\times2}$ and $Y_{2\times1}$ are variable matrices, and it needs 22 multipliers and 10 adders, without considering the cost to compute $det(\left[\begin{smallmatrix} 2 & 1 \\ 3 & 1 \end{smallmatrix}\right])$. Our module can transform this expression into the optimized form, shown on the right of the figure, implementing the same expression with only 10 multipliers and 6 adders.

```
//Datatype for matrices
type Mat
//Datatype for the matrix elements
type T = Int
//To create a Mat from Rep-type elements
def newMat(rows: Int, cols: Int, data: Rep[T]*):
     Rep[Mat]
//To create a Mat from non-Rep elements
def newcMat(rows: Int, cols: Int, data: T*): Mat
//Scalar-Matrix multiplication
def mult(x: Rep[T], y: Rep[Mat]): Rep[Mat]
//Matrix-Matrix multiplication
def mult(x: Rep[Mat], y: Rep[Mat]): Rep[Mat]
//Matrix-Matrix addition
def add(x: Rep[Mat], y: Rep[Mat]): Rep[Mat]
```

Fig. 4. Defining the datatypes and operators for the custom DSL (`MatrixDSL`) that enables users to write computation on matrices efficiently.

```
 1  def func(a: Rep[T], b:Rep[T], c:Rep[T],
 2           d:Rep[T], e:Rep[T], f:Rep[T])={
 3    val X = newMat(2,2, a,b,c,d)
 4    val Y = newMat(2,1, e,f)
 5    val C = newcMat(2,2, 2,1,3,1)
 6    val t1 = mult(X, X)
 7    val t2 = mult(2, t1)
 8    val t3 = mult(det(C), Y)
 9    add(mult(t2, t3), mult(X, Y)) // Return value
10  }
```

Fig. 5. `MatrixDSL`-program for the matrix expression on the left in Figure 3.

A. Designing an Optimization Module

As explained in Section III-A, there are four aspects to developing an optimization module. We will now see these aspects in the context of the matrix-level optimization module.

1) Specifying the DSL

To write matrix expressions efficiently, we can easily define a custom DSL, called *MatrixDSL*, by defining its data-types and operators, as shown in Figure 4; note that the syntax used here is the standard Scala syntax. In the figure, `Mat` and `T` are datatypes for matrices and matrix elements, respectively. `newMat` and `newcMat` are operators to compose matrices from scalar values of type `T`. Here, `mult` is an overloaded operator for multiplying matrices or a scalar with a matrix, and `add` specifies additions between matrices. The other data-types and operators of the DSL are defined in a similar manner.

The `Rep[]` in the code is a special type-qualifier defined by LMS, used to mark variables that will be part of the IR constructed by it. Operators such as `newMat`, `mult` and `add` that produce or use `Rep`-type variables also become part of this IR. In LMS, this process is called *staging*.

Now, we can express the matrix expression in Figure 3 in `MatrixDSL` as shown in Figure 5. Here, lines `2–4` construct the matrices $X_{2\times2}$, $Y_{2\times1}$ and the constant matrix, $C_{2\times2}$. Lines `5–8` express the computation on these matrices. As mentioned earlier, only those operations that either use or produce `Rep`-type variables are made part of the IR. Others, like `det(C)`, will be evaluated and replaced by its constant value in the IR. In this manner, LMS performs a partial-evaluation of the DSL-programs while staging it.

2) Specifying the IR

LMS provides the facility to represent DSL-programs in an IR-format that makes it easier to perform optimizations on it. The IR-format used by LMS is a directed *sea-of-nodes* graph

```
1  case class MulMM (x:Rep[Mat], y:Rep[Mat])
         extends Def[Mat]
2  case class MulSM (x:Rep[T], y:Rep[Mat])
         extends Def[Mat]
3  case class AddMM (x:Rep[Mat], y:Exp[Mat])
         extends Def[Mat]
4  override mult(x:Rep[Mat], y:Rep[Mat])=MulMM(x,y)
5  override mult(x:Rep[T], y:Rep[Mat])=MulSM(x,y)
6  override add(x:Rep[Mat], y:Rep[Mat])=AddMM(x,y)
```

Fig. 6. IR-nodes for some operators defined by `MatrixDSL`. These nodes are used in the IR-graph constructed while staging a `MatrixDSL`-program.

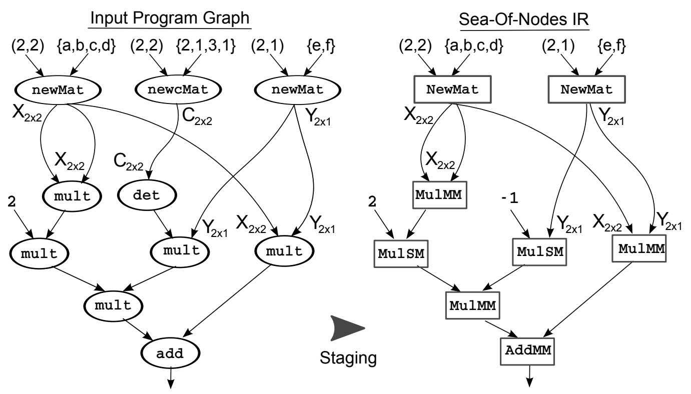


Fig. 7. *Staging* the DSL-program. This figure illustrates the *staging* process. On the left we have the function `func()` (defined in Figure 5) represented in a graph-format and on the right is the *sea-of-nodes* IR-format constructed from *staging* this function.

that captures the direct dependencies between operations; this makes it easy to analyze and specify optimization rules on it. To represent a `MatrixDSL`-program as an IR-graph, we specify the IR-nodes for the operators defined in `MatrixDSL`, as shown for the `mult` and `add` operators in Figure 6 (lines `1–3`). We also redefine the original `mult` and `add` operators (lines `4–6`) so as to instantiate the respective IR-nodes for each use of these operators. In addition to the IR-nodes for operators used in the DSL-program, we need to also define IR-nodes that may be created during our optimization steps. Figure 7 shows the graph format IR resulting from *staging* the code in Figure 5.

3) Performing Analysis and Transformation

Once the DSL-program is expressed as an IR-graph, we can perform analyses and optimizations on it. LMS provides two mechanisms to perform optimizations: *staging-time macros* and *IR-transformers*.

Staging-Time Macros. Staging-time macros are rewriting rules that are applied automatically as the IR-graph for the DSL-program is being constructed. Here, the tool-flow designer specifies IR-graph patterns and how those pattern must be rewritten. When any of these patterns appear in the IR-graph, the rewrite rule corresponding to it is automatically applied. LMS uses Scala's powerful pattern matching construct to make them easy to specify. Figure 8 shows some of the staging-time macros that are applied to the `mult` operators in the IR-graph on Figure 6. Figure 9 illustrates the steps through which these staging-time macros transform the initial IR-graph. Here, first the rules on lines `12–13` get applied to move the scalar multiplications $-1 \cdot Y_{2\times1}$ and $2 \cdot X_{2\times2}^2$ to occur after the matrix multiplication that initially succeeded it. Now, the rule on line `8` rewrites the two successive scalar multiplications in $2 \cdot (-1 \cdot X_{2\times2}^2 \cdot Y_{2\times1})$ into $(2 \cdot -1) \cdot X_{2\times2}^2 \cdot Y_{2\times1}$. Finally, the

```
 1  // Mult operator
 2  def mult(x:Rep[T], y:Rep[T]) = (x,y) match {
 3    // Constant propagation
 4    case(Const(m), Const(n)) => Const(m*n)
 5    ... }
 6  def mult(x:Rep[T], y:Rep[Mat]) = (x,y) match {
 7    // Constant propagation
 8    case(x, Def(MulSM(s, n))) => mult(mult(x,s),n)
 9    ... }
10  def mult(x:Rep[Mat], y:Rep[Mat]) = (x,y) match {
11    // Forward scalar multiply
12    case(x, Def(MulSM(s, n))) => mult(s,mult(x,n))
13    case(Def(MulSM(s, m)), y) => mult(s,mult(m,y))
14    ... }
```

Fig. 8. An example of Scala's powerful pattern matching construct used to specify optimization rules in staging-time macros.

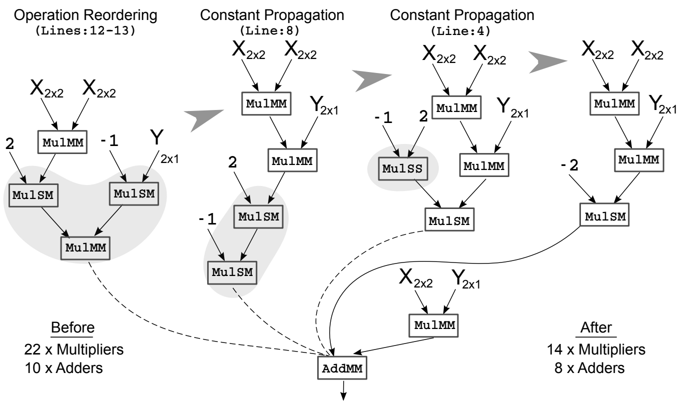


Fig. 9. Staging-time macros get automatically applied as the IR-graph for the user design is being constructed (step shown in Figure 7). This figure shows how staging-time macros progressively optimize one of the operands to the AddMM-node. The figure also shows the line numbers of the rules in Figure 8 that are being applied in each step. (The NewMat nodes in this graph have been substituted with just $X_{2\times2}$ and $Y_{2\times1}$ to keep the figure simple to understand).

rule on line 4 performs constant propagation, rewriting $2 \cdot -1$ as -2.

IR-Transformers. Staging-time macros are very convenient, but they only perform localized rewrites without having a global view of the program. Furthermore, the tool-flow designer has no explicit control on when they are applied. To overcome these limitations, LMS provides the ability to perform analysis passes and *IR-transformer* (which is similar to typical compiler pass). IR-transformers, unlike the staging-time macros, are called by the Runner function and can be preceded by analysis passes that collect additional information to give it a global view of the program. Additionally, they also use Scala's pattern matching syntax which makes them easy to specify.

Applying staging-time macros reduced the cost for implementing the initial matrix expression to 14 multipliers and 8 adders, as shown in Figure 9. We can now apply the distributive property of matrices to further reduce the cost, but this presents us with three options as shown in Table I. To select the best one, we must first use analysis passes to gather global information on the cost associated to each of these options. With this knowledge, we now use an IR-transformer to move just the multiplication with $X_{2\times2}$ after the addition to obtain the result shown in Figure 3. If we had to use staging-time macros here, without the global view, we might easily end up choosing one of the other alternatives, achieving only sub-optimal results.

In our HLS flow, we also use a variant of IR-transformers,

TABLE I. COMPARISON OF ALTERNATIVES AFTER APPLYING STAGING-TIME MACROS

Expression after applying staging-time macros	Multipliers	Adders
$-2 \cdot X^2_{2\times2} \cdot Y_{2\times1} + X_{2\times2} \cdot Y_{2\times1}$	14	8

Implementation alternatives	Multpliers	Adders
$X_{2\times2}(-2 \cdot X_{2\times2} \cdot Y_{2\times1} + Y_{2\times1})$	10	6
$(-2 \cdot X^2_{2\times2} + X_{2\times2})Y_{2\times1}$	16	10
$X_{2\times2}(-2 \cdot X_{2\times2} + \left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right))Y_{2\times1}$	12	8

```
// Produces "sym = x * y;" in the C code
case MulSS(x, y) =>
   stream.println("%s=%s*%s;".format(quote(sym),
        quote(x), quote(y)))
```

Fig. 10. This figure shows how the instances of MulSS-nodes are translated during code generation into C-code for scalar multiplications.

which we call IR-to-IR transformer, to connect different optimization modules to each other. For instance, to connect the matrix-level optimization module to the one at the scalar-level, we use an IR-transformer that rewrites each matrix-level IR-node using multiple scalar-level IR-nodes. This effectively represents the matrix-level computation using operations at the scalar-level enabling us to use standard scalar-level optimizations to further optimize the design. Using this technique, we can interconnect different optimization modules as needed while creating custom HLS flows.

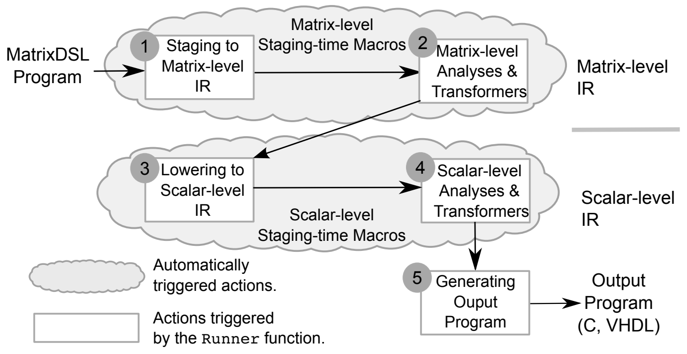
4) Generating the Optimized Design as Output

To produce an output, LMS traverses the IR-nodes in the optimized IR-graph in the topological order, i.e., a node is visited only after all the nodes it depends on have been visited, and translates it into the output design. Using this facility, we only need to specify the translation rules for the IR-nodes that appear in the optimized IR-graph to generate the optimized output. During code generation, since LMS only visits IR-nodes which affect the final output, it also performs dead-code elimination.

Our matrix-level optimization module can generate a C program from its IR-format. Figure 10 shows how the MulSS IR-node that represents scalar multiplication is translated into the equivalent C-code where the result of x*y is assigned to sym. An optimization module can have multiple sets of translation rules to support different output formats, like different variants of C-code, VHDL-code, and have specific interfaces to different external tools. For instance, our scalar-level optimization module produces both C and VHDL codes from its scalar-level IR-nodes. Additionally, during VHDL code-generation, it can generate a combinatorial design or interface with FloPoCo (an arithmetic core generator) to create the arithmetic components needed to generate a pipelined design. The module also uses the feedback from FloPoCo to know the pipeline stages in each generated component in order to produce a glue-logic containing registers to have a correctly pipelined datapath. We will not detail the development of the scalar-level optimization module separately since it uses the same LMS features that we have just described.

B. Orchestrating the Entire Flow

Figure 11 shows the different phases in transforming a high-level MatrixDSL-program into an optimized hardware design. Referring back to Figure 2, as the byte-code is being executed in the JVM, the execution control is handed over to the Runner function. This function then generates the IR-graph for the DSL-program at the matrix-level and calls the individual analysis and transformation steps in the matrix-level



MatrixDSL Program
1 Staging to Matrix-level IR
Matrix-level Staging-time Macros
2 Matrix-level Analyses & Transformers
Matrix-level IR
3 Lowering to Scalar-level IR
Scalar-level Staging-time Macros
4 Scalar-level Analyses & Transformers
Scalar-level IR
Automatically triggered actions.
Actions triggered by the Runner function.
5 Generating Ouput Program
Output Program (C, VHDL)

Fig. 11. This figure shows the steps involved in transforming a high-level `MatrixDSL`-program into optimized output program at the scalar-level.

optimization module. The staging-time macros in this module are always automatically applied whenever the one of its patterns appear in the IR-graph. Once the matrix-level optimizations are completed, the `Runner` function calls the IR-to-IR transform that converts the matrix-level IR into scalar-level IR. Then, the analysis and transform functions at the scalar-level are called to further optimize the DSL-program. Once again, the staging-time macros at the scalar-level get automatically applied whenever their control patterns appear in the graph. After the DSL-program has been optimized at this level, the `Runner` calls the code-generator to translate this scalar-level IR into the final output design.

V. Results and Discussion

In this section, we evaluate the quality of the results produced by our HLS flow through the different levels of optimizations. Our objective is to demonstrate the reuse and integration flexibility of our approach and to show how it can be useful in a practical context.

Benchmark Designs. We use four designs listed in Table II for our evaluation; each of these focuses on a specific type of matrix-level optimization. Matrices in these designs were created by a random assignment of 10 independent scalar variables. We did this to limit the number of input arguments to the wrapper function. These designs are used to point out that existing general-purpose HLS tools do not perform higher order, domain-specific optimization and to illustrate how our design approach can avert this limitation.

In the design M1, our matrix-level optimization module reorders multiplications to reduce its implementation cost. For M2, it performs common subexpression extraction to avoid the repeated computation of $B_{4\times4} \cdot C_{4\times4}$. In M3, the module uses the distribute property to extract the common operand, $C_{3\times3}$, from both the terms. M4 is the running example used in Section IV; in this case, the optimization-module performs the transformation shown in Figure 3.

For all designs, our scalar-level optimizations include common subexpressions elimination, like removing the repeated instance of $a+b$ in $\binom{a}{b} + \binom{b}{a}$, and operator strength-reduction, like replacing multiplication by a power-of-two value with shift operation, that are found in many existing HLS tools.

Integration with C-to-RTL Tools. We first consider the case where the matrix elements are fixed-point integers and we integrate our HLS flow to LegUp [7], an open-source C-to-RTL tool, to generate the hardware design. For each design, our flow generates three C-programs: Exp1, where our tool performs no optimizations; Exp2, with only matrix-level optimizations; and Exp3, with both matrix-level and scalar-level optimizations[1]. These C-programs are then compiled using LegUp with its best optimization setting and the generated RTL designs are synthesized using the Altera Quartus II toolset [16] to target a Stratix IV FPGA device. To make it possible to compare across designs, we restricted Quartus II from using any DSP blocks. Table II shows the result from synthesis in terms of number of *Logic Elements* (LEs), number of registers (Reg), circuit latency and maximum frequency (Fmax) for each case. Comparing the results from Exp1 and Exp2 on Table II, it is easy to see that our matrix-level optimizations consistently reduce the resources needed for the implementation without impacting the frequency or latency significantly. These results reveal the potential of high-level optimization and how our methodology can complement existing tools to benefit from them.

The scalar-level optimizations bring further improvement in terms of area, as revealed by the results of Exp3 on Table II. The area saving mostly comes from the common subexpression elimination since LegUp already performs the strength reduction optimizations. This is easily evident in M4 where we get no further area improvement through the scalar-level optimization as the program on Figure 5 offers no opportunity for scalar-level subexpression elimination. There is no consistent trend with Fmax and latency because there are two opposing effects: The smaller design size helps to improve Fmax and to lower the latency. But, since some of the optimizations increase the amount value sharing between different parts of the design, it places more pressure on the routing resources and has an adverse affect on both Fmax and latency. But, overall, the improvement in area is much more significant compared to the variations in timing. These experiments show that performing optimizations at different abstraction levels can progressively improve the design quality. Furthermore, implementing both of our optimization modules in LMS required less than $2,000$ lines of code in total (including all the debugging code, comments and empty lines for formatting). This gives an indication of how little effort is needed to develop custom optimization modules for a new application.

We now consider the case of the matrices with floating-point elements to investigate the impact of such a change in our HLS flow. Since our matrix-level optimizations are agnostic to the element-type, we can reuse them by modifying only the type used for representing matrix elements (`T` in Figure 4). However, we do need to make some modifications to the scalar-level optimization module and its code-generator to support this feature. The total change involved adding/editing less than 50 lines of code, which illustrates the flexibility and the reuse possible in our approach. Table III shows the results obtained using LegUp after this modification and applying all our optimizations. The benefit of the different levels of optimizations follow similar trends as seen in Table II, hence we have not reported them separately.

Integration with IP-Core Generators. If we now want to improve the throughput of our design, we need to pipeline it. Unfortunately, LegUp 3.0 does not support pipelining floating-point computation. While future versions of the tool may remove

[1]Since LegUp can only supports one output port in the hardware it synthesizes, we adapted our C-program generator by adding all the elements in the resultant matrix to produce a scalar value as output. To maintain complete fairness, we do this for every design we use in our study, even when we use other tools.

TABLE II. Results Using LegUp to Generate Fixed-Point Datapaths

#	Matrix Expression	Exp1: C with no opt.				Exp2: C with only matrix-level opt.				Exp3: C with two level opt.			
		# LEs	# Regs	Latency (μs)	Fmax (*MHz*)	# LEs	# Regs	Latency (μs)	Fmax (*MHz*)	# LEs	# Regs	Latency (μs)	Fmax (*MHz*)
M1	$4 \cdot A_{1\times5} \cdot B_{5\times5} \cdot C_{5\times3} \cdot D_{3\times1}$	10,356	2,141	0.11	126	7,167	1,434	0.11	132	5,359	1,226	0.10	127
M2	$A_{4\times4} \cdot B_{4\times4} \cdot C_{4\times4} + B_{4\times4} \cdot C_{4\times4} \cdot D_{4\times4}$	39,541	5,688	0.15	117	35,322	4,825	0.15	117	13,158	3,072	0.17	106
M3	$A_{3\times3} \cdot C_{3\times3} + B_{3\times3} \cdot C_{3\times3}$	3,661	1,494	0.07	181	2,281	729	0.08	159	1,608	155	0.07	140
M4	$2 \cdot det(\begin{smallmatrix}2&1\\3&1\end{smallmatrix}) \cdot X^2_{2\times2} \cdot Y_{2\times1} + X_{2\times2} \cdot Y_{2\times1}$	1,748	463	0.07	131	1,547	309	0.06	148	1,547	309	0.06	154

TABLE III. Results Using LegUp to Generate Single Precision Floating-Point Datapaths With Two Level Opt.

#	Matrix Expression	# LE	# Regs	Latency (μs)	Fmax (*MHz*)	Throughput ($10^6\cdot$ Results/s)
M1	$4 \cdot A_{1\times5} \cdot B_{5\times5} \cdot C_{5\times3} \cdot D_{3\times1}$	10,178	11,806	0.89	235	1.1
M2	$A_{4\times4} \cdot B_{4\times4} \cdot C_{4\times4} + B_{4\times4} \cdot C_{4\times4} \cdot D_{4\times4}$	64,520	89,693	1.22	166	0.8
M3	$A_{3\times3} \cdot C_{3\times3} + B_{3\times3} \cdot C_{3\times3}$	11,281	14,097	0.57	231	1.8
M4	$2 \cdot det(\begin{smallmatrix}2&1\\3&1\end{smallmatrix}) \cdot X^2_{2\times2} \cdot Y_{2\times1} + X_{2\times2} \cdot Y_{2\times1}$	5,295	6,742	0.38	282	2.6

TABLE IV. Results Using FloPoCo to Generate Pipelined Datapaths with Two Level Opt.

#	Matrix Expression	Exp5: FloPoCo-single precision equivalent					Exp6: FloPoCo-custom precision				
		# LE	# Regs	Latency (μs)	Fmax (*MHz*)	Throughput ($10^6\cdot$ Results/s)	# LE	# Regs	Latency (μs)	Fmax (*MHz*)	Throughput ($10^6\cdot$ Results/s)
M1	$4 \cdot A_{1\times5} \cdot B_{5\times5} \cdot C_{5\times3} \cdot D_{3\times1}$	46,967	53,180	0.87	176	176	15,237	27,639	0.50	242	242
M2	$A_{4\times4} \cdot B_{4\times4} \cdot C_{4\times4} + B_{4\times4} \cdot C_{4\times4} \cdot D_{4\times4}$	174,193	181,619	1.36	116	116	59,787	88,504	0.65	188	188
M3	$A_{3\times3} \cdot C_{3\times3} + B_{3\times3} \cdot C_{3\times3}$	31,942	35,860	0.53	186	186	11,245	16,331	0.33	231	231
M4	$2 \cdot det(\begin{smallmatrix}2&1\\3&1\end{smallmatrix}) \cdot X^2_{2\times2} \cdot Y_{2\times1} + X_{2\times2} \cdot Y_{2\times1}$	9,657	12,452	0.32	212	212	2,919	3,830	0.17	312	312

this limitation, it is still an excellent example where selective integration to another tool may be beneficial. To overcome this limitation, we added a VHDL code-generator to the scalar-level optimization module. Additionally, to generate pipelined floating-point operators, we integrated FloPoCo, an open-source arithmetic core generator, into our flow. During code-generation, our tool directs FloPoCo to create the necessary floating-point cores. The code-generator also uses the pipelining information from FloPoCo to generate the glue-logic that connects the individual cores through registers, implementing a correctly pipelined datapath. As shown under Exp5 on Table IV, this design has a much higher throughput because of to the pipelined architecture, but also needs significantly higher area. If the target application only needs floating-point operators of smaller precision, we can modify two configuration parameters in our HLS flow to generate custom cores from FloPoCo that trade-off precision for area. By using floating-point operators that have a 10-bit mantissa and 5-bit exponent, we get the results shown under Exp6 on Table IV.

The results thus illustrate how our approach can be used to perform higher order and domain-specific optimizations that typically go untapped in general purpose HLS tools; our examples are simple and well understood, but the whole approach is generally applicable. We showed that performing optimizations at different abstraction levels can progressively improve the result. We also demonstrated the flexibility and reuse potential of our approach by integrating different tools and reusing our optimization modules to create different output designs in a cost-effective manner.

VI. Related Work

There has been broad interest in HLS tools to make reconfigurable technology more accessible to designers. Consequently, there have been many research efforts, each focusing on different aspects, like target domain, implemented architectures and input languages. Hence, this brief survey only tries to enumerate the some of the main approaches, rather than cover all attempts.

Domain-specific synthesis tools can make hardware synthesis more accessible to users in different domains. These include tools like PARO [17] and MMAlpha [4] which focus on loop transformations to implement highly parallel systems; Spiral [1] which synthesizes hardware for linear transforms in signal processing applications; Optimus [3] which focuses on streaming applications; and HDL coder [2] which synthesizes hardware from Matlab/Simulink systems. They all use a DSL that makes expressing input specifications convenient for their users and help to generate efficient hardware designs. While they have huge potential, the high effort needed to built these tools makes this approach inaccessible to many areas. Our methodology can lower this effort by providing a flexible framework that exhibits modularity and reuse. We have also shown how we can further reduce this effort by integrating with existing tools and by building these tools incrementally.

Although there are sound arguments against using C as an input language for HLS [18], the main-stream interest has been to aid system designers design hardware with a C-like language. These efforts include Vivado High-Level Synthesis [11], Handle-C [19], Catapult [20], ROCCC [21], Gaut [22], LegUp [7] and Trident [13], to name just a few. Among them, ROCCC and Gaut focus on streaming oriented and DSP applications and Trident focuses on scientific computation. Since extracting coarse-grain parallelism from a C program is difficult [23], researchers have also used explicitly parallel C-like languages, like OpenCL [24] and CUDA [25] to design hardware. Notable efforts here include OpenCL-to-FPGA [12], SOpenCL [26] and FCUDA [27]. The problem with using C-like languages is that they are often too low-level and require the user to have detailed knowledge of the optimization potentials of the application and manually perform some optimization [28]. Yet, as we showed using LegUp, our approach can be used along with these tools to develop high-level and more application-oriented tool-chains that are easier to use. By leveraging on the domain-specific knowledge to perform some of the optimizations, our approach can enable these tools to yield excellent results more consistently.

As an alternative to C-like languages, other researchers have proposed to use custom high-level languages, like LIME [29], BlueSpec Verilog [30] and Kiwi [31], for hardware synthesis. Our approach leverages on domain-specific knowledge to automatically perform optimizations and is, therefore, complementary to these efforts. Additionally, our approach can also aid these tools to achieve higher quality results.

In the software domain, the Delite project [15] uses an approach very similar to what we propose. Delite also employs LMS as its backend, but targets portability and high performance for applications running on heterogeneous platforms. To accomplish this, they define a set of custom IR-nodes that are used to capture the parallelism in the applications. To suit the purpose of hardware design, we focus on developing reusable optimization modules and on integrating with external tools to create flexible HLS flows with low development effort.

VII. Conclusions

Domain-specific hardware synthesis tools can make reconfigurable technology more accessible to domain experts who have little hardware design knowledge. In this paper, we have presented an approach to reduce the effort needed to develop such tools. To achieve this, we use LMS as a common platform on which we develop standalone optimization modules that can be easily reused across different HLS flows. To further reduce the design effort, we integrate with external tools, when possible, building the domain-specific HLS tool incrementally. We illustrated this approach with a simple tool to synthesize matrix expressions into hardware and used it to describe the development of the optimization modules. As we showed in the evaluation section, our methodology is flexible to accommodate changes, needs low-development effort, and is able to effectively leverage domain-specific optimizations to produce better designs. As done in the case of LegUp, our approach can also be used to augment existing general-purpose tools with additional domain-specific optimizations, enabling them to produce better results. In conclusion, our approach significantly reduces the cost of building domain-specific hardware synthesis tools, enabling a widespread application of a technology that is able to conciliate high productivity and efficient hardware generation.

References

[1] P. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Computer Generation of Hardware for Linear Digital Signal Processing Transforms," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 17, no. 2, p. 15, 2012.

[2] MathWorks, "HDL Coder," http://www.mathworks.com/products/hdl-coder/, [Accessed: 9-Oct-2013].

[3] A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah, "Optimus: Efficient Realization of Streaming Applications on FPGAs," in *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. ACM, 2008, pp. 41–50.

[4] S. Derrien, S. Rajopadhye, P. Quinton, and T. Risset, "High-Level Synthesis of Loops Using the Polyhedral Model," in *High-level synthesis*. Springer, 2008, pp. 215–230.

[5] "The Scala Programming Language," http://www.scala-lang.org/, [Accessed: 9-Oct-2013].

[6] T. Rompf and M. Odersky, "Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs," *Communications ACM*, vol. 55, no. 6, pp. 121–130, 2012.

[7] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 2011, pp. 33–36.

[8] F. De Dinechin and B. Pasca, "Designing Custom Arithmetic Data Paths with FloPoCo," *Design & Test of Computers, IEEE*, vol. 28, no. 4, pp. 18–27, 2011.

[9] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.

[10] GNU, "GNU C Compiler," http://gcc.gnu.org, [Accessed: 9-Oct-2013].

[11] Xilinx, "Vivado High-Level Synthesis," http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/index.htm, [Accessed: 9-Oct-2013].

[12] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From OpenCL to High-Performance Hardware on FPGAs," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE, 2012, pp. 531–534.

[13] J. L. Tripp, M. B. Gokhale, and K. D. Peterson, "Trident: From High-Level Language to Hardware Circuitry," *Computer*, vol. 40, no. 3, pp. 28–37, 2007.

[14] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky, "Optimizing Data Structures in High-Level Programs: New Directions for Extensible Compilers Based on Staging," in *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2013, pp. 497–510.

[15] K. Brown, A. Sujeeth, H. Lee, T. Rompf, H. Chafi, and K. Olukotun, "A Heterogeneous Parallel Framework for Domain-Specific Languages," in *20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[16] Altera, "Quartus II Handbook Version 13.0," http://www.altera.com/literature/hb/qts/quartusii_handbook.pdf, 2013, [Accessed: 9-Oct-2013].

[17] F. Hannig, H. Ruckdeschel, H. Dutta, and J. Teich, "PARO: Synthesis of Hardware Accelerators for Multi-Dimensional Dataflow-Intensive Applications," in *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2008, pp. 287–293.

[18] S. A. Edwards, "The Challenges of Synthesizing Hardware from C-like Languages," *Design & Test of Computers, IEEE*, vol. 23, no. 5, pp. 375–386, 2006.

[19] Mentor Graphics, "Handle-C," http://www.mentor.com/products/fpga/handel-c/, [Accessed: 9-Oct-2013].

[20] Calypto, "Catapult," http://calypto.com/en/products/catapult/overview, [Accessed: 9-Oct-2013].

[21] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing Modular Hardware Accelerators in C with ROCCC 2.0," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*. IEEE, 2010, pp. 127–134.

[22] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin, "GAUT: A High-Level Synthesis Tool for DSP Applications," in *High-Level Synthesis*. Springer, 2008, pp. 147–169.

[23] 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*, vol. 30, no. 4, pp. 473–491, 2011.

[24] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science & Engineering*, vol. 12, no. 3, p. 66, 2010.

[25] Nvidia, "Nvidia CUDA Programming Guide," 2009.

[26] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos, "Synthesis of Platform Architectures from OpenCL Programs," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*. IEEE, 2011, pp. 186–193.

[27] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. Hwu, "FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs," in *Application Specific Processors, 2009. SASP'09. IEEE 7th Symposium on*. IEEE, 2009, pp. 35–42.

[28] K. Rupnow, Y. Liang, Y. Li, and D. Chen, "A Study of High-Level Synthesis: Promises and Challenges," in *ASIC (ASICON), 2011 IEEE 9th International Conference on*. IEEE, 2011, pp. 1102–1105.

[29] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: a Java-Compatible and Synthesizable Language for Heterogeneous Architectures," *ACM Sigplan Notices*, vol. 45, no. 10, pp. 89–108, 2010.

[30] R. Nikhil, "Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications," in *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*. IEEE, 2004, pp. 69–70.

[31] D. Greaves and S. Singh, "Kiwi: Synthesis of FPGA Circuits from Parallel Programs," in *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on*. IEEE, 2008, pp. 3–12.