This is the author's version of the work. The definitive work was published in *Proceedings of the 25th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 173-177, IEEE, Zurich, Switzerland, Jun. 18-20, 2014.*

Domain-Specific Augmentations for High-Level Synthesis

Moritz Schmid, Alexandru Tanase,
Frank Hannig, and Jürgen Teich
University of Erlangen-Nürnberg
91058 Erlangen, Germany

Vivek Singh Bhadouria and Dibyendu Ghoshal
Department of Electronics and Communication Engineering,
National Institute of Technology,
Agartala, India, 799055

***Abstract*—High-Level Synthesis (HLS) has become a very popular instrument to facilitate rapid development of production-ready implementations for FPGAs. Ever increasing flexibility of the frameworks, however, demands a very high level of domain-specific knowledge from the designer. Examples for such knowledge in window-based image processing are median computation and border handling. Depending on the size of the considered window, writing the code to perform such operations may become overwhelming even at very high abstraction levels. To increase productivity and to make the underlying architecture accessible to non-experts, we propose to combine HLS with domain-specific augmentations. Specifically, we propose a new language extension in form of a reduction for sorting and median computation. Furthermore, we introduce a new high-level transformation to perform multiple kinds of border treatment automatically. Both augmentations may reduce the required amount of code lines considerably. The increase in productivity is analyzed by comparing the lines of code necessary to specify a median filter for HLS in PAULA for synthesis using PARO and in C++ for synthesis using a commercial HLS tool.**

I. Introduction

High-level synthesis (HLS) has become an invaluable tool for the automatic generation of hardware descriptions from behavioral models on a high abstraction level, so that target-specific HLS frameworks are meanwhile offered by many major OEMs [1] [2]. Due to the high flexibility, efficient usage of these tools often requires detailed domain-specific knowledge, ideally about the design target and the application domain. Moreover, HLS aims at elevating the design phase to a very high abstraction level, on which it may also sometimes be difficult to implement specific structures to obtain a certain functionality. A very important application domain for HLS is signal and image processing, in which several important areas of expertise exist. For example, a signal processing expert knows when to use average or median filters, the parallelization expert knows how to efficiently parallelize the desired functionality and the hardware expert knows how to efficiently implement it on the targeted platform. In order to productively develop highly efficient hardware accelerators for this domain, the specific knowledge of all three areas of expertise must be combined. *Domain-specific augmentations* bring together these different areas of design experience and aim at making the efficient implementation of a certain behavior available at the high abstraction level without requiring the designer to know about the specific implementation. Although normal arithmetic operations can be handled efficiently by many HLS frameworks, advanced algorithms may require more sophisticated knowledge about the implementation of the target architecture. In this work, we augment the HLS framework *PARO* [3] with domain-specific reductions and transformations in order to facilitate sorting for median computation and border treatment to efficiently handle image borders. Border treatment is very important for image processing pipelines, since image dimensions must not change and not processing borders may leave visible artifacts, which may become quite large for wide kernels. To demonstrate the effectiveness of the proposed augmentations, we have analyzed the implementation of a median filter using PAULA and a commercial C-based HLS tool. The contributions of this work can be summarized as follows.

- We propose a new high-level transformation to perform several types of border treatment automatically.
- We introduce a new domain-specific language extension in form of a reduction for sorting and median computations.
- We demonstrate the productivity gain of using the proposed augmentations by comparing the lines of code necessaery to implement a median filter.

The rest of this paper is organized as follows. We begin by introducing high-level synthesis and specifically the PARO HLS framework in Section II. Moreover, the section presents the proposed domain-specific augmentations for image processing. Section III presents experimental results. The work is concluded in SectionIV. Related work and background is presented where appropriate.

II. High-Level Synthesis

Recent years have shown rapid growth in the development of tools for the synthesis of hardware implementations from high-level algorithm descriptions for FPGAs. Many of the leading FPGA and IDE vendors offer their own custom solution, such as Xilinx Vivado HLS [1], the Altera SDK for OpenCL [2], and Catapult C by Calypto Design Systems [4], among others. Moreover, HLS is also researched in academia, as for instance, SPARK [5], ROCCC [6], GAUT [7], LegUp [8], and Trident [9]. All aforementioned design tools start from a subset of C, C++, or SystemC code, which already determines the execution order of the program. In contrast to the aforementioned approaches, Bluespec [10] is a design system, based on System Verilog, which targets both compute and control intensive applications. Other examples for generating hardware from custom languages are LIME [11] and Kiwi [12]. In addition, HLS from domain-specific languages aims at making hardware-related aspects of high-level synthesis more accessible to users by providing abstractions related to the

domain. Existing tools include Spiral [13], which is closely oriented on signal processing, Optimus [14] concentrates on streaming applications, or a recently proposed tool by George et al. [15], introducing a DSL for hardware implementations based on Scala. Most of the parallelism contained in the original mathematical model of the algorithm is lost during the transformation to sequential code. Although this enables synthesis of sequential hardware, extensive effort must be applied in order to obtain a parallel implementation. To still remain in the popular C-domain, efforts have been made to target C-like parallel languages, like OpenCL, for example by SOpenCL [16] or Coole et al. in [17], and CUDA in [18]. The here used high-level synthesis framework PARO [3] allows for the formulation of the algorithm in very close relation to the mathematical description using a functional language, called PAULA [19] and uses the polyhedron model [20] to optimize the program for parallel implementation in the form of a processor array.

A. The PARO HLS Framework

The design flow of PARO is depicted in Figure 1. High-level transformations, which can also often be found in modern software compilers, optimize the program to be represented in the polyhedron model. As a second step, polyhedral transformations are used to restructure the program for parallel implementation as a hardware accelerator. Polyhedral transformations, supported in PARO, include, for example, affine transformations, such as loop reversal, loop interchange, and loop skewing, which are popular instruments for the parallelization of algorithms. Moreover, the polyhedral transformations in PARO perform *loop perfectization* and *loop unrolling* to enhance and expose parallelism. Specifically advantageous to signal processing algorithms are transformations to avoid bottlenecks, such as *localization*, and *loop tiling* [21]. In order to preserve clean program code, instructions for transformations and optimizations are collected separately in a script file. After the program has been suitably restructured, PARO performs a so called *space-time mapping* to place and schedule the operations. The space-time mapping assigns each iteration point a processor, referred to as *allocation*, and a point in time to execute the iteration. For hardware implementations, it is not sufficient to only determine a *global schedule* for the iterations. Instead, also a *local schedule* must be determined to specify the start points for the individual statements of the loop iteration. Here, the local schedule must be determined to achieve an ideal overlapping of the iterations (software pipelining) to obtain the required iteration interval (II), that is, the amount of clock cycles between the start of successive iterations. In order to achieve maximum throughput for an accelerator, it is often more important to obtain the minimum II and allow a longer latency for the local schedule. Scheduling is performed in close relation to the assignment of functional operators to the arithmetic operations of the statements, which is referred to as *binding*. Hardware synthesis of the restructured program after allocation, scheduling and binding is the last step, which yields a synthesizable hardware description in a language, such as VHDL. For this, PARO must generate descriptions of the processor elements, the interconnection and control structure, as well as the interfaces. Although the framework supports interfacing RAM structures, limited internal memory of FPGAs leads to choosing streaming interfaces for

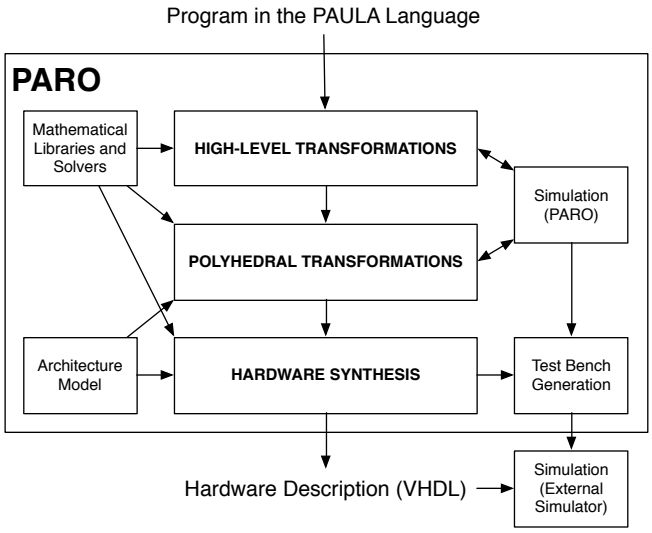
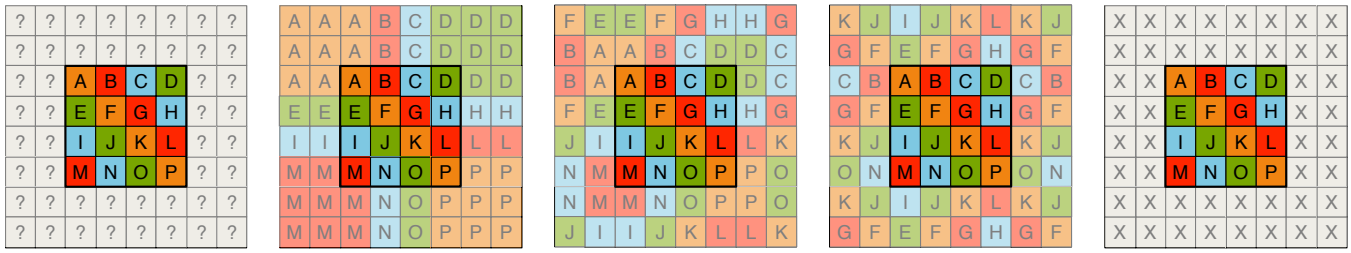


Fig. 1. PARO design flow [3].

image processing. In order to avoid expensive re-reads, PARO automatically determines the life time of internal variables and generates appropriate *input pixel buffers* accordingly. The write semantics of the buffer equals those of a shift register. During the development process, the designer can validate the performed transformations via the built-in simulator, which can also be used to generate a VHDL testbench in order to verify the resulting hardware description in an external simulator against the golden reference model.

B. Image Processing and Border Handling in PAULA

Image processing can be broadly classified into *point*, *global*, and *local operators*. A point operator maps each pixel of the input image to a single pixel in the output image. Typically, point operators are used to perform tasks, such as inversion, brightness and contrast alterations or color scheme transformations. Global operators, in contrast, require the complete image for the computation of the output. Local operators use a local region of pixels in the input image to derive a single pixel of the output image. Examples are window-based image filters. Using *local operators* as in filtering an image with a (constant) kernel is one of the most common tasks. It is a discrete 2D convolution of the image and the kernel, basically a 2D FIR filter and is therefore a linear operation. Typical kernels are, for example, a Gaussian, rectangular, or Hann window for smoothing the image, and Sobel or Laplacian for segmenting images like detecting edges. Despite the obvious benefits of using general purpose languages, such as C/C++ or JAVA for design entry, these languages have the disadvantage that their semantics already enforce a certain order of program execution. Instead, we propose the use of a functional domain-specific language, called PAULA, that is especially well suited for specifying applications with multi-dimensional data-flow, for example in the form of nested loop programs, which usually occur in image processing with a local operator. Our functional language is based on the mathematical foundation of *dynamic piecewise linear/regular algorithms (DPLA)* [19]. When modeling image processing algorithms, designers naturally consider mathematical equations.



(a) Undefined (b) Clamp (c) Mirror (d) Mirror 101 (e) Constant

Fig. 2. Border treatment variants for window-based image processing for windows of radius $r = 2$.

Specifying these in the PAULA language is very intuitive, as reductions that implement mathematical operations, such as *sum* ($\sum$), or *product* ($\prod$) can be used. For illustration, consider a simple averaging filter. We denote images X by a two dimensional array of dimension $(M \times N)$. The address (i, j) defines the spatial location of a certain element of the image, where $i\,(0 \leq i < M)$ and $j\,(0 \leq j < N)$ are the row and column indices, respectively. The picture element $X(i, j)$, often referred to as a *pixel*, represents an intensity value. Applying an averaging filter as a local operator on a window of size $w \times w$ (with $w \mod 2 = 1$, that is, w is odd), can be mathematically described using the radius $r = \lfloor w/2 \rfloor$ of the window as follows

$$\widetilde{X}(i,j) = \frac{1}{(2r+1)^2} \sum_{l=-r}^{r} \sum_{m=-r}^{r} X(i+l, j+m).$$

In the following, we will show that the filter specification in PAULA, is as compact as the mathematical formulation. A natural question that arises when considering window-based algorithms is how to treat the image borders, for which a part of the window lies outside of the image data. One option is to simply ignore the border pixels which will decrease the output image in size by r in each dimension. Such a size reduction is often unwanted, since successive sliding window operations would decrease the image size at each turn. Another option is to perform border treatment and, for example, only process the inner image elements for which the window covers valid image data as illustrated in Figure 2(a) and also shown in the following PAULA code listing.

```
PAR (i >= 0 and i < M and
     j >= 0 and j < N)
{
   x_out[i,j] = SUM[l>=-R and l<=R and
                    m>=-R and m<=R]
                    (x[i+l,j+m])/W*W
     if(i >= R  and i < M-R and
        j >= R and j < N-R);

   x_out[i,j] = x[i,j]
     if(i < R or i >= M-R or
        j < R or j >= N-R);
}
```

Here, the condition proceeding the statements is an iteration dependent condition, which causes the statement to only be scheduled in the specified iterations. Although this type of handling the image borders might be tolerated for small window sizes, using the technique for large kernels would leave large areas of an image unprocessed. Using the techniques shown in Figures 2(b) to (d) mostly give acceptable results, meaning that there is no perceptible degradation towards the borders of the image. Although a slight distortion (depending on the window size) may occur, if there are some diagonal structures. Filling up the missing pixels with a constant value, as shown in Figure 2(e) tends to *pull* the filtered value towards this constant because more and more of the window is now filled with this constant. For example, choosing zero as the constant causes the image to become darker at the borders. Specifying the mentioned border handling strategies can be a lot of code to write, even for small windows. For instance, a window with radius $r = 1$ (3×3 pixels) already must consider nine different cases for every pixel in the window, mirroring border treatment requires 81 additional statements, as there are nine different cases to consider. As a contribution in this work, we introduce a new high-level transformation to easily facilitate *border handling in high-level synthesis* for FPGAs. The transformation supports several possibilities for border handling, which are depicted in Figure 2. By performing border handling using the transformation, the iteration dependent conditions in the above presented example can be removed, which greatly simplifies the code. Internally, the transformation creates a new variable for each window element and assigns iteration dependent conditionals to the variables. Let $\mathbf{W}_X(i, j)$ be a square 3×3 matrix, representing the neighborhood of pixels in a local window centered on a location (i, j) of the image X with radius $r = 1$.

$$\mathbf{W}_X(i,j) = \begin{bmatrix} X(i-1,j-1) & X(i-1,j) & X(i-1,j+1) \\ X(i,j-1) & X(i,j) & X(i,j+1) \\ X(i+1,j-1) & X(i+1,j) & X(i+1,j+1) \end{bmatrix} \quad (1)$$

For the upper left element of $\mathbf{W}_X(i, j)$, $X(i-1, j-1)$, the transformation for mirroring border treatment (refer to Fig. 2(d), using *Mirror 101*) replaces all occurrences of the element with a new variable `x_0` and produces the following PAULA code to specify the border cases.

```
PAR (i >= 0 and i < M and j >= 0 and j < N)
{
  // inner image
  x_0[i,j]=x[i-1,j-1]
    if(i>0 and i<M and j>0 and j<N);

  // upper left corner
  x_0[i,j]=x[i+1,j+1] if(i==0 and j==0);

  // upper border
  x_0[i,j]=x[i+1,j-1] if(i==0 and j>0 and j<N);

  // upper right corner
  x_0[i,j]=x[i+1,j-1] if(i==0 and j==N-1);

  // left border
```

```
  x_0[i,j]=x[i-1,j+1] if(i==0 and j>0 and j<N-1);

  // right border
  x_0[i,j]=x[i-1,j-1] if(i==M-1 and j>0 and j<N-1);

  // lower left corner
  x_0[i,j]=x[i-1,j+1] if(i==M-1 and j==0);

  // lower border
  x_0[i,j]=x[i-1,j-1] if(i==M-1 and j>0 and j<N-1);

  // lower right corner
  x_0[i,j]=x[i-1,j-1] if(i==M-1 and j==N-1);
}
```

As mentioned before, PARO analyzes input data dependencies and creates appropriate input pixel buffers to minimize memory accesses. As the border treatment only reads from the input pixel buffer, it does not require any additional memory and only creates minimum logic overhead.

C. Sorting and Median Computation

In addition to common arithmetic operations, image processing also often requires to determine the median, as for example, when using median filters for reducing salt-and-pepper noise in a corrupted image. To compute the median of a set of values, it is first required to at least partially sort the set. Functional languages provide a rather convenient means of implementing this, for example, a classic solution exists in the single assignment sorting algorithm presented by Rao, et al., in [22]. As reduction operators are a key feature of the PAULA language we augment its capabilities in the image processing domain by including sorting and median computation as the `SORT` and `MED` reduction operators, respectively. The syntax for describing such a reduction operation, which allows for a compact and intuitive description style, is as follows:

```
// sorting
SORT [iteration_space] (expression);

//median computation
MED [iteration_space] (expression);
```

For instance, a noise reduction median filter operating on 3×3 windows can be intuitively written in PAULA.

```
PAR (i >= 0 and i < M and
     j >= 0 and j < N)
{
  y[i,j] = MED[k>=-1 and k<=1 and
               l>=-1 and l<=1]
    (x[i+k,j+l]);
}
```

The first step that PARO must perform for both reduction operators, is to automatically implement a sorting algorithm for the given input space. The two-dimensional sliding-window is linearized and embedded into a four-dimensional data structure, where apart from i and j defining the first two dimensions, the third dimension is used to determine the largest value of the current window and the fourth dimension propagates the minimum between two adjacent values (see Fig. 3). PARO automatically inserts the necessary instructions as well as iteration dependent conditions for sorting the given input space. Note, that although it is necessary to use the full systolic implementation for sorting, the median computation only requires a subset of the necessary operations.

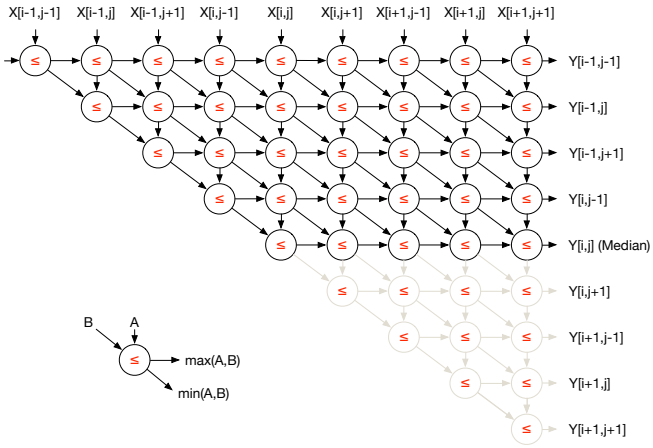


Fig. 3. Overview of the systolic sorting algorithm, a linearized 3×3 window is applied as input on the first row, the sorted window can be observed on the last column. For median computation, the gray lower part of the sorter can be omitted.

III. Experimental Results

We have analyzed the domain-specific augmentations proposed in this work with respect to the gain of productivity and the quality of the generated results (QoR). The gain in productivity can be substantial, especially for border treatment of large windows. To also quantify the quality of the generated result, we have designed a comparable accelerator architecture for streaming data and a median filter using the same algorithm for sorting in C++ and used a commercial HLS tool for synthesis. Both designs were scheduled for an iteration interval II=1, i.e., a new pixel is processed in each clock cycle, and were implemented using Xilinx Vivado 2013.4 for a Xilinx Kintex-7 (XC7K325T-2FFG900C) FPGA. The required lines of code (LoC), type of border treatment (BT), achieved local and global latency (L_l and L_g, respectively), as well as post place and route results, specifying the required amount of look-up tables (LUT), registers (FF), block RAM (BRAM) and the maximum clock frequency (F [MHz]), are listed in Table I. As border treatment and median filter computation are achieved using the proposed domain-specific augmentations, the window size of the median computation can be increased without having to change the actual implementation. In contrast, the C++ implementation requires more LoC, since many parts of the streaming-data based design, such as an appropriate input pixel buffer, the border treatment and the sorting algorithm must be hand-coded. PARO, on the other hand, generates a buffering structure for input pixels automatically. Moreover, the border treatment and the median computation must also be included explicitly in the C++ code. The number of lines of code required for the specification of the median filter in PAULA (LoC_P) and C (LoC_C) indicate that the augmentations may provide a substantial gain in productivity (Prod. = $\text{LoC}_\text{C}/\text{LoC}_\text{P}$). Also, the generated implementations are in an equal range for QoR. Small filter window sizes can be implemented rather efficiently, reaching very high clock frequencies of close to 400 MHz and using only a very small subset of FPGA resources. As the window size increases,

TABLE I. SYNTHESIS AND IMPLEMENTATION RESULTS OF A MEDIAN FILTER FOR 512×512 IMAGES.

		PARO							C++ HLS							
$w \times w$	BT	LoC_P	L_l	L_g	LUT	FF	BRAM	F [MHz]	LoC_C	L_l	L_g	LUT	FF	BRAM	F [MHz]	Prod.
3×3	None	22	13	263181	689	753	2	372.4	159	14	263183	857	962	2	390.3	7.2
3×3	Mirroring	22	13	263181	1116	900	2	256.9	248	14	263183	1391	1159	2	321.4	11.3
5×5	None	22	37	264232	5599	5277	4	318.4	159	30	264226	6843	6409	4	270.1	7.2
5×5	Mirroring	22	37	264232	8573	5550	4	222.5	812	30	264226	7137	6215	4	220.5	36.9
7×7	None	22	73	265297	32427	19733	6	291.9	159	54	265279	44286	29788	6	258.3	7.2
7×7	Mirroring	22	73	265297	34510	20202	6	193.5	2636	54	265279	43076	27834	6	159.2	119.8

however, the filters demand a substantial amount of FPGA resources and the routing congestion significantly deteriorates the achievable clock frequency.

IV. CONCLUSIONS

In this paper, we have presented domain-specific augmentations for high-level synthesis in the image processing domain. We have shown efficient handling of image borders in form of a high-level transformation and we have proposed new reductions to facilitate sorting and median computation. Both concepts are very important for advanced image processing algorithms, however, their implementation requires domain-specific knowledge and usually involves many lines of code. Including these operations as *domain-specific augmentations* in HLS makes efficient implementations accessible on a high abstraction level and may improve productivity significantly. To the best of our knowledge, no other established HLS tool may start with an algorithmic description being so close to the domain of image processing and as compact as mathematical reduction operators. Our evaluation has shown that by using the augmentations, productivity can be increased by lowering the amount of required lines of code significantly, whereas the generated accelerator achieves similar performance and resource requirements as an equal representation synthesized using a commercial HLS tool. Moreover, the proposed augmentations allow the designer to change the window size of local operators without having to adjust the implementation. In comparison, the C-based approach, lacking such augmentations, must be adapted to the window size, which requires the designer to create different implementations to explore the design characteristics.

REFERENCES

[1] Xilinx Inc., "Vivado Design Suite User Guide – HLS," 2013.

[2] Altera Corp., "Altera SDK for OpenCL Programming Guide," 2013.

[3] F. Hannig, H. Ruckdeschel, H. Dutta, and J. Teich, "PARO: Synthesis of hardware accelerators for multi-dimensional dataflow-intensive applications," in *Proceedings of the Fourth International Workshop on Applied Reconfigurable Computing (ARC)*, ser. Lecture Notes in Computer Science (LNCS), vol. 4943. Springer, 2008, pp. 287–293.

[4] Calypto Design Systems Inc., "Calypto Product Family Datasheet," 2012.

[5] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations," in *Proceedings of the 16th International Conference on VLSI Design*, Jan. 2003, pp. 461–466.

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

[7] 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 Netherlands, 2008, pp. 147–169.

[8] 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 (FPGA)*. ACM, 2011, pp. 33–36.

[9] J. Tripp, M. Gokhale, and K. Peterson, "Trident: From high-level language to hardware circuitry," *Computer*, vol. 40, no. 3, pp. 28–37, Mar. 2007.

[10] Arvind and R. Nikhil, "Hands-on introduction to Bluespec System Verilog (BSV)," in *Proceedings of the 6th International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, 2008, pp. 205–206.

[11] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: A Java-compatible and synthesizable language for heterogeneous architectures," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications OOPSLA*. ACM, 2010, pp. 89–108.

[12] S. Singh and D. J. Greaves, "Kiwi: Synthesis of FPGA circuits from parallel programs," in *Proceedings of the 16th International Symposium on Field-Programmable Custom Computing Machines (FCMM)*. IEEE Computer Society, 2008, pp. 3–12.

[13] P. A. 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*, vol. 17, no. 2, 2012.

[14] A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah, "Optimus: Efficient realization of streaming applications on FPGAs," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. ACM, 2008, pp. 41–50.

[15] N. George, D. Novo, T. Rompf, M. Odersky, and P. Ienne, "Making domain-specific hardware synthesis tools cost-efficient," in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, Dec. 2013, pp. 120–127.

[16] M. Owaida, N. Bellas, K. Daloukas, and C. Antonopoulos, "Synthesis of platform architectures from OpenCL programs," in *Proceedings of the IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2011, pp. 186–193.

[17] J. Coole and G. Stitt, "Fast, flexible high-level synthesis from OpenCL using reconfiguration contexts," *Micro, IEEE*, vol. 34, no. 1, pp. 42–53, Jan 2014.

[18] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *Proceedings of the IEEE 7th Symposium on Application Specific Processors (SASP)*, Jul. 2009, pp. 35–42.

[19] F. Hannig, "Scheduling techniques for high-throughput loop accelerators," Dissertation, University of Erlangen-Nuremberg, Germany, 2009, verlag Dr. Hut, Munich, Germany.

[20] P. Feautrier and C. Lengauer, "Polyhedron model," in *Encyclopedia of Parallel Computing*, D. A. Padua, Ed. Springer, 2011, pp. 1581–1592.

[21] J. Xue, *Loop Tiling for Parallelism*. Kluwer, 2000.

[22] S. Rao and T. Kailath, "Regular iterative algorithms and their implementation on processor arrays," *Proceedings of the IEEE*, vol. 76, no. 3, pp. 259–269, 1988.