This is the author's version of the work. The definitive work was published in *Parallel Computing: Accelerating Computational Science and Engineering (CSE), volume 25 of Advances in Parallel Computing, pp. 497-506, IOS Press, 2014.*

High-Level Synthesis Revised:

Generation of FPGA Accelerators from a Domain-Specific Language using the Polyhedron Model

Moritz SCHMID, Frank HANNIG, Alexandru TANASE and Jürgen TEICH
Hardware-Software Co-Design, University of Erlangen-Nürnberg, Germany

Abstract. In this work, we provide an overview of out high-level synthesis framework PARO. PARO is targeted at data flow dominant algorithms where most of the computational load lies in loop nests, defined by affine expressions. In $\mathbb{Z}^n$ these loop definitions can be interpreted as half-spaces, which intersect to form convex polyhedra around the sets of loop iterations. Hence, we employ the polyhedron model to analyze and restructure these algorithms to derive highly parallel and energy efficient implementations on massively parallel architectures. Specifically, in this work, we discuss the implementation of dedicated FPGA accelerators and showcase the capabilities of out framework for the development of a range image conditioning pipeline for smart range sensing cameras.

Keywords. Polyhedron, Polytope, FPGAs, Accelerators

Introduction

Recent years have seen a constantly decreasing feature size allowing ever more complex system on chip architectures. As major OEMs excel in bringing more and more cores and computational power to a wide area of devices, this very welcome development also raises numerous complications as parallel computers and architectures involving multiple cores have become ubiquitous. Challenging problems include the augmentation of reliability issues caused by imperfections in the manufacturing process or hindrance in scalability. Also, power constraints have become so stringent that only a continuously decreasing part of a chip can be actively used at a time to remain within the allotted budget [6]. Instead of introducing more and more computing power, an essential step towards a solution is to decrease the energy per computation in order to achieve the demanded energy efficiency in today's systems. A proven way to enhance the energy footprint of a system is to use multiple computational units, for example, in the form of multi- and many-core systems, as well as to employ heterogeneous systems comprised of domain specific components, such as customized instruction set processors or hardware accelerators for compute intensive tasks.
Next to the constantly increasing productivity in software development, the hardware process is still far behind. To bridge this gap, high-level synthesis (HLS) has constantly been a very popular topic in industry and academia. Despite early failures, the technique has meanwhile matured enough to be able to effectively support design engineers in the development process. Traditional hardware development can be seen as a series of transformations starting from a functional specification, which describes what the algorithm is supposed to do. The next step is decide on *how* the functionality is to be

achieved based on architectural decisions. This behavioral model is then hand coded to create a synthesizable description on the register transfer level. The overall process is very complex, requires sophisticated design knowledge and takes a much longer time than developing a comparable solution in software. Especially the final phase of hand coding is highly error prone. In HLS, RTL is automatically created from the functional specification. The methodology can significantly improve development time, since design implementation and verification can be performed at a very high abstraction level. Just as in the case of software, this exhibits less complexity and requires less detailed knowledge of the underlying hardware. Moreover, HLS enables rapid design space exploration and may be able to perform design optimization at a level which would be very difficult to achieve by hand, even for highly skilled engineers.
A desirable task of HLS is automatic or at least guided parallelization of algorithms. Most of the execution time in scientific applications and digital signal processing is spent in computationally intensive loop kernels, which typically make up only a very small fraction of the code. The parallelism in such loop programs is not contained within a single iteration, but between successive iterations of the kernel. As loop kernels are usually defined using affine expressions, the polyhedron model is ideally suited for their representation. When modeled as a polyhedron, a loop program can be thoroughly analyzed, restructured for efficient parallel implementation and it can be verified that the resulting program is equivalent to its original.
Our high-level synthesis frame work PARO [8], which is the main focus of this work, is capable of automatically generating highly parallel hardware accelerators for a broad variety of multi-dimensional dataflow dominant applications. PARO is based on the polyhedron model and can therefore perform a wide range of polyhedral optimizations to expose an algorithms parallelism and restructure it for implementation on a massively parallel hardware target.
The rest of this work is organized as follows. Work related to the topic at hand is discussed in Section 1. Out high-level synthesis framework PARO is introduced in Section 2. We motivate and explain our method for design entry using a functional language in Section 3. Various high-level and polyhedral transformations that can be applied in PARO are discussed in Sections 4 and 5, respectively. The synthesis of hardware from the program is presented in Section 6. We conclude the work by presenting a case study related to the high-level synthesis of image filters comprising a conditioning pipeline for range imaging.

1. Related Work

Advances in high-level synthesis (HLS) enable automatic generation of hardware accelerators from a description of the algorithm in a high-level language, which can severely reduce the development time [10]. 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, meanwhile offer their own custom solution, such as Xilinx Vivado HLS [18], the Alera SDK for OpenCL [1], and Catapult C by Calypto Design Systems [3], among others. Moreover, HLS is also researched in academia. For instance, the SPARK [7] synthesis methodology is particularly targeted at control-intensive signal processing applications, however, it can only handle one dimensional arrays. ROCCC 2.0 from Jacquard Systems was originally developed at the University of Riverside, California, USA, [16]. It is a SUIF-based compiler for configurable computing, which employs many conventional transformations such as scalar replacement in order to separate computations from memory accesses. In [15], Thielmann

et al. propose *Precore*, to automatically generate data-paths and application-specific logic from high-level C descriptions. All aforementioned design tools start from a subset of C, C++, or SystemC code. However, starting with sequential languages often already determines the execution order of the program. In contrast to the aforementioned approaches, Bluespec [2] is a design system, based on System Verilog, which targets both compute and control intensive applications.
Most of the parallelism contained in the original mathematical model of the algorithm is lost during the transformation to sequential code. The here used high-level synthesis framework PARO allows for the formulation of the algorithm in very close relation to the mathematical description using a functional language, called PAULA [9].

2. High-level Programming Methodology

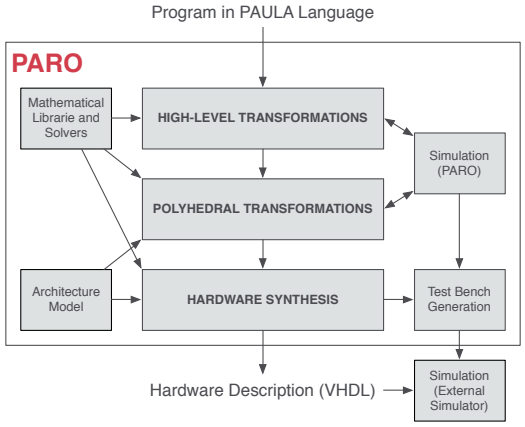

Figure 1. Overview of the PARO framework for high-level synthesis of dedicated FPGA accelerators.

An overview of our high-level programming methodology is depicted in Fig. 1. Its main steps such as the *design entry in form of a domain-specific language*, *polyhedral transformations*, *space-time mapping*, and *hardware-synthesis* are described in the following sections. The proposed approach is based on a high-level synthesis method (PARO) for generating throughput-optimized IP cores in the form of highly parallel dedicated hardware accelerators.

3. Design Entry in Form of a Domain-Specific Language

Despite the obvious benefits of using general purpose programming languages, such as C or Java for design entry, these languages have the disadvantage that their semantics already enforce a certain order of program execution. Most of the parallelism contained in the original mathematical specification of an algorithm is lost when it is specified in sequential code. Consider, for example, the following simple summation $s = \sum_{i=0}^{7} a[i]$. In an imperative language, this is often expressed as a *for* loop:

```
int s = 0;
for (int i=0; i<=7; i++) { s += a[i]; }
```

Instead, we propose the use of a functional domain-specific language (DSL), called *PAULA*, that is especially well tailored for specifying applications with multidimensional data flow, for example, in the form of nested loop programs. The application domain is comprised of digital signal processing such as audio, image, and video processing, as well as algorithms from linear algebra, as for example, matrix-matrix multiplication and LU decomposition, among others. The mapping of such algorithms onto massively parallel architectures, such as FPGAs, requires exact dependence analysis. Imperative languages make this process very complex since they allow variables that are once defined to be overwritten arbitrarily often. To circumvent this problem, modern software compilers, such as gcc [5] or LLVM [12], transform a program into *static single assignment (SSA)* form as an intermediate representation (IR) where each variable is written only once. SSA allows to apply manifold compiler optimizations and transformations in a very efficient way. However, since SSA is only used on the basic block level, these compilers are still unable to solve the data dependence analysis problem for multidimensional arrays. Our functional programming language is based on the mathematical foundation of *dynamic piecewise linear/regular algorithms (DPLA)* [11]. The language consists of a set of recurrence equations defined over a (parametric) polyhedron as multidimensional iteration domain as it occurs in loop nests.

When modeling signal processing algorithms, a designer naturally considers mathematical equations. Specifying these in the PAULA is very intuitive. To allow irregularities in a program, an equation may have iteration and run-time dependent conditions. Furthermore, reductions that implement mathematical operators such as $\sum$ or $\prod$ can be used. For illustration, denote a two-dimensional Gaussian filter applied to an image. A popular approximation for hardware implementation is the binomial filter which reduces the computation to convolving the window with a discretized coefficient matrix. For instance, smoothing an image of size 1920×1080, using a discretized 3×3 windowing filter, is defined by

$$w = \begin{pmatrix} w_{-1,-1} & w_{0,-1} & w_{1,-1} \\ w_{-1,0} & w_{0,0} & w_{1,0} \\ w_{-1,1} & w_{0,1} & w_{1,1} \end{pmatrix} = \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} \tag{1}$$

$$img^{out}_{x,y} = \frac{1}{16} \sum_{i=-1}^{1} \sum_{j=-1}^{1} img^{in}_{x+i,y+j} \cdot w_{i,j} \qquad \forall\, 2 \le x \le 1919 \;\wedge\; 2 \le y \le 1079 \tag{2}$$

In our DSL PAULA, the same filter can be specified as follows:

```
... // variable declarations are omitted
w[-1,-1] = 1; w[ 0,-1] = 2; w[ 1,-1] = 1;
w[-1, 0] = 2; w[ 0, 0] = 4; w[ 1, 0] = 2;
w[-1, 1] = 1; w[ 0, 1] = 2; w[ 1, 1] = 1;

par (x >= 2 and x <= 1919 and y >= 2 and y <= 1079)
{ h[x,y] = SUM[i>=-1 and i<=1 and j>=-1 and j<=1]
              (img_in[x+i,y+j] * w[i,j]);
  img_out[x,y] = h[x,y] >> 4;  // divided by 16
}
```

The code shown above is very similar to its mathematical representation in Eq. (2). As PAULA is a functional language, the order of appearance of statements in a program is not of importance and can thus be arbitrarily interchanged. This implies also the strict

SSA semantics of our DSL, i. e., each instance of an indexed variable is only defined once. Moreover, the bounded polyhedron, or polytope, defined by the `par` statement does not impose any execution order on the subsequent loop body, it simply defines at which iteration points the code in the closure has to be executed. Even for loop-carried data dependences, the programmer neither needs to worry about a valid execution order nor how to parallelize the code. The decision to design our DSL as an external one has the advantage that the underlying semantic model is also tailored for expressing nested loop program specifications. Instead of using an abstract syntax tree (AST) as IR in the compiler, we use a graph-based IR that clearly separates data from static control flow. We call this IR the *reduced dependence graph* (RDG). The iteration variables are generally increased or decreased by a constant value (regularity of the iteration domain). Each loop bound defines a half space and the intersection of all half spaces describes a polyhedron. Loop parallelization in the polyhedron model is a powerful technique [4], therefore in the following, the iteration domains[1] are formulated as polyhedra. An *iteration domain* $\mathcal{I}$ denotes the set of integral points within a convex polyhedron in $\mathbb{Z}^n$. It is defined by $\mathcal{I} = \{I \in \mathbb{Z}^n \mid AI \geq b\}$ where $A \in \mathbb{Z}^{m \times n}$ and $b \in \mathbb{Z}^m$.

4. High-Level Transformations

Based on a given algorithm in form of our DSL notation, various high-level transformations and optimizations can be applied within the design system (see Fig. 1). Among others, these transformations include:

Constant and variable propagation. The propagation of variables and constants leads to a more compact code and decreased register usage.

Common sub-expression elimination. By data flow analysis, identical expressions within a program can be identified. Subsequently, it can be analyzed if it is worthwhile to replace an expression with an intermediate variable to store the computed value.

Dead-code elimination. By static program analysis, program code can be determined that does not affect the program at all. This code, called *dead code*, can either be code that is unreachable or it affects variables that are neither defined as output variables nor used somewhere else in the program. Dead code might result from other transformations such as *common sub-expression elimination*.

Strength reduction of operators. Strength reduction is a compiler transformation that systematically replaces operations by less expensive ones. For instance, in our compiler, multiplications and divisions by constant values can be replaced by shift and add operations.

5. Polyhedral Transformations

Transformations on programs represented in the polyhedron model serve to restructure the program so that it exhibits certain properties desirable for the implementation. After the program has been restructured, the iterations of the program must be scheduled, i.e., associated with a logical date and allocated to a suitable functional resource. As these two

[1] It should be noted that we consider in our design flow an extended definition of an iteration domain, which can handle also lattices in order to reflect non unit loop strides. However, for illustration purposes throughout this paper, Definition **??** is sufficient.

points are closely related, we refer to this procedure as the space-time mapping in our high-level synthesis framework.

5.1. Transformations for Program Optimization

Program optimization transformations aim at improving certain characteristics of the represented loop program. Polyhedral program transfromations supported in PARO are listed in the following.

Affine transformations. Affine transformations of the polyhedron are a popular instrument for the parallelization of algorithms. Transformations such as *loop reversal*, *loop interchange*, and *loop skewing* can be expressed by affine transformations [17]. In addition, affine transformations can be used to embed variables of lower dimension into a common iteration space.

Loop perfectization. Loop perfectization transforms non-perfectly nested loop programs into perfectly nested loops [19].

Loop unrolling. Loop unrolling is a major optimization transformation to expose parallelism in a loop program. Loop unrolling expands the loop kernel by a factor of n by copying $n-1$ consecutive iterations, which leads to larger data flow graphs at the benefit of possibly more instruction level parallelism.

Whereas the aforementioned transformations are well established and widely used in production compilers for single core or shared-memory systems, there exist some transformations, such as *localization* and *partitioning*, that are specifically advantageous for signal processing on FPGAs.

Localization. Localization systematically replaces affine dependencies by regular dependencies in order to increase the regularity of communication and to avoid bottlenecks ([14]).

Partitioning. Partitioning is a well known transformation, which covers the iteration domain of a computation using congruent tiles as for example, hyperplanes, hyperquaders, or parallelotopes. Partitioning is often applied in order to match an algorithm to given architectural constraints. Common terms for partitioning in literature [17,20] are *loop tiling*, *blocking*, or *strip mining*. Tiling, also known as LPGS (local parallel global sequential) partitioning as a loop transformation increases the depth of the loop nest from an n-deep nest to a $2n$-deep nest, where n represents the number of dimensions. Clustering and blocking, also known as LPGS (local parallel global sequential) is a key technique for FPGAs to be able to cope with large image diemension. With this technique, the image is split up into blocks, which are processed in parallel.

5.2. Space-Time Mapping

A very important polyhedral transformation in the design flow is the space-time mapping. It involves placing and scheduling the executions. Scheduling maps each instance in the iteration domain to a logical point in time. The number of instances to schedule is typically large if not infinite in the case of polyhedra. Therefore the schedule must be a closed-form function of the iteration vector. The space-time mapping assigns each iteration point $I \in \mathcal{I}$ a processor index $P \in \mathcal{P}$, referred to as allocation and a time index $t \in \mathcal{T}$, also known as scheduling, defined by the following affine polyhedral transformation:

$$\begin{pmatrix} p \\ t \end{pmatrix} = \begin{pmatrix} \mathbf{Q} \\ \boldsymbol{\lambda} \end{pmatrix} \cdot I + \begin{pmatrix} \mathbf{q} \\ \gamma \end{pmatrix}$$

$\mathcal{T} \in \mathbb{Z}$ is called the time space, that is the set of all times steps where an execution takes place. $\mathcal{P} \in \mathbb{Z}$ is called the processor space, whereas $\mathbf{Q} \in \mathbb{Z}^{s \times n}$ is called the allocation matrix, which determines which processor carries out an execution of a statement. λ is the schedule vector and provides a start time for each iteration point I.
The assignment of a start time to each iteration is not sufficient for hardware implementation. Instead, the start time of each statement in the loop body must be determined. Therefore, we extend the time mapping in order to inlcude the offset for computation of each left hand side variable in the loop body and each node in the reduced dependence graph. Denote offset $\tau(v_i)$ for node v_i. The start time of v_i at iteration point I is: $t(v_i(I)) = \lambda \cdot I + \gamma + \tau(v_i)$.
The purpose of scheduling is to determine a global schedule for the iterations (λ) and a local schedule for the execution of the statements of the loop body (τ). The assignment of a functional unit to each node of the RDG for execution is referred to as binding. Binding in PARO is very versatile, as it is possible to bind operations to virtually any type of functional resource. For instance, operations on floating point data can be bound to an IP core optimized for the target architecture or to a target independent implementation. Furthermore it is possible to integrate data-flow optimized super-expressions, as they recently became available in the FloPoCo framework. Necessary information to integrate functional units in PARO is the latency of the operator as well as the supported pipelinerate and the specific interface. After binding, each node is annotated with an execution time, denoted by $w(v_i)$. The problem of resource constrained scheduling and binding can be solved by mixed integer linear programming.

6. Hardware Synthesis

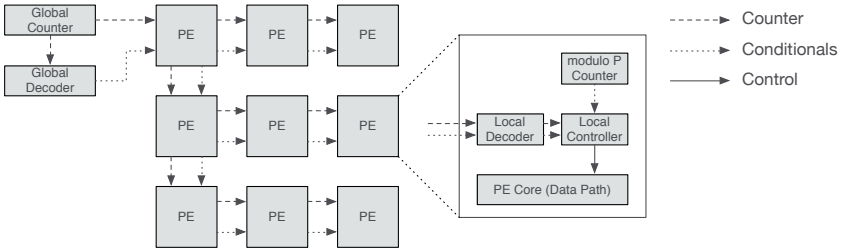

Figure 2. Structural representation of the architecture create during hardware synthesis.

The transformed program is synthesized in the form of a processor array, as illustrated in Figure 2. Hereby the processor is the a platform and language independent intermediate representation, which can be transformed into a synthesizable hardware description in a language, such as VHDL. The synthesis as a processor array consists of the synthesis of the processor elements, the array interconnect structure as well as the control path. Furthermore, hardware synthesis is responsible for deriving a suitable interface for the IP core.

6.1. Synthesis of the Processor Elements

A processor element (PE) consists of the processor core and the local controller. The synthesis of a PE requires a scheduled reduced dependence graph. The core performs the

actual operations of the loop body, which were assigned to functional units during binding. Hence, the synthesis of the PEs requires the instantiation the assigned functional units in the core. In case of operator sharing, appropriate multiplexers must be generated and instantiated, as well.

6.2. Synthesis of the Interconnection Structure

The interconnect structure of the processor array can be derived directly form the RDG. It requires the analysis of dependences and must use scheduling and placement information. In the following, consider an input specification $x_i[I] = f(\ldots, x_j[I + d_{ji}], \ldots)$ with space-time mapping as in Equation 5.2. The synthesis of processor interconnections for data communication is obtained by determining data dependences and a possible processor displacement, meaning that data produced by a certain processor is used by another processor and must therefore be transported. Determining the timing displacement between operations placed on different processors yields the amount of registers necessary to store produced data before it is consumed. The timing displacement is obtained as: $d^l_{ji} = \lambda \cdot (I + d_{ji}) + \tau(x_j) + w(x_j) - \lambda \cdot I - \tau(x_i) = \lambda \cdot d_{ji} + \tau(x_j) + w(x_j) - \tau(x_i)$ In order to build an efficient infrastructure, the synthesis must be able to generate appropriate delay structures, such as shift registers for short delays, as well as FIFOs to replace only sparsely filled registers.

6.3. Synthesis of the Control Structure

Regular processor arrays require a control structure to orchestrate the computations. To allow for scalability, the size of the control path must be independent from the problem size and the size of the processor array. Key characteristics of our synthesized control structure is the use of combined a global and local control facilities. Control signals common to all processor elements are generated by the global controller and propagated through the array. Local controllers are only necessary for signals that differ among the processor types. The combined use can significantly reduce the control structure in size and may allow for higher clock frequencies. The central component of the global controller is the global counter which generates the non-constant parts of the iteration vector. The is taken to compute iteration independent conditionals

6.4. Interface Synthesis

The duty of the interface is to provide data and configuration exchange between the accelerator and external logic. High throughput data exchange can be achieved by using ready-valid data channels, as defined by the AXI4 bus standard for streaming data communication. Ready-valid-interfaces are related to acknowledging write and read operations. In contrast, the handshake signals can be continuously asserted to indicate that the IP core is able to consume or produce data, which exhibits more information than the acknowledge interface. Furthermore, it enables easy chaining of accelerators without requiring extra control logic. As polyhedral program transformations can produce various streaming patterns, the interface must be included in the synthesis process to generate appropriate control logic.

7. Case Study: Range Image Preprocessing

A relatively new field for the application of dedicated hardware accelerators are smart cameras involving range image sensors. The sensors provide the depth information of

a scene, however, the measured values are subject to noise which causes bumps and distorted surfaces in the range data. The data can be conditioned by applying a sequence of digital filters. Temporal noise can be reduced through temporal averaging on a sequence of images. Typically, the sensors are highly sensitive to reflective and translucent surfaces. The corrupted data can be repaired through defect pixel interpolation. Gaussian noise in the images can be reduced by edge-preserving noise filtering. We have created dedicated hardware accelerators to implement the individual steps of the conditioning pipeline and implemented these for PCI express based prototyping design on a Xilinx Virtex-6 FPGA [13]. The filters use single precision floating point arithmetic for the calculations. Since the pipeline is targeted to a Xilinx FPGA, we have included a library of floating point functional units optimized for the Virtex-6 from Xilinx. Post place and route (PPNR) resource requirements for the implementation of the prototyping system including host-to-FPGA communication via PCI express are presented in Table 1.

PPNR Resource Requirements							
radius	ii	reg	lut	bram	dsp	fps	lat[ms]
2 Frame Bilateral Temporal Averaging							
1	1	77,560	49,233	96	136	488	2.049
2	2	92,455,	59,591	109	181	244	4.098
3	2	133,801	94,281	113	349	244	4.098
4 Frame Bilateral Temporal Averaging							
1	1	81,125	50,823	96	156	488	2.049
2	2	94,517	60,590	109	191	244	4.098
3	2	135,890	96,625	113	359	244	4.098
8 Frame Bilateral Temporal Averaging							
1	2	67,686	41,982	102	137	244	4.098
2	2	99,061	63,377	111	211	244	4.098
3	2	140,496	100,298	115	379	244	4.098

Table 1. Resource requirements for various combinations of filter modules after synthesis, placement and routing. Window-based filters can be synthesized for different kernel sizes (*radius*) and iteration intervals (*ii*). The temporal averaging can be varied in terms of how many consecutive frames are used for the computation. FPGA resources are given in registers (*reg*), lookup tables (*lut*), block RAMs (*bram*) and DSP slices (*dsp*).

The designs were synthesized for image dimensions of the Microsoft Kinect range sensor, yielding 640×480 pixels per frame at a clock frequency of 150 MHz. The iteration interval (*ii*) specifies how many clock cycles it takes to start a new iteration. As the conditioning is only a preliminary step before more advanced processing, such as feature extraction, it is important that as little time as possible is spent for processing the images. Filter pipelines as this are an ideal target for high-level synthesis, since the methodology is not only applicable for range images obtained from structured light, but also for other range sensing modalities, such as time of flight.

8. Conclusions

In this work we have provided an overview of our high-level synthesis framework PARO. Its main target are data flow dominant applications domains where heavy computational load stems from loop nests defined by affine expressions. The polyhedron model is highly suitable to analyze and restructure these kind of algorithms to achieve highly optimized implementations on various massively parallel targets, such as dedicated accelerators on FPGAs. We have demonstrated PAROs capabilities in a case study to develop a

conditioning pipeline for range image processing, which can be implemented on an FPGA in conjunction with a range sensor as a smart camera.

References

[1] Altera Corp. Altera SDK for OpenCL programming guide, 2013.

[2] Arvind and R. Nikhil. Hands-on introduction to Bluespec System Verilog (BSV). In *Formal Methods and Models for Co-Design, 2008. MEMOCODE 2008. 6th ACM/IEEE International Conference on*, pages 205–206, 2008.

[3] Calypto Design Systems Inc. Calypto product family datasheet, 2012.

[4] P. Feautrier and C. Lengauer. Polyhedron Model. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1581–1592. Springer, 2011.

[5] GCC, the GNU Compiler Collection, 2013.

[6] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The GreenDroid mobile application processor: An architecture for silicon's dark future. *IEEE Micro*, 31(2):86–95, March–April 2011.

[7] 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*, pages 461–466, Jan. 2003.

[8] 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)*, volume 4943 of *Lecture Notes in Computer Science (LNCS)*, pages 287–293, London, United Kingdom, 2008. Springer.

[9] F. Hannig, H. Ruckdeschel, and J. Teich. The PAULA language for designing multi-dimensional dataflow-intensive applications. In *Proceedings of the GI/ITG/GMM-Workshop – Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 129–138. Shaker, 2008.

[10] F. Hannig, M. Schmid, J. Teich, and H. Hornegger. A deeply pipelined and parallel architecture for denoising medical images. In *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT)*, pages 485–490, Beijing, China, 2010. IEEE.

[11] F. Hannig and J. Teich. Resource constrained and speculative scheduling of an algorithm class with run-time dependent conditionals. In *Proceedings of the 15th IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP)*, pages 17–27, Galveston, TX, USA, 2004. IEEE Computer Society.

[12] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, Palo Alto, CA, USA, Mar. 2004.

[13] M. Schmid, M. Blocherer, F. Hannig, and Teich. Real-time range image preprocessing on FPGAs. In *Proceedings of the 2013 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2013*. IEEE, Dec. 2013.

[14] L. Thiele and V. Roychowdhury. Systematic Design of Local Processor Arrays for Numerical Algorithms. In E. Deprettere and A. van der Veen, editors, *Proceedings of the International Workshop on Algorithms and Parallel VLSI Architectures*, volume A: Tutorials, pages 329–339, Amsterdam, 1991. Elsevier.

[15] B. Thielmann, J. Huthmann, and A. Koch. Precore – a token-based speculation architecture for high-level language to hardware compilation. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 123–129, 2011.

[16] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing modular hardware accelerators in C with ROCCC 2.0. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:127–134, 2010.

[17] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[18] Xilinx Inc. Vivado design suite user guide – high-level synthesis, 2013.

[19] J. Xue. Unimodular transformations of non-perfectly nested loops. *Parallel Computing*, 22(12):1621–1645, 1997.

[20] J. Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.