Noname manuscript No.
(will be inserted by the editor)

Type-Driven Automated Program Transformations and Cost Modelling for Optimising Streaming Programs on FPGAs

Wim Vanderbauwhede · Syed Waqar Nabi · Cristian Urlea

Received: date / Accepted: date

Abstract In this paper we present a novel approach to program optimisation based on compiler-based type-driven program transformations and a fast and accurate cost/performance model for the target architecture. We target streaming programs for the problem domain of scientific computing, such as numerical weather prediction. We present our theoretical framework for type-driven program transformation, our target high-level language and intermediate representation languages and the cost model and demonstrate the effectiveness of our approach by comparison with a commercial toolchain.

Keywords FPGA · type-driven program transformations · analytical cost models

1 Introduction

1.1 The promise of FPGAs in HPC

The promise of energy efficiency combined with higher logic capacity and maturing High-level Synthesis (HLS) tools are pushing FPGAs into the mainstream of heterogeneous High-Performance Computing (HPC) and Big Data. FPGAs allow configuration to a custom design at fine granularity. The advantage of being able to customize the circuit for the application comes with the challenge of finding and programming the best possible implementation.

HLS tools such as Maxeler [14], Altera-OpenCL [5] and Xilinx SDAccel [19] have raised the abstraction of design entry considerably. However, even

Wim Vanderbauwhede
School of Computing Science
University of Glasgow
Glasgow, UK
E-mail: wim.vanderbauwhede@glasgow.ac.uk, syed.nabi@glasgow.ac.uk, c.urlea.1@research.gla.ac.uk

with such tools, parallel programmers with highly specialised expertise are still needed to fine-tune the application for performance and efficiency on the target FPGA device.

We contend that the design flow for HPC needs to evolve beyond current HLS approaches to address this productivity gap. Our proposition is that the design entry should be at a higher abstraction, preferably that of the legacy CPU code, and the task of generating architecture-specific parallel code should be done by the compilers. Such a design-entry point will be truly performance-portable, and accessible to programmers who do *not* have FPGA and parallel programming expertise.

1.2 The optimal FPGA programming model

The FPGA implements a program as a synchronously clocked logic circuit. The resources used by the program are therefore limited by the amount of space available on the chip for each type of resource (logic gates, flip-flops, memory cells, routing). Furthermore, FPGAs excel at operation-level parallelism, i.e. the optimal programming model is a deep pipeline working on a stream of data (as opposed to data parallelism in e.g. a GPU). This is a consequence of the relatively low clock speed of the FPGA. The implication is that for use in HPC, where high throughput is key, the target programs to be deployed on FPGAs are ideally static in terms of memory allocation and are data flow dominated (as opposed to control flow dominated). With that assumption, the optimal deployment of a program onto an FPGA translates to the optimal spatial layout of the data flow pipeline, and determining this optimal layout is our aim.

1.3 The TyTra project: heterogeneous programming and HPC

This work is part of the EPSRC *TyTra* project[1], which aims to address the wider challenge of programming heterogeneous HPC platforms through a type-driven program transformation approach. However, our particular focus in this paper is on the compiler-based program transformations required to optimise performance of scientific, array-based programs on FPGAs. Our proposed approach is inspired by functional languages with expressive type systems such as Haskell[2] and Idris[3] and based on *type-driven program transformations* and *analytical cost models*, both implemented in the *TyTra* compilation toolchain.

The ultimate aim of our work is to compile HPC applications such as climate and weather simulators, written typically in Fortran, to FPGAs.

The focus of this paper is the language and compilation framework for type-driven program transformation. This framework allows us to create very large

[1] http://www.tytra.org.uk
[2] http://www.haskell.org
[3] http://www.idris-lang.org/

numbers of variants of a given program with provably identical functionality but different performance and resource utilisation. We combine this program transformation framework with a fast and accurate analytical cost/performance model for the target FPGA architecture to obtain the performance and resource utilisation for each variant. In this way we can obtain the optimal program variant by exploring the program search space. Because of the size of the search space, fast cost/performance calculation is essential. Using a normal commercial toolchain for FPGA programming, it takes several minutes to obtain a performance estimate and several hours to obtain the final resource cost for a single variant.

1.4 Contributions of this paper

We show in this paper that the parallelisation of the dataflow graph (netlist) of the FPGA implementation of a program can be expressed through the *types* of functions in a functional program, while the operations at the nodes of the graph are expressed through the *definition* of the functions. We show that there is an equivalence between our proposed high-level functional coordination language and the dataflow graph on the FPGA, so that we can reshape the dataflow graph in terms of the parallelism that it exposes by transforming the program. Furthermore, crucially, we show that the transformation of the program is automatically derived from the type transformation.

2 Related Work

Others have also observed the need for a higher abstraction design entry. For example, researchers have proposed algorithmic skeletons to separate algorithm from architecture-specific parallel programming[4], and a thread pool abstraction framework for accelerator programming [10]. SparkCL[15] adds support for increasingly diverse architectures, including FPGAs, into the familiar Apache Spark framework through integration with OpenCL.

In the context of high-level programming specifically for FPGAs, there is considerable prior work, including compiler optimizations and design-space exploration. Such approaches raise the abstraction of the design-entry from HDL to typically a C-type language, and apply various optimizations to generate an HDL solution. Our observation is that most solutions have one or more of these limitations that distinguish our work from them: (1) design entry is in a *custom* high-level language that nevertheless is not a pure software language (even if it looks superficially like C) and requires knowledge of target hardware and the programming framework([14,5,8]), (2) compiler optimizations are limited to improving the overall architecture already specified by the programmer, with no real *architectural* exploration([14,5,8,1]), (3) solutions are based on a creating soft microprocessors on the FPGA and not optimized for HPC([1,9]), (4) the exploration requires evaluation of variants that take a prohibitively

long amount of time[8], or (5) the flow is limited to very specific application domain e.g. for image-processing or DSP applications[7]. The *Geometry of Synthesis* project[17] is provides design entry in a functional paradigm and generation of RTL code for FPGAs, but does not include automatic generation and evaluation of architectural design variants as envisioned in our project. The work described in [16] on extending the roof-line analysis for FPGAs is quite relevant – we are this analysis as a representation our cost-model – but the parallels do not extend beyond this aspect.

3 Expressing Spatial Layouts through Types

3.1 Netlists

At the lowest level of abstraction using in digital circuit design, a program is a directed graph (called *netlist*) where every edge is a wire and every node a logic gate, a memory element (e.g. flip-flop, SRAM cell) or an I/O pin. In practice, netlists are hierarchical. For our purpose, the building blocks of the netlists correspond to functions in the original program. More specifically, each node in our netlists processes finite streams (vectors). Simplifying, each node corresponds to a nested loop in the original program, working on a static array.

3.2 Transforming netlist nodes through type transformations

Our aim is to transform the nodes spatially to obtain higher throughput for the overall program. Essentially, this means replicating nodes to achieve a degree of data parallelism. As we discuss in more detail in Section 4, the nodes in our programs process vectors by application of *map* or *fold* on pure functions. We will discuss the limitations of this assumption in Section 8.1.

However, our fundamental observation is that, because of this assumption on the nature of the nodes, we only need to know their types in order to perform the parallelism transformations: the type is the communication interface of the node, and the parallelization is only affected by the communication, not by the actual computation performed by the node. Similarly, we do not need to know the actual data that is processed by the odes, only its type. In the rest of this Section we formalise this concept.

3.3 Type Variables and Sizes

a is an *atomic* type variable, i.e. representing a nullary type constructor. In practice, this means a is not a vector.
b,c are general type variables, not necessarily atomic, i.e. they can be vectors.
k,l,m,n are *sizes*, so $k, m, n \in \mathbb{N}_{>0}$

3.4 Vector types

- Let $\text{Vec}\, k\, b$ be a *vector type*, i.e. it represents a vector of length k containing values of type b as a simple dependent type (similar to the vector type in Idris).
- The *total size* of a nested vector type is defined as the product of all sizes:

$$\mathcal{N}(\text{Vec}\, n_1\, \text{Vec}\, n_2 ... \text{Vec} ... \text{Vec}\, n_k\ b) \triangleq \prod_{i=1}^{k} n_i \triangleq n$$

- Given an atomic type a, we can generate the set of all vector types $V(a, n)$ for a with total size n:

$$\begin{cases} a \notin V(a,n) \\ \forall\, b\ \in\ V(a,n), \forall\, k\ \in\ [0,n]\,|\,\text{Vec}\, k\, b \in V(a,n) \end{cases}$$

  In other words, this set V(a,n) is formed of all possible types resulting from reshaping a vector $\text{Vec}\, n\, b$ through nesting.

3.4.1 Transformations on Vector Types

We now introduce transformations of vector types that will allow us to create any element of V(a,n).

Functions operating on types start with an uppercase letter, e.g. F, G. They are general, right-associative functions operating on a single type. So we can write e.g. $G\ F\ b$.

We posit two fundamental transformations on vector types, Split and Merge. These correspond to the familiar transformation used in the algorithmic skeleton literature [6,3,4], but operate on on *types* rather than on values.

Split: converting a 1-D vector type to 2-D vector type

$$\text{Split}\, k\, \text{Vec}\, (m.k)\, b\ \triangleq\ \text{Vec}\, k\, \text{Vec}\, m\, b$$

Merge: converting a 2-D vector type to 1-D vector type

$$\text{Merge}\, \text{Vec}\, k\, \text{Vec}\, m\, b\ \triangleq\ \text{Vec}\, (k.m)\, b$$

Note that a transformation on vector types does not have any effect on *atomic types*:

$$\text{Split}\, k\, \text{Vec}\, a = \text{Merge}\ \text{Vec}\, a = a$$

It is easy to show that $V(a, n)$ is closed under *Split* and *Merge*. Furthermore, it should be obvious that application of *Split* and *Merge* preserves the *total size* of a vector type.

Essentially, what these operations describe is a mechanism to partition vectors so that the total size and ordering are conserved.

For convenience, we introduce a shorted notation for a multi-dimensional vector

$$\text{Vec}\, n_1\, \text{Vec}\, n_2\, \text{Vec}\, n_3\, ...\, \text{Vec}\, n_k\, b\ =\ \text{NDVec}\, [n_1..n_k]\, b$$

3.4.2 Graph interpretation of Split *and* Merge

- The *Split* reshaping operation splits a vector into multiple vectors, which corresponds to a demultiplexer
- The *Merge* reshaping operation combines multiple vectors into one, which corresponds to a multiplexer
- As shown if Figure 1, the process of demultiplexing requires replication of the node. We will explain in Section 4.1 how this replication can be derived from the transformations.

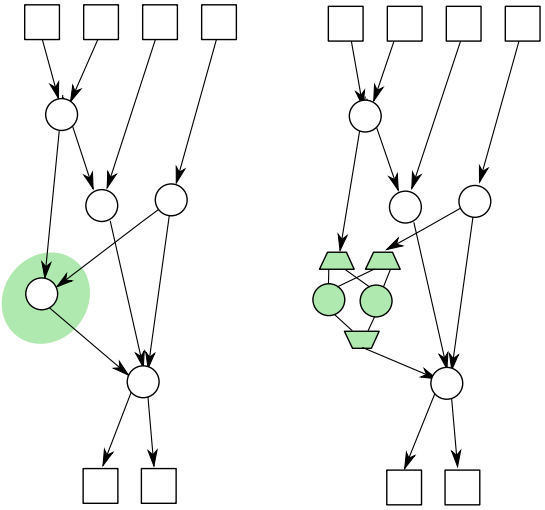

Fig. 1 Parallelisation of a node in the computational graph

We should note that the interpretation of *split* and *merge* in this way is just one possible interpretation of the effect of splitting and merging vectors; it is however the most common one. We will see in Section that the actual parallelism depends on the chosen implementation.

3.5 Tuple types

Let $Tup\, b_1\, b_2 \ldots b_m \stackrel{not.}{=} (b_1, b_2, ..., b_m)$ be a *tuple type*, i.e. it represents a record containing values of different types.

3.5.1 Vector type transformations and tuple types

By definition, applying a vector type transformation F to a tuple of vectors results in application to every element in the tuple:

$$F\,(\mathrm{Vec}\,k_1\,a_1, ..., \mathrm{Vec}\,k_m\,a_m) \triangleq (F\,\mathrm{Vec}\,k_1\,a_1, ..., F\,\mathrm{Vec}\,k_m\,a_m)$$

3.5.2 Graph interpretation of tuples

Tuple types are used to describe computations with multiple return values. We will further define functions to represent fan-in to a node (*zipping*) and fan-out from a node (*unzipping*).

3.6 Function types

Function types are types of the form $b_1 \to b_2 \to ... \to b_m$ representing the type of all arguments and the return value. As in Haskell the arrow is right-associative, and partial application is possible.

By definition, applying a vector type transformation F to a function type results in application to every vector type. For example, applying a type transformation F to the type of the *map* function gives:

$$F\,(a_1 \to a_2) \to \mathrm{Vec}\,k\,a_1 \to \mathrm{Vec}\,k\,a_2 \triangleq (a_1 \to a_2) \to F\,\mathrm{Vec}\,k\,a_1 \to F\,\mathrm{Vec}\,k\,a_2$$

4 TyTra-CL: A functional coordination language for streaming programming

With the above definitions of vector types and their transformations, we can now show how the actual program transformation can be derived from the type transformations.

To describe the program netlists, we will use TyTra-CL[4] "Coordination Language", a very simple statically typed functional language with dependent types. It is in fact a subset of Haskell, except for the use of dependent vector types. Furthermore, as it is a coordination language, functions and input vectors only have a type declaration but no implementation.

The core language consists of:

- *vector* and *tuple* types (dependent types) as defined above;
- a set of opaque functions on atomic types $f_j :: a_i \to a_k$, i.e. the implementation of the function is not part of the language;
- the following primitive higher-order functions, with semantics defined in terms of Haskell Prelude functions:

```
map = Prelude.map
fold = Prelude.foldl
unzipt = Prelude.unzip
zipt = \(v1,v2) -> Prelude.zip v1 v2
```

  - *primitive* in this context means that e.g. `map` is not defined in terms of any other lower-level language construct.

[4] http://https://github.com/wimvanderbauwhede/tytra

```
-- type signatures for input vector and
-- opaque functions omitted for brevity
v_1 = map f_1 v_in
(v_1,l,v_1,r) = unzipt v_1
s_2,l = fold f_2,l s_0 v_1,l
v_2,r = map f_2,r v_1,r
v_out = map (f_3 s_2,l) v_2,r
```

Listing 1: Example TyTra-CL program

  - `map` and `fold` apply the opaque functions to vectors;
  - `zipt` and `unzipt` convert between tuples of vectors and vectors of tuples.
- function composition (∘) and lambda functions
- `let`-bindings (assignment)

This set of primitives allows the creation of surprisingly complex and flexible programs.

In the remainder we will normalise the shape of the program somewhat: for the sake of simplicity and clarity, every program will be written as a series of let bindings. This is not a limitation as let-bindings are syntactic sugar for lambda expressions. However, a program written in this shape makes the edges and nodes in the graph explicit. For example, consider the program in Listing 1 and the corresponding graph representation in Figure 2. The correspondence between the variables and function applications in the code and the edges and nodes in the graphs is quite clear.

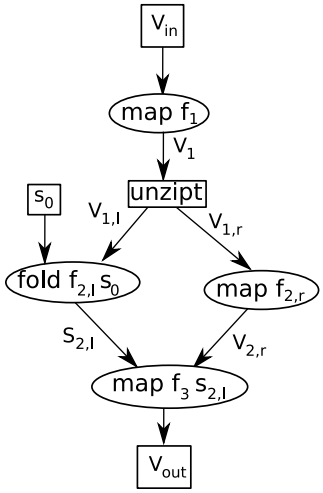

Fig. 2 Dataflow graph for example program

4.1 Deriving program transformations from type transformations

Given a type transformation on a vector type, what is the corresponding transformation on the program type and on the actual program? Within the constraints of the language definition, it is straightforward to derive the transformed program:

- The *Split* and *Merge* type transformations have corresponding `split` and `merge` functions that operate on the vector values:

$$split :: (k :: Int) \to \mathrm{Vec}\, k.m\, b \to \mathrm{Vec}\, k\, \mathrm{Vec}\, m\, b$$
$$merge :: \mathrm{Vec}\, k\, \mathrm{Vec}\, m\, b \to \mathrm{Vec}\, k.m\, b$$

  - The identity $\texttt{merge}\,(\texttt{split}\, k\, v) = v$ holds iff $v :: \mathrm{Vec}\ k.m\ b$;
  - the inverse $\texttt{split}\, k\,(\texttt{merge}\, v) = v$ holds iff $v :: \mathrm{Vec}\ k\ \mathrm{Vec}\ m\ b$.
- The program transformations derived from the *Split* are:

```
map f v = merge ((map . map) f (split k v))
fold f acc v = (fold . fold) f acc (split k v))
zipt (v1,v2) =  merge . (map zipt) . zipt  (split k v1, split k v2)
unzipt v = merge (unzipt . (map unzipt) . (split k v))
```

- The program transformations derived from *Merge* are:

```
(map . map) f v = split k ((map . map) f (merge v))
(fold . fold) f acc v = fold f acc (merge v)
(map zipt) .  zipt (v1,v2) = split k (zipt  (merge v1, merge v2))
(unzipt . (map unzipt)) v = split k (unzipt (merge v))
```

- Thus the transformed program produces provably the same result as the original program.

4.2 Generalising the primitive functions

In general, every vector can be split in many different ways. We therefore generalise the functions `split`, `merge`, `map`, `fold`, `zipt` and `unzipt` into n-dimensional versions *ndsplit*, *ndmerge*, *ndmap*, *ndfold*, *ndzipt* and *ndunzipt*:

- Definitions:

$\texttt{ndsplit :: Int}\ n, k_1, k_2, ... \,|\, \prod\ k_i = n \Rightarrow [k_1, k_2, ...] \to \texttt{Vec}\ n\, a \to\ \texttt{NDVec}\,[k_1, k_2, ...]\, a$
$\texttt{ndmerge :: Int}\, n, k_1, k_2, ... \,|\, \prod\ k_i = n \Rightarrow\ [k_1, k_2, ...] \to\ \texttt{NDVec}\,[k_1, k_2, ...]\, a \to\ \texttt{Vec}\, n\, a$
$\texttt{ndmap :: } (a \to\ b) \to\ \texttt{NDVec}\,[\ k_1, k_2, ...]\, a \to\ \texttt{NDVec}\,[\ k_1, k_2, ...]\, b$
$\texttt{ndfold :: } (a \to\ b \to\ a) \to\ a \to\ \texttt{NDVec}\,[\ k_1, k_2, ...]\, b \to\ a$

- We can show that for every valid list of factors ns:

```
(ndmerge ns) . (ndmap f) . (ndsplit ns) = map f
(ndfold f acc) . (ndsplit ns) = fold f acc
```

- We can redefine the LHS as a function of ns:

```
pndmap ns f = (ndmerge ns) . (ndmap f) . (ndsplit ns)
pndfold ns f acc = (ndfold f acc) . (ndsplit ns)
```

- For completeness we also define *ndzipt* and *ndunzipt*:

```
ndzipt :: Int n, k1, k2, ... | ∏ ki = n ⇒
[k1, k2, ...] → ( NDVec [k1, k2, ...] a, NDVec [k1, k2, ...] b, ...) → NDVec [k1, k2, ...] (a, b, ...)
ndunzipt :: Int n, k1, k2, ... | ∏ ki = n ⇒
[k1, k2, ...] → NDVec [k1, k2, ...] (a, b, ...) → (NDVec [k1, k2, ...] a NDVec [k1, k2, ...] b, ...)
```

– Again, we can show that for every valid list of factors ns:

```
zipt = (ndmerge ns) . (ndzipt ns) . (\(v1,v2,...)→
(ndsplit ns v1, ndsplit ns v2, ...)
unzipt = (\(v1,v2,...)→
(ndmerge ns v1, ndmerge ns v2, ...) . (ndunzipt ns) . (ndsplit ns)
```

and we can redefine the LHS to *pndzipt* and *pndunzipt*

– In conclusion, we have shown that for every valid list of factors ns:

```
pndmap ns   = map
pndfold ns  = fold
pndzipt ns   = zipt
pndunzipt ns  = unzipt
```

Thus we have shown that in order to transform the actual netlist (dataflow graph) of the program to optimize its throughput, we do not actually need to transform the program at all! All that is required is the trivial operation of substitution of the primitive functions with their pnd* counterparts.

5 Transforming the TyTra-CL AST

In practice, we do not transform programs in the TyTra-CL. Instead, we transform the Abstract Syntax Tree (AST) of the program.

5.1 Abstract Syntax for the TyTra-CL

The definition of the core of the TyTra-CL abstract syntax in Haskell is given in Listing 2.

The TyTra-CL program is parsed, normalised to a list of assignments and transformed into the actual AST.

The key points to note about this AST are the following:

– The opaque function nodes MOpaque and FOpaque, which represent functions $a \rightarrow b$ and $b \rightarrow a \rightarrow b$ respectively, have an associated PerfCost field which holds the performance and cost figures bases on the TyTraIR implementation (see Section 6). The FOpaque node also has an attribute Assoc to indicate if the operation is associative or not. The [Expr] field is used for extra arguments, see the example.

– The NDMap node takes a list of tuples [(Integer, MVariant)], and similar for NDFold. The variant indicates how the map or fold is implemented. A map can be implemented purely sequentially, as a streaming pipeline or in parallel; a fold (reduction) can be implemented purely sequentially or as a tree if the operation is associative.
For example for a 2-D vector of size 1024, [(1024,Seq)] would mean process 1024 elements sequentially; [(8,Par),(128,Pipe)] means that there will be 8

```
data Type = Vec Type      | Tuple [Type] | Prim ...
data Action =
    MOpaque Name [Expr] Type Type PerfCost |
    FOpaque Assoc Name [Expr] Expr Type Type PerfCost |
    PNDMap [Int] Action |
    PNDFold [Int] Action |
    NDMap [(Int, MVariant)] MVariant Action |
    NDFold [(Int, FVariant)] FVariant Action |
    NDSplit [Int] |
    NDMerge [Int] |
    NDDistr [Int] [Int] |
    NDZipT [Int] [Type] |
    NDUnzipT [Int] Type |
    Compose [Action] |
    Let Expr Expr
data Expr = Var Name Type | Res Action Expr | Tup [Expr]
data Assignment = Assign Expr Expr
data MVariant = Par | Pipe | Seq
data FVariant = Tree | FPipe | FSeq
type TyTraCLProgram = Assignment
```

Listing 2: TyTra-CL Abstract Syntax Tree

parallel pipelines that each process 128 scalar elements; [(128 ,Pipe),(8,Par)] would mean that there is a single pipeline which processes 128 elements as vectors of size 8.
Each of these choices comes with a different cost in terms of requirements for logic gates and buffers, and also a different performance in terms of latency and throughput.

- As pointed out earlier, NDSplit, NDMerge and NDDistr represent (de)multiplexing operations, and as such the incur a cost and impact on the performance, so they also carry the (Performance, Cost) tuple.
- Initially, the partitioning lists for the various node types, e.g. the [Int] in PNDMap, are empty, which means that no vectors have been transformed.

5.2 AST Transformations and Cost/Performance Calculations

The AST is transformed in three steps:

- Decompose PND* nodes:
  This step is performed only once. The PNDMap and PNDFold operations are replaced by their definitions in terms of NDMerge, NDSplit and NDMap or NDFold and similar for PNDZipT and PNDUnzipT .
- Combine NDMerge/NDSplit pairs into NDDistr:
  Consecutive NDMerge/NDSplit pairs are replaced by NDDistr. This step is also performed only once, and is essential because NDMerge will always return a 1-D vector (i.e. a sequential stream), which is then split by a subsequent NDSplit. This is a potential bottleneck. Instead, NDDistr will generate the most efficient $n \times m$ multiplexer.

```
p :: TyTraCLProgram
p = Assign (Var (MkName "v_out") (Vec (Prim T_out)  n))
 (Res (Let (Var (MkName "v_1") (Vec (Tuple [Prim T_fl ,Prim T_fl])  n))
 (Res (Let (Tup [Var (MkName "v_1,l") (Vec (Prim T_v1,l)  n),Var (MkName "v_1,r") (Vec (Prim T_v1,r)  n)]) (Res (Let (Var (MkName
 (Res (Let (Var (MkName "v_2,r") (Vec (Prim T_v2,r)  n))
 (Res (PNDMap [] (MOpaque (MkName "f_3") [Var (MkName "s_2,l") (Prim T_s2,l)] (Prim T) (Prim T) cp_f3))
 (Var (MkName "v_2,r") (Vec (Prim T_v2,r)  n))
))
 (Res (PNDMap [] (MOpaque (MkName "f_2,r") [] (Prim T) (Prim T) cp_f2,r))
 (Var (MkName "v_1,r") (Vec (Prim T_v1,r)  n))
))
) (Res (PNDFold [] (FOpaque (MkAssoc True) (MkName "f_2,l") [] (Var (MkName "s_0") (Prim T_s0))
 (Prim T) (Prim T) cp_f2,l))
 (Var (MkName "v_1,l") (Vec (Prim T_v1,l)  n))
))
) (Res (NDUnzipT []) (Var (MkName "v_1") (Vec (Tuple [Prim T_v1,l ,Prim T_v1,r])  n))
))
) (Res (PNDMap [] (MOpaque (MkName "f_1") [] (Prim T) (Tuple [Prim T,Prim T]) cp_f1))
 (Var (MkName "v_in") (Vec (Prim T_in)  n))
))
```

Listing 3: AST for example program

– Partitioning and Variants:
  The AST in this form can be used for obtaining cost/performance estimates simply by populating the partitioning lists (the list describing the nesting of a vector) and variants (i.e. sequential, parallel or pipelined) and generating the TyTraIR. This means that the actual program transformation are essentially expressed as lists of numbers, which makes it possible to use a variety of optimisation and machine learning approaches to find the optimal variant.

The AST for the above example is given by Listing 3. The types of the input vectors and all opaque functions are provided by the type signatures in the program.

6 The TyTra Intermediate Representation Language

Programs written in the TyTra-CL are not directly costable. To obtain costs we could emit HDL and perform synthesis and Place & Route, but that is very time-consuming. Our approach is to define an Intermediate Representation (IR) language, which we call the *TyTra-IR*. The design variants obtained by transforming the AST are converted into a TyTra-IR representation and are costed by parsing and analysing the IR. The IR has semantics that can express the platform, memory, execution, design-space and streaming data-pattern models described in the previous section. It is also used to generate the final HDL for synthesis and deployment on the FPGA.

The TyTra-IR is currently used to express (and cost) the device-side code only, and as our high-level programming model is dataflow, it models all computations on a dataflow machine rather than a Von-Neumann architecture.

Our approach for providing an API to access the FPGA configuration generated from a TyTra-IR description is to encapsulate the generated HDL code as a kernel of a commercially available HLS framework. This enables the use of memory controllers and peripheral logic generated by the HLS framework, as well as being able to make use of its API. Xilinx, Altera, and Maxeler provide routes to integrating HDL code into their HLS frameworks, and Xilinx and Altera provide OpenCL compatible API.

In terms of syntax, the TyTra-IR is strongly and statically typed, and all computations are expressed using Static Single Assignment (SSA). The syntax (but not the semantics) is based on the LLVM-IR, with extensions for parallelism suitable for an FPGA target. This choice provides us with a baseline for designing our language, and leaves the route open to explore LLVM optimizations, as e.g. the LegUp [1] tool does.

The TyTra-IR code for a design has two components. The *Manage-IR* and the *Compute-IR*. The motivation behind dividing TyTra-IR into two components is to separate the pure dataflow architecture operating on data *streams*, from the control and peripheral logic that creates these streams.

The Manage-IR has semantics to instantiates *memory-objects* which is abstraction for any entity that can be the source or sink for a stream. In most cases, a memory object's equivalent in a software description would be an array in main memory. It also has *stream-objects*, connecting a streaming port in the processing element to a memory-object.

The compute-IR describes the processing element(s), which by default is a streaming datapath implementation of the kernel. A design is constructed by creating a hierarchy of IR *functions*, which may be considered equivalent to *modules* in an HDL like Verilog. However, these functions are described at a much higher abstraction than HDL, with a keyword specifying the parallelism pattern to use in this function. These keywords are: `pipe` (pipeline parallelism), `par` (thread parallelism), `seq` (sequential execution) and `comb` (a custom combinatorial block). By using different parent-child and peer-peer combinations of functions of these four types, we can practically capture the entire design-space of the FPGA in terms of expressed parallelism. The TyTra compiler parses the IR description of a design-variant expressed using these parallelism constructs, and extracts the architecture from it.

Listing 4 shows the TyTra-IR for the baseline configuration which is a single kernel-pipeline, for a successive over-relaxation (SOR) kernel. The Manage-IR which declares the memory and stream objects is not shown.

Note the creation of offsets of input stream `p` in lines `6-9`, which create streams for the six neighbouring elements of `p`. These offset streams, together with the input streams shown in lines `2-4` form the *input tuple* that is fed into the datapath pipeline described in lines `10-15`. Figure 4 shows the kernel's realization as a pipeline. The same SOR example can be expressed in the IR to represent *thread-parallelism* by adding multiple *lanes*, corresponding to a reshaped data along e.g four rows, by encapsulating multiple instances of the kernel-pipeline function shown in Listing 4 into a top-level function of type

```
; **** COMPUTE-IR ****
@main.p  = addrSpace(12) ui18,
           !"istream", !"CONT", !0, !"strobj_p"
;...[more inputs]...
define void @f0(...args...) pipe {
  ;stream offsets
  ui18 %pip1=ui18 %p, !offset, !+1
  ui18 %pkn1=ui18 %p, !offset, !-ND1*ND2
    ;...[more stream offsets]...
  ;datapath instructions
  ui18 %1 = mul ui18 %p_i_p1, %cn2l
  ui18 %2 = mul ui18 %p_i_n1, %cn2s
    ;..[more instructions]...
  ;reduction operation on global variable
  ui18 @sorErrAcc=add ui18 %sorErr, %sorErrAcc
}
define void @main () {
  call @f0(..args...) pipe }
```

Listing 4: Abbreviated TyTra-IR code for the SOR kernel configured as a single pipeline lane.

`par`, and creating multiple stream objects to service each of these parallel kernel-pipelines. This is shown in page 14.

```
; **** COMPUTE-IR ****
@main.p0  = addrSpace(12) ui18,
            !"istream", !"CONT", !0, !"strobj_p"
@main.p1  = ...
@main.p2  = ...
@main.p3  = ...
;...[other inputs]...
define void @f0(...args...) pipe {...}
define void @f1 (...args...) par {
  call @f0(...args...) pipe
  call @f0(...args...) pipe
  call @f0(...args...) pipe
  call @f0(...args...) pipe }
define void @main () {
  call @f1(..args...) par }
```

Listing 5: Abbreviated TyTra-IR code for the SOR kernel configured with multiple pipelines lanes corresponding to reshaped data.

7 The TyTra Cost Model

The TyTra cost model is an analytical model that uses parameters obtained from data sheets as well as test benches. It is explained in detail in [12,11]. In this section we focus on the use of this cost model rather than its implementation details.

We have developed a back-end compiler[5] that accepts a design variant in TyTra-IR, costs it and, if required, generates the HDL code for it, as shown in Figure 3.

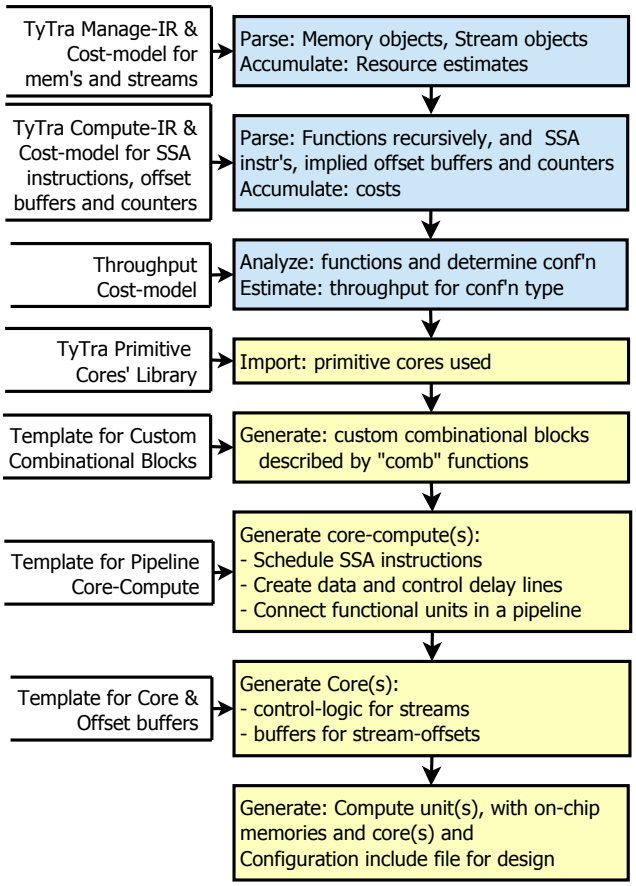

Fig. 3 The TyTra back-end compiler flow, showing the estimation flow (blue/first three stages) and code generation flow (yellow). The starting point for this subset of the entire flow is the TyTra-IR description representing a particular design variant, ending in the generation of synthesizeable HDL which can then be integrated with a HLS framework.

As discussed earlier, we can use the type transformations to generate variants of the program by reshaping the data, which means we can take a single stream of size N and transform it into L streams of size $\frac{N}{L}$, where L is the number of concurrent lanes of execution in the corresponding design variant. Figure 5 shows evaluation of variants thus generated.

For maximum performance, we would like as many lanes of execution as the resources on the FPGA allow, or until we saturate the IO bandwidth.

[5] https://github.com/waqarnabi/tybec

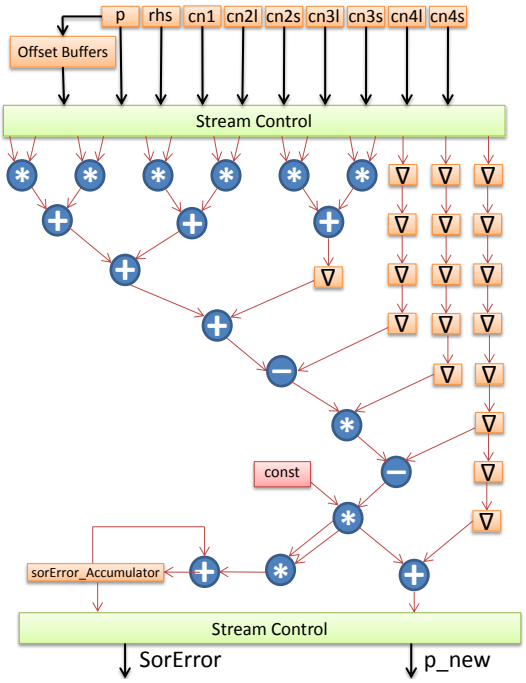

Fig. 4 Illustration of the pipelined datapath of the SOR kernel generated by our compiler. Only pass-through pipeline buffers are shown; all functional units have pipeline buffers as well. The blocks at edges refer to on-chip memory for each data.

If data is transported between the host and device, then beyond 4 lanes, we encounter the *host communication wall*.

If all the data is made available in the device's global (on-board) memory then the communication wall moves to about 16 lanes. We encounter the *computation-wall* at six lanes, where we run out of LUTs on the FPGA. However, we can see other resources are underutilized, and some sort of resource-balancing can lead to further performance improvement.

We would like to highlight here that the estimator is very fast: the current implementation, although written in Perl, takes only 0.3 seconds to evaluate one variant. This is more than $200\times$ faster than e.g. the preliminary estimates generated by SDAccel which takes close to 70 seconds. We expect that for larger designs the relative performance would be even better.

7.1 Accuracy of the cost model

Preliminary results on relatively small but realistic scientific kernels have been very encouraging. We evaluated the estimated vs actual utilization of resources, as well as throughput measured in terms of cycles-per-kernel-instance in Table 1. We tested the cost model by evaluating the integer version of

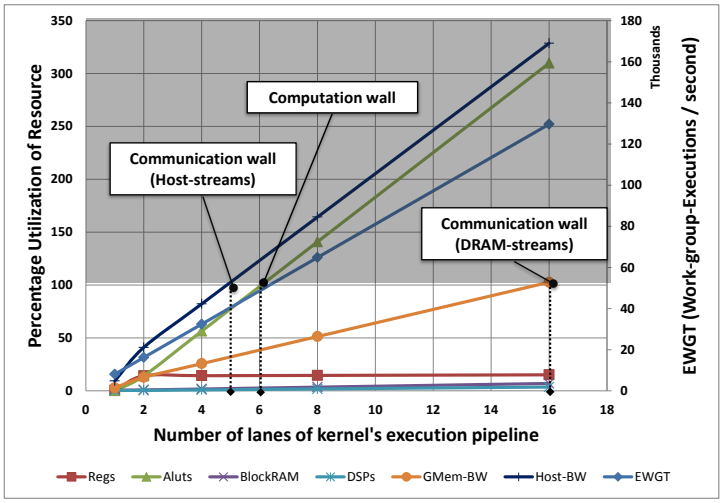

Fig. 5 Evaluation of variants for the SOR kernel generated by changing the number of kernel-pipelines (16 data points and 10 kernel iterations).

Kernel		ALUT	REG	BRAM	DSP	C/KI
Hotspot (Rodinia)	Estimated	391	1305	32.8K	12	262.3K
	Actual	408	1363	32.7K	12	262.1K
	% error	4	4.2	0.3	0	0.07
LavaMD (Rodinia)	Estimated	408	1496	0	26	111
	Actual	385	1557	0	23	115
	% error	6	3.9	0	13	3.4
SOR	Estimated	528	534	5418	0	292
	Actual	534	575	5400	0	308
	% error	1.1	7.1	0.3	0	5.2

Table 1 The estimated vs actual performance and utilization of resources, the former measured in terms of cycles-per-kernel-instance (CKI), for three scientific kernels. Percentage errors also shown.

kernels chosen from three HPC scientific applications: 1) The successive over-relaxation kernel from the LES weather model that has been discussed earlier; 2) The *hotspot* benchmark from the Rodinia HPC benchmark suite[2], used to estimate processor temperature based on an architectural floorplan and simulated power measurements;

3) The *lavaMD* molecular dynamics application also from Rodinia, which calculates particle potential and relocation due to mutual forces between particles within a large 3D space

These results confirm our observation that an IR constrained at an appropriate abstraction will allow quick estimates of cost and performance that are accurate enough to make design decisions.

8 Exemplar: Successive Over-Relaxation (SOR)

We consider an SOR kernel (Listing 6), taken from the code for the Large Eddy Simulator for Urban Flows, an experimental weather simulator [13]. The kernel iteratively solves the Poisson equation for the pressure and is the most time-consuming part of the simulator. The main computation is a stencil over the neighbouring cells (which is inherently parallel).

```
do l=1,nmaxp
  do k=1,km
    do j=1,jm
      do i=1,im
          reltmp = omega*(cn1(i,j,k)*(cn2l(i)*p(i+1,j,k) &
          +cn2s(i)*p(i-1,j,k) &
          +cn3l(j)*p(i,j+1,k)+cn3s(j)*p(i,j-1,k) &
          +cn4l(k)*p(i,j,k+1)+cn4s(k)*p(i,j,k-1) &
          -rhs(i,j,k))-p(i,j,k))
          p(i,j,k) = p(i,j,k) + reltmp
      end do
    end do
  end do
end do
```

Listing 6: SOR kernel Fortran code.

8.1 The route from Fortran

In practice, most scientific code in the field of numerical weather prediction and climate simulation is written in Fortran. It might therefore seem that our functional language based approach is impractical. In this section we explain the route from Fortran to the TyTra-CL and TyTra-IR.

Our observation has been that programs in our application domain are typically using loops to operate on static arrays. The SOR kernel above is a good example. We have created a compiler[6] which analyses Fortran code in terms of *map* and *fold* and generates a subroutine corresponding to the body of every loop nest. The code in this form is not only ready for parallelisation with e.g. OpenCL, but effectively consists of sequences of applications of maps and folds applied to opaque functions, so that we can convert it directly into a TyTra-CL program.

Our compiler transforms the complete Large Eddy Simulator main loop code (i.e. everything except file I/O) into 33 kernels, 29 of which are maps and 4 folds. This code is representative for many computational fluid dynamics codes and illustrates the validity of our approach.

[6] https://github.com/wimvanderbauwhede/AutoParallel-Fortran

We have also created a compiler which refactors the code[7] to create code in the form required by our opaque functions, i.e. instead of array accesses, the functions can only take scalars. In practice this means that the function takes a tuple of variables corresponding to every array access in a loop nest. The compiler has a C backend and we use the LLVM clang front-end to generate LLVM IR, which is used as input for our TyTra-IR toolchain. The TyTra-IR compiler transforms the original LLVM IR into a pipelined datapath as explained in Section 6. Consequently, at the level of the opaque function the operation is always pipelined.

This is currently work in progress because the emitter from the map-and-fold based transformed Fortran code to TyTraCL is not ready. Our current flow emits OpenCL host and kernel code [18].

8.2 Transforming the SOR kernel

The TyTra-CL version of the above program is very simple:

```
p_in :: Vec (im*jm*km) Float
f_sor :: Float -> Float
p_out = map f_sor p_in
```

The corresponding AST is equally simple:

```
sor :: TyTraCLProgram
sor = Assign
  (Var "p_out" (im*jm*km) Float )
  (Res
     (PNDMap [] (MOpaque "f_sor" [] Float Float cp_sor ))
     (Var "p_in" (im*jm*km) Float ))
```

We can now apply a transformation where we for example split the vector into 4 parts and interpret this as 4 parallel lanes by using the Par variant of NDMap. We have expanded the AST as described before. The cost and performance for the actual kernel f_{sor} have been obtained from its IR representation using the TyTra-IR compiler.

```
sor :: TyTraCLProgram
sor = Assign
  (Var "p_out" (im*jm*km/4) Float )
  (Res (NDMerge [4])
    (Res (NDMap [(4,Pipe)] Par
       (MOpaque "f_sor" [] Float Float cp_sor))
         (Res (NDSplit [4]) (Var "p_in" (im*jm*km/4) Float ))
  ))
```

The IR resulting from this transformation is shown in Figure 5. Clearly, we can now generate many variants by changing the number of parts into which we split the vector and the variant for each part. However, as explained in Section 7, the cost model informs us that for splits higher than 4 we already hit the resource limits so the above transformation is the best candidate.

[7] https://github.com/wimvanderbauwhede/RefactorF4Acc

8.3 Case Study: A TyTra generated solution vs a commercial HLS solution

A working solution using an FPGA accelerator requires a "base platform" design on the FPGA to deal with off-chip input/output and other peripheral functions, as well as an API for accessing the FPGA accelerator. Our approach in creating working solutions with our flow is to use a commercially available HLS solution – Maxeler – and insert the HDL code generated for the design by our back-end compiler into that framework. We have demonstrated a prototype solution using the Maxeler HLS flow, which allows one insertion of custom HDL in their otherwise high-level design entry language *MaxJ*. Maxeler is an HLS design tool for FPGAs, and provides a Java meta-programming model for describing computation kernels and connecting data streams between them. Integrating custom code with Maxeler requires creation of a wrapper kernel written in its kernel language *MaxJ* for the custom HDL module.

Our setup is a Maxeler desktop solution, with an intel-i7 quad-core processor at 1.6GHz, and 32 GB RAM. The FPGA board is a *Maxeler Maia DFE*, which contains an Altera Stratix-V-GSD8 device with 695K Logic Elements. The host-device communication is over PCIe-gen2-x8.

For performance comparison, we have collected the runtime of the SOR kernel's three different implementations (Figure 6). The baseline is a simple Fortan-based CPU implementation (*cpu*) compiled with `gcc -O2`. The first FPGA implementation is using only the Maxeler flow (*fpga-maxJ*), which incorporates pipeline parallelism automatically extracted by the Maxeler compiler. The second FPGA implementation (*fpga-tytra*) is the design variant generated by the TyTra back-end compiler, based on a high type-transformation that introduced thread-parallelism (4 lanes) in addition to pipeline parallelism. We collected results for different dimensions of the input 3D arrays, i.e. `im, jm, km`, ranging from 24 elements along each dimension (55 KB) to 194 elements (57 MB). The number of iterations of the SOR kernel per run was fixed at at `nmaxp=1000` [8].

Apart from the smallest grid-size, *fpga-tytra* consistently outperforms *fpga-maxJ* as well as *cpu*, showing up to 3.9x and 2.6x improvement over *fpga-maxJ* and *cpu* respectively. At small grid-sizes though, the overhead of handling multiple streams per input and output array dominates and we have relatively less improvement or even a decrease in performance. In general, FPGA solutions tend to perform much better than CPU at large dimensions.

An interesting thing to note for comparison against the baseline CPU performance is that at the typical grid-size where this kernel is used in weather models (around 100 elements / dimension), the *fpga-maxJ* version is *slower* than *cpu*, but *fpga-tytra* is 2.75x faster. These performance results clearly indicate that an straightforward implementation of a kernel using an HLS tool may not fully exploit the parallelism and performance achievable on an FPGA device.

[8] Our results show that the relative performance and energy consumption results hold across different values of `nmaxp` – the number of times the SOR kernel repeats – and changes only with changing grid-size.

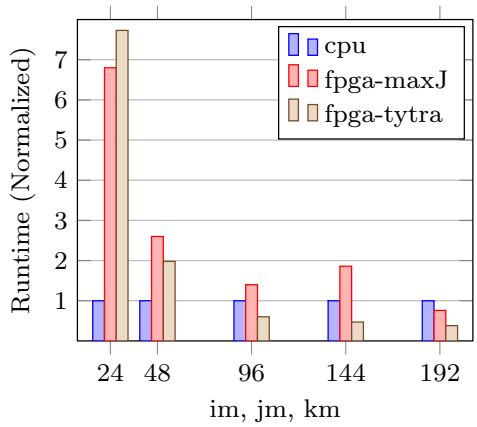

Fig. 6 Runtime of the SOR kernel for different sizes of grid, normalized against the CPU-only solution. The figures are for 1000 iterations of the kernel.

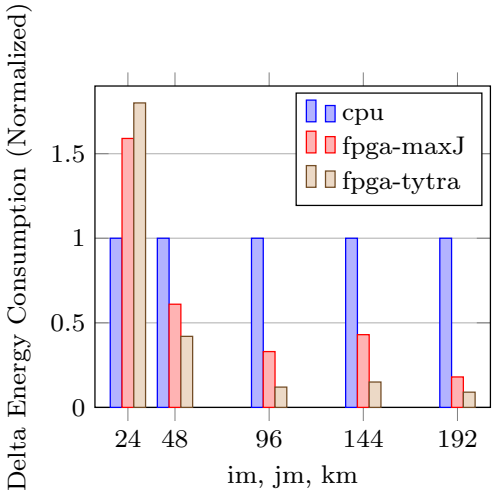

Fig. 7 *Increase* from idle energy-consumption, for calculating the SOR kernel for different sizes of grid, normalized against the CPU-only solution. The figures are for 1000 iterations of the kernel.

For the energy figures, we used the actual power consumption of the host+device node on a power-meter. For a fair comparison, we noted the increase in power from the idle CPU power, for both CPU-only and CPU-FPGA solutions. As shown in Figure 7, FPGAs very quickly overtake CPU-only solutions, and *fpga-tytra* solution shows up to 11x and 2.9x power-efficiency improvement over *cpu* and *fpga-maxJ* respectively. The energy comparison further demonstrates the utility of adopting FPGAs in general for scientific kernels, and specifically our approach of using type transformations for finding the best design variant.

9 Conclusion

FPGAs are increasingly being used in HPC for acceleration of scientific kernels. While the typical route to implementation is the use of High-Level Synthesis tools like Maxeler or OpenCL, such tools may not necessarily fully expose the parallelism in the FPGA in a straightforward manner. Hand-tuning designs to exploit the available FPGA resources on these HLS tools is possible but still requires considerable effort and expertise.

We have presented an original compilation flow that, starting from high-level program in a functional coordination language based on opaque, costable functions and higher-order functions, generates correct-by-construction design variants using type-driven program transformations, evaluates the generated variants using an analytical cost model on an intermediate description of the kernel, and emits the HDL code for the optimal variant.

We have introduced the theoretical framework of the vector type transformations and shown how program transformations in our TyTra-CL language can be automatically derived from the type transformations, with guaranteed correctness. We have explained compilation from TyTra-CL to TyTra-IR and the costing of the designs using the analytical cost model. The accuracy of the cost model was shown across three kernels: a kernel from the LES weather simulator, and two kernels from the Rodinia benchmark.

A case study based on the successive over-relaxation kernel from a real-world weather simulator was used to demonstrate the high-level type transformations. It was also used to give an illustration of a working solution based on HDL code generated from our compiler, shown to perform better than the baseline Maxeler HLS solution.

Our work presents a proof of concept for a solution which allows high level programming with a route from legacy Fortran code, and which will automatically converge on the best design variant from a single high-level description of the algorithm in a functional language through a combination of correct-by-construction program transformation and fast analytical cost models. Our future work focuses on completing the compiler toolchain, and in particular investigating the use of machine learning to explore the potentially very large search space of transformed programs.

Acknowledgements The authors acknowledge the support of the UK EPSRC for the TyTra project (EP/L00058X/1).

References

1. Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J.H., Brown, S., Czajkowski, T.: Legup: High-level synthesis for FPGA-based processor/accelerator systems. In: Proceedings of the 19th ACM/SIGDA International Symposium on FPGAs, FPGA '11, pp. 33–36. ACM, New York, NY, USA (2011)
2. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: Workload Characteriza-

tion, 2009. IISWC 2009. IEEE International Symposium on, pp. 44–54 (2009). DOI 10.1109/IISWC.2009.5306797
3. Cole, M.: Algorithmic skeletons. In: Research Directions in Parallel Functional Programming, pp. 289–303. Springer (1999)
4. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Computing **30**(3), 389 – 406 (2004). DOI http://dx.doi.org/10.1016/j.parco.2003.12.002
5. Czajkowski, T., Aydonat, U., Denisenko, D., Freeman, J., Kinsner, M., Neto, D., Wong, J., Yiannacouras, P., Singh, D.: From OpenCL to high-performance hardware on FPGAs. In: Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on, pp. 531–534 (2012). DOI 10.1109/FPL.2012.6339272
6. Darlington, J., Field, A., Harrison, P., Kelly, P., Sharp, D., Wu, Q., While, R.: Parallel programming using skeleton functions. In: PARLE'93 Parallel Architectures and Languages Europe, pp. 146–160. Springer (1993)
7. Kaul, M., Vemuri, R., Govindarajan, S., Ouaiss, I.: An automated temporal partitioning and loop fission approach for fpga based reconfigurable synthesis of dsp applications. In: Proceedings of the 36th Annual ACM/IEEE Design Automation Conference, DAC '99, pp. 616–622. ACM, New York, NY, USA (1999). DOI 10.1145/309847.310010
8. Keinert, J.e.a.: Systemcodesigner;an automatic esl synthesis approach by design space exploration and behavioral synthesis for streaming applications. ACM Trans. Des. Autom. Electron. Syst. **14**(1), 1:1–1:23 (2009)
9. Keutzer, K., Ravindran, K., Satish, N., Jin, Y.: An automated exploration framework for fpga-based soft multiprocessor systems. In: Hardware/Software Codesign and System Synthesis, 2005. CODES+ISSS '05. Third IEEE/ACM/IFIP International Conference on, pp. 273–278 (2005). DOI 10.1145/1084834.1084903
10. Korinth, J., de la Chevallerie, D., Koch, A.: An open-source tool flow for the composition of reconfigurable hardware thread pool architectures. In: Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on, pp. 195–198. IEEE (2015)
11. Nabi, S.W., Vanderbauwhede, W.: Using type transformations to generate program variants for fpga design space exploration. In: 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig), pp. 1–6 (2015). DOI 10.1109/ReConFig.2015.7393365
12. Nabi, S.W., Vanderbauwhede, W.: A fast and accurate cost model for fpga design space exploration in hpc applications. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 114–123 (2016). DOI 10.1109/IPDPSW.2016.155
13. Nakayama, H., Takemi, T., Nagai, H.: Large-eddy simulation of urban boundary-layer flows by generating turbulent inflows from mesoscale meteorological simulations. Atmospheric Science Letters **13**(3), 180–186 (2012)
14. Pell, O., Averbukh, V.: Maximum performance computing with dataflow engines. Computing in Science Engineering **14**(4), 98–103 (2012). DOI 10.1109/MCSE.2012.78
15. Segal, O., Colangelo, P., Nasiri, N., Qian, Z., Margala, M.: Sparkcl: A unified programming framework for accelerators on heterogeneous clusters. CoRR **abs/1505.01120** (2015)
16. da Silva, B., Braeken, A., D'Hollander, E.H., Touhafi, A.: Performance modeling for FPGAs: Extending the roofline model with high-level synthesis tools. International Journal of Reconfigurable Computing (2013). DOI 10.1155/2013/428078
17. Thomas, D.B., Fleming, S.T., Constantinides, G.A., Ghica, D.R.: Transparent linking of compiled software and synthesized hardware. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2015, pp. 1084–1089 (2015)
18. Vanderbauwhede, W., Davidson, G.: Domain-specific acceleration and pragmatic auto-parallelization of legacy scientific code in fortran 77 using source-to-source compilation. In: Proceedings of ParCFD2017, the 29th International Conference on Parallel Computational Fluid Dynamics, Glasgow, UK, 15-17 May 2017, pp. 1–2 (2017)
19. Xilinx: The Xilinx SDAccel Development Environment (2014). URL `http://www.xilinx.com/publications/prod_mktg/sdx/sdaccel-backgrounder.pdf`