Correct-by-Construction Program Transformations and Cost Modelling for Optimising Streaming Programs on FPGAs

Wim Vanderbauwhede Syed Waqar Nabi

School of Computing Science
University of Glasgow, Glasgow, UK
{wim.vanderbauwhede,syed.nabi}@glasgow.ac.uk

Abstract

In this paper we present a new approach to program optimisation based on type-driven, correct-by-construction 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, in particular 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.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Compilers

Keywords FPGA, type-driven program transformations, cost models

1. Introduction

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 possible implementation. HLS tools such as Maxeler(Pell and Averbukh 2012), Altera-OpenCL(Czajkowski et al. 2012) and Xilinx SDAccel(sda 2014) have raised the abstraction of design entry considerably. Parallel programmers with highly specialised expertise are however 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, 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. This observation of a requirement for a higher abstraction design-entry is not novel. For example, researchers have proposed algorithmic skeletons to separate algorithm from architecture-specific parallel programming(Cole 2004). SparkCL(Segal et al. 2015) brings increasingly diverse architectures, including FPGAs, into the familiar Apache Spark framework.

In the context of high-level programming specifically for FPGAs, there is considerable prior work, there is considerable work that deals with high-level programming of FPGAs, 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 and requires knowledge of target hardware and the programming framework((Pell and Averbukh 2012; Czajkowski et al. 2012; Keinert 2009)), (2) compiler optimizations are limited to improving the overall architecture already specified by the programmer, with no real *architectural* exploration((Pell and Averbukh 2012; Czajkowski et al. 2012; Keinert 2009; Canis et al. 2011)), (3) solutions are based on a creating soft-microprocessors on the FPGA and not optimized for HPC((Canis et al. 2011; Keutzer et al. 2005)), (4) the exploration requires evaluation of variants that take a prohibitively long amount of time(Keinert 2009), or (5) the flow is limited to very specific application domain e.g. for image-processing or DSP applications(Kaul et al. 1999). The *Geometry of Synthesis* project(Thomas et al. 2015) is similar with its 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 (da Silva et al. 2013) on extending the roof-line analysis for FPGAs is quite relevant and something we are looking into for a more useful representation our cost-model, but the parallels do not extend beyond this aspect. Our proposal however is to allow design-entry at a more fundamental and generic abstraction, inspired by functional languages with expressive type systems such as Haskell[1] or Idris[2]. The resultant flow, which we call the *TyTra* flow, is based on *type-based program transformations*. The design-entry in our TyTra-HLL language is at a pure *functional* abstraction, and we leave the task of generating the best possible variant for a given target architecture to the compiler. Program variants are generated using type transformations and translated to the TyTra-IR intermediate language. The compiler calculates cost and performance estimates for the variants and emits HDL code. The HDL kernel-code is integrated with an existing HLS framework.

[1] http://www.haskell.org

[2] http://www.idris-lang.org/

The ultimate aim of our work is to compile scientific programs such as simulators for climate and weather to FPGAs. In this paper, we present a theoretical framework based on type-driven program transformation. This framework allows us to create very large numbers of variants of a given program with a guarantee of correctness. We combine this with a fast and accurate 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 application domain that we are interested in and the fact that FPGAs perform best when used as stream computation devices, we focus on programs that work on arrays of fixed size and that process these arrays in streaming fashion. In other words, we are concerned with dataflow graphs for processing finite streams of data. We will describe such finite stream through the abstraction of *vectors*, although in our work *vector* and *array* can often be used interchangeably.

We show in this paper that the topology of a dataflow graph 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 (isomorphism) between our proposed high-level functional array 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.

In what follows we assume the reader is familiar with statically type functional languages such as Haskell or Idris. In particular, we will use a Haskell-like syntax throughout.

The structure of the paper is as follows: first we define the relationship between dataflow graphs and types. Next we discuss vector types and their transformations. We introduce our high-level functional language and show how the transformation of the *program* can automatically be derived from he transformation of the *types* of the arguments. We then present an abstract syntax for our language that allows us to express these derived program transformations and generate code in our intermediate representation language, the TyTra-IR, which allows us to obtain performance and cost estimates. We briefly introduce the TyTra-IR and the cost model and finally present an example illustrating our approach and its effectiveness.

2. Defining Graphs through Types

Programs in our language (see Section 4) are purely functional and statically typed. Every program consists essentially of a tree of function calls. We slightly abuse the notion of a *type* and use it to define the topology of the function call graph.

- Every edge is identified by a unique vector type or scalar type;
  - Every node is either a computational node identified by a unique function type
  - or a tuple manipulation node used to marshall inputs and outputs of computational nodes.
- By *type transformation* we mean the transformation of the node type as a result of the transformation of the incoming edge types.

3. Types and Type Transformations

3.1 Type Variables and Sizes

a is an *atomic* type variable, i.e. representing a nullary type constructor.

b*,*c are general type variables, not necessarily atomic.

k*,*l*,*m*,*n are *sizes*, so $k, m, n \in \mathbb{N}_{>0}$

3.2 Vector types

- Let Vec k b be a *vector type*, i.e. it represents a vector of length k containing values of type b.
- 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 as follows:

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

3.2.1 Transformations on Vector Types

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

We posit two fundamental transformations on vector types:

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 these transformations, and in general any transformation on vector types, does not have any effect on *atomic types*:

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

In Appendix 1, we 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

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

3.2.2 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.

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.

3.3 Tuple types

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

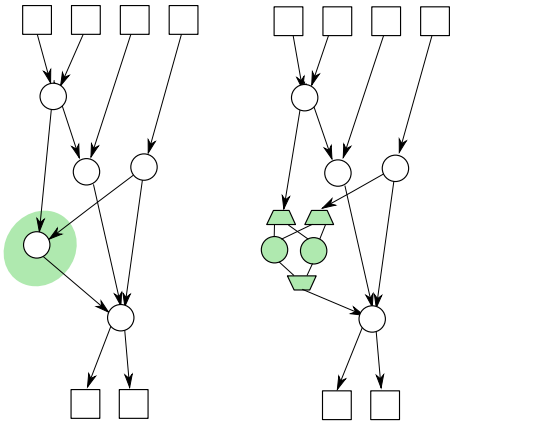

Figure 1. Parallelisation of a node in the computational graph

3.3.1 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:

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

3.3.2 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 (*uzipping*).

3.4 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:

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

4. TyTra-HLL: A simple functional high-level language for streaming programming

Now that we have defined vector types and their transformations, we want to show how the actual program transformation can be derived from the type transformations. To do so we first need to define our programming language.

TyTra-HLL is a very simple statically typed functional language with dependent types. The core language consists of:

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

```
map = Prelude.map
fold = Prelude.foldl
unzipt = Prelude.unzip
```

Listing 1 Example TyTra-HLL program

```
– type signatures for input vector and
– opaque functions omitted for brevity
v1 = map f1 v_in
(v_{1,l},v_{1,r}) = unzipt v1
s_{2,l} = fold f_{2,l} s0 v_{1,l}
v_{2,r} = map f_{2,r} v_{1,r}
v_out = map (f3 s_{2,l}) v_{2,r}
```

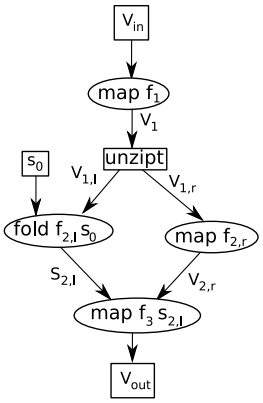

Figure 2. Dataflow graph for example program

```
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.
  - `map` and `fold` apply the opaque functions to vectors;
  - `zipt` and `unzipt` convert between tuples of vectors and vectors of tuples.
- function composition ($\circ$) 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.

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? We have shown in Appendix 2 that within the constraints of the language definition, it is possible and indeed straightforward to derive the transformed program:

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

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

The identity $\texttt{merge}\,(\texttt{split}\,k\,v) = v$ holds if $v :: \text{Vec}\ k.m\ b$; the inverse $\texttt{split}\,k\,(\texttt{merge}\,v) = v$ holds if $v :: \text{Vec}\ k\ \text{Vec}\ m\ b$.

- The program transformations derived from the type transformation *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 the type transformation *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 exactly the same result as the original program.

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:

- We define *ndsplit*, *ndmerge*, *ndmap* and *ndfold*:
  $\texttt{ndsplit} :: \text{Int}\, n, k_1, k_2, ... \mid \prod k_i = n \Rightarrow [k_1, k_2, ...] \rightarrow$
  $\text{Vec}\, n\, a \rightarrow \text{NDVec}\, [k_1, k_2, ...]\, a$
  $\texttt{ndmerge} :: \text{Int}\, n, k_1, k_2, ... \mid \prod k_i = n \Rightarrow [k_1, k_2, ...] \rightarrow$
  $\text{NDVec}\, [k_1, k_2, ...]\, a \rightarrow \text{Vec}\, n\, a$
  $\texttt{ndmap} :: (a \rightarrow b) \rightarrow [a]\langle k_1, k_2, ...\rangle \rightarrow [b]\langle k_1, k_2, ...\rangle$
  $\texttt{ndfold} :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b]\langle k_1, k_2, ...\rangle \rightarrow a$
- 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*:
  $\texttt{ndzipt} :: Int\, n, k_1, k_2, ... \mid \prod k_i = n \Rightarrow [k_1, k_2, ...] \rightarrow$
  $(\text{NDVec}\, [k_1, k_2, ...]\, a, \text{NDVec}\, [k_1, k_2, ...]\, b, ...) \rightarrow$
  $\text{NDVec}\, [k_1, k_2, ...]\, (a, b, ...)$
  $\texttt{ndunzipt} :: Int\, n, k_1, k_2, ... \mid \prod k_i = n \Rightarrow [k_1, k_2, ...] \rightarrow$
  $\text{NDVec}\, [k_1, k_2, ...]\, (a, b, ...) \rightarrow$
  $(\text{NDVec}\, [k_1, k_2, ...]\, a, \text{NDVec}\, [k_1, k_2, ...]\, b, ...)$
- 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)`

It is possible[3] to implement these polymorphic functions in Haskell, and with the correct implementations we can indeed verify that for all valid lists of factors the programs using the nd* functions produce an output identical to the original program. However, that is intrinsically a pointless exercise as the proofs already show that this is the case. The real value of the generalised functions lies in their use as code emitters for the transformed versions of the program.

[3] https://github.com/wimvanderbauwhede/tytra

Listing 2 TyTra-HLL Abstract Syntax Tree

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

5. Abstract Syntax for the TyTra-HLL

In practice, we do not transform programs in the TyTra-HLL. Instead, we transform the Abstract Syntax Tree (AST) of the program. The definition of the core of the TyTra-HLL abstract syntax in Haskell is given in Listing 2.

The TyTra-HLL program is parsed into this AST. In practice, the program is parsed into a parse tree; this parse tree is transformed to normalise the program, i.e. reduce it to the form of the example, a list of let-style assignments. Then the parse tree is 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 (Performance, Cost) tuple, which depends on their TyTraIR implementation (see Section). 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 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.

Listing 3 AST for example program

```
p :: TyTraHLLProgram
p = Let [
  Assign
    (Var "v_1" (Vec (Tuple [Prim T_{f,l},Prim T_{f,r}]) n_in))
    (Res
      (PNDMap [] (MOpaque "f_1" [] (Prim T_in) (Tuple
[Prim T_{f,l},Prim T_{f,r}]) (p_{f_1}, c_{f_1})))
      (Var "v_0" (Vec Prim n_in))),
  Assign
    (Tuple [Var "v_1l" (Vec (Prim T_{f,l}) n_in),Var "v_1r"
(Vec (Prim T_{f,r}) n_in)])
    (Res
      (NDUnzipt [] (Vec (Tuple [Prim T_{f,l},Prim T_{f,r}])
n_in)
      (Var "v_1" (Vec (Tuple [Prim T_{f,l},Prim T_{f,r}])
n_in))),
  Assign
    (Var "s_2l" (Prim T_{s_0})
    (Res
      (PNDFold [] (FOpaqueBin True "f_2l" [] (Var "s_0"
(Prim T_{s_0})) (Prim T_{s_0}) (Prim T_{f,l}) (p_{f_2l}, c_{f_2l})))
      (Var "v_1l" (Vec (Prim T_{f,l}) n_in))),
  Assign
    (Var "v_2r" (Vec (Prim T_{f_{2,r}}) n_in))
    (Res
      (PNDMap [] (MOpaque "f_2r" [] (Prim T_{f,r}) (Prim
T_{f_{2,r}}) (0,0)))
      (Var "v_1r" (Vec (Prim T_{f,r}) n_in))),
  Assign
    (Var "v_out" (Vec (Prim T_{f_3}) n_in))
    (Res
      (PNDMap [] (MOpaque "f_3" [Var "s_2" (Prim T_{s_0})]
(Prim T_{f_{2,r}}) (Prim T_{f_3}) (p_{f_3],c_{f_3}}))
      (Var "v_2r" (Vec (Prim T_{f_{2,r}}) n_in))))
  ] (Var "v_out" (Vec (Prim T_{f_3}) n_in))
```

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. Then the AST is transformed in three steps:

- First, the PNDMap and PNDFold operations are replaced by their definitions in terms of NDMerge, NDSplit and NDMap or NDFold
- Then, subsequent NDMerge/NDSplit pairs are replaced by NDDistr. This is because NDMerge will always return a 1-D vector, which is then split by a subsequent NDSplit. This is a potential bottleneck. Instead, NDDistr will generate the most efficient $n \times m$ multiplexer.

The AST in this form can be used for obtaining cost/performance estimates simply by populating the partitioning lists and variants and generating the TyTraIR. This is where the actual implementations of *ndmap* etc. are used: for each of the ND* nodes there is a corresponding nd* function which will generate the corresponding portion of the IR. The cost and performance of the resulting IR program are obtained using the cost model.

6. The TyTra Intermediate Representation Language

Programs written in the TyTra-HLL 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 (Canis et al. 2011) 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 `par`, and creating multiple stream objects to service each of these parallel kernel-pipelines. This is shown in page 6.

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

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

```
; **** 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 }
```

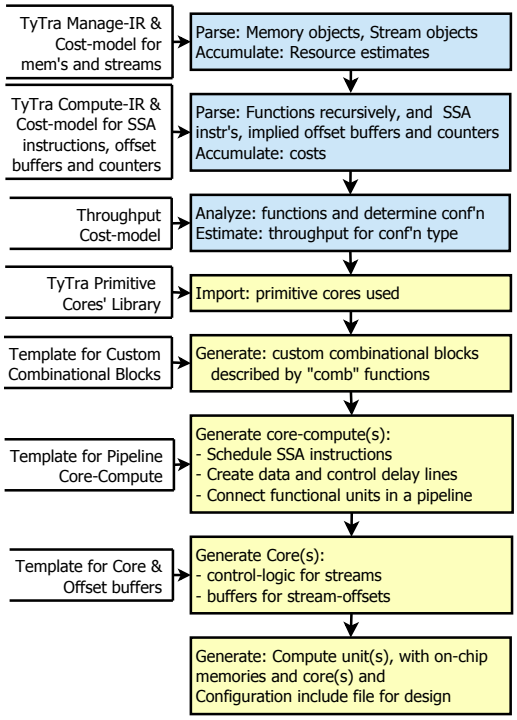

Figure 3. The TyTra back-end compiler flow, showing the estimation flow (blue/first three stages) and code generation flow (yellow). The starting point for this subset of the entire flow is the TyTra-IR description representing a particular design variant, ending in the generation of synthesizeable HDL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

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 (Nabi and Vanderbauwhede 2016, 2015). In this section we focus on the use of this cost model rather than its implementation details.

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

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. 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 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(Che et al. 2009), 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, taken from the code for the Large Eddy Simulator, an experimental weather simulator (Nakayama et al. 2012). The kernel iteratively solves the Poisson equation for the pressure. The main computation is a stencil over the neighbouring cells (which is inherently parallel).

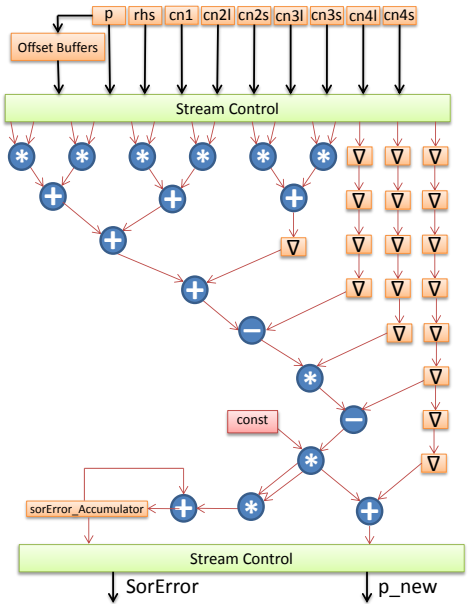

Figure 4. Illustration of the pipelined datapath of the SOR kernel generated by our compiler. Only pass-through pipeline buffers are shown; all functional units have pipeline buffers as well. The blocks at edges refer to on-chip memory for each data.

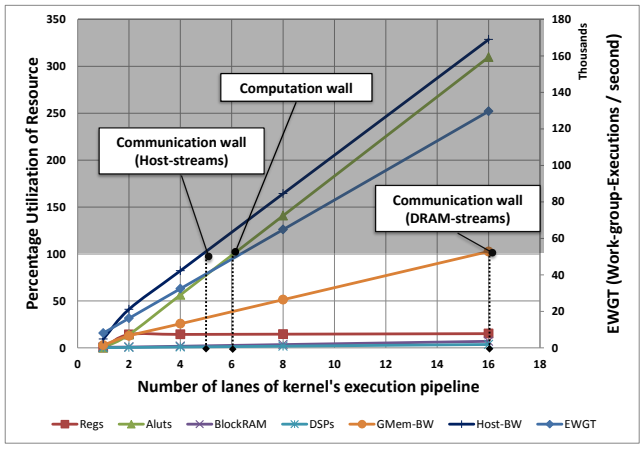

Figure 5. Evaluation of variants for the SOR kernel generated by changing the number of kernel-pipelines (16 data points and 10 kernel iterations).

```
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
```

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-HLL and TyTra-IR.

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.

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[4] 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 to opaque functions, so that we can convert it directly into a TyTra-HLL program.

We have also created a compiler which refactors the code[5] 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. We can translate this code to C and 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.

8.2 Transforming the SOR kernel

The TyTra-HLL 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 :: TyTraHLLProgram
sor = Let [
  Assign
    (Var "v_out" (Vec (Prim Float) (im*jm*km)))
      (Res
        (PNDMap [])
          (Opaque "f_sor" []
            (Prim Float)
            (Prim Float)
            (c_sor,p_sor)
          )
        )
        (Var "v_in" (Vec (Prim Float) (im*jm*km))))
  ] (Var "v_out" (Vec (Prim Float) (im*jm*km)) )
```

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

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

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

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 :: TyTraHLLProgram
sor = Let [
  Assign
    (Var "v_s"
      (Vec (Vec Float (im*jm*km/4)) 4))
    (Res
      (NDSplit [4])
      (Var "v_in" (Vec (Prim Float) (im*jm*km)))),
  Assign
    (Var "v_m"
      (Vec (Vec Float (im*jm*km/4) (4)))
    (Res
        (NDMap [(4, Par)]
          (Opaque "f_sor" [] (Prim Float)
          (Prim Float) (c_sor,p_sor)))
        (Var "v_s"
          (Vec (Vec Float (im*jm*km/4)) 4))),
  Assign
    (Var "v_out" (Vec (Prim Float) (im*jm*km)))
    (Res
      (NDMerge [4])
      (Var "v_m"
        (Vec (Vec Float (im*jm*km/4)) 4)))
  ] (Var "v_out" (Vec (Prim Float) (im*jm*km)))
```

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.

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.

Figure 6 illustrates our setup.

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 7). 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).

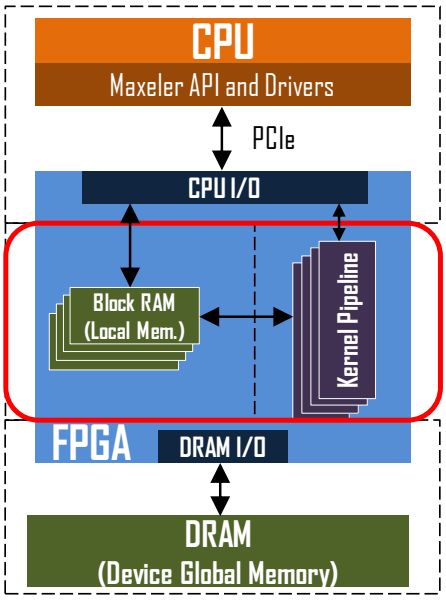

Figure 6. The Maxeler-TyTra integrated solution. The black dotted line identifies what is programmed using the Maxeler HLS tool. The solid/red line identifies the logic programmed with TyTra generated code. The overlap indicates that stream generation from on-chip Block-RAMs may be done by either in our flow.

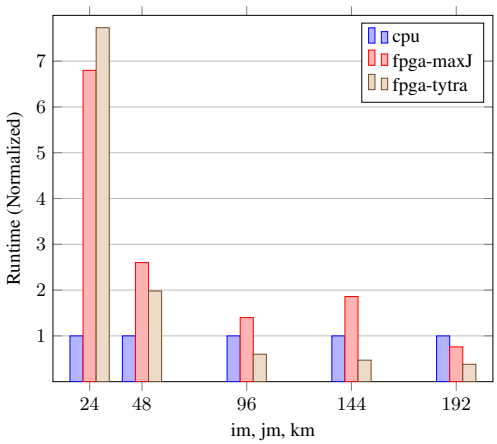

Figure 7. Runtime of the SOR kernel for different sizes of grid, normalized against the CPU-only solution. The figures are for 1000 iterations of the kernel.

The number of iterations of the SOR kernel per run, `nmaxp`, was fixed at at 1000. [6]

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 improve-

[6] 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.

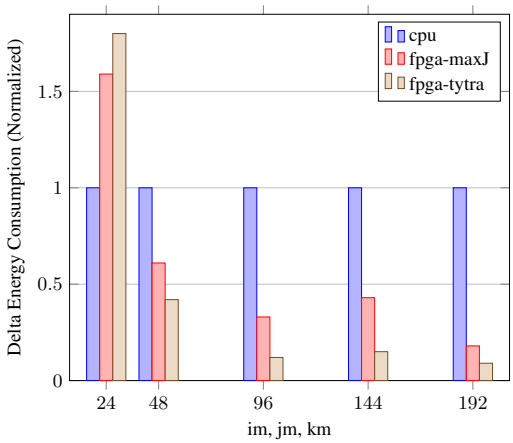

Figure 8. *Increase* from idle energy-consumption, for calculating the SOR kernel for different sizes of grid, normalized against the CPU-only solution. The figures are for 1000 iterations of the kernel.

ment 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.

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 8, 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 scientific kernels. While the typical route to implementation is the use of HLS tools like Maxeler or OpenCL, such tools may not necessarily fully expose the parallelism in the FPGA in a straightforward manner. 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 flow that, starting from high-level design entry in a functional 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-HLL language can be automatically derived from the type transformations, with guaranteed correctness. We have explained compilation from TyTra-HLL to TyTra-IR and the costing of the design 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 has a high abstraction design-entry 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.

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

References

The Xilinx SDAccel Development Environment, 2014. URL `http://www.xilinx.com/publications/prod_mktg/sdx/sdaccel-backgrounder.pdf`.

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 FPGAs*, FPGA '11, pages 33–36, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0554-9.

S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009. doi: 10.1109/IISWC.2009.5306797.

M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389 – 406, 2004. ISSN 0167-8191. doi: http://dx.doi.org/10.1016/j.parco.2003.12.002.

T. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. Singh. From opencl to high-performance hardware on FPGAs. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 531–534, Aug 2012. doi: 10.1109/FPL.2012.6339272.

B. da Silva, A. Braeken, E. H. D'Hollander, and A. Touhafi. Performance modeling for FPGAs: Extending the roofline model with high-level synthesis tools. *International Journal of Reconfigurable Computing*, 2013. doi: 10.1155/2013/428078.

M. Kaul, R. Vemuri, S. Govindarajan, and I. Ouaiss. 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, pages 616–622, New York, NY, USA, 1999. ACM. ISBN 1-58113-109-7. doi: 10.1145/309847.310010.

J. e. a. Keinert. 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, Jan. 2009. ISSN 1084-4309.

K. Keutzer, K. Ravindran, N. Satish, and Y. Jin. 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*, pages 273–278, Sept 2005. doi: 10.1145/1084834.1084903.

S. W. Nabi and W. Vanderbauwhede. Using type transformations to generate program variants for fpga design space exploration. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, Dec 2015. doi: 10.1109/ReConFig.2015.7393365.

S. W. Nabi and W. Vanderbauwhede. 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)*, pages 114–123, May 2016. doi: 10.1109/IPDPSW.2016.155.

H. Nakayama, T. Takemi, and H. Nagai. Large-eddy simulation of urban boundary-layer flows by generating turbulent inflows from mesoscale meteorological simulations. *Atmospheric Science Letters*, 13(3):180–186, 2012.

O. Pell and V. Averbukh. Maximum performance computing with dataflow engines. *Computing in Science Engineering*, 14(4):98–103, July 2012. ISSN 1521-9615. doi: 10.1109/MCSE.2012.78.

O. Segal, P. Colangelo, N. Nasiri, Z. Qian, and M. Margala. Sparkcl: A unified programming framework for accelerators on heterogeneous clusters. *CoRR*, abs/1505.01120, 2015.

D. B. Thomas, S. T. Fleming, G. A. Constantinides, and D. R. Ghica. Transparent linking of compiled software and synthesized hardware. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2015*, pages 1084–1089, March 2015.

Appendix 1: Type Transformations

In this appendix we provide the basic theorem governing the *Split* and *Merge* vector type transformations.

Theorem 1. *For any given pair of vector types* v_1 *and* $v_2 \in V(a,n)$ *there exists a type transformation* F *comprised of the operations* Split *and* Merge *so that* $F\, v_1 = v_2$.

Proof.

- Every valid m-dimensional partition of a 1-D vector type Vec $n\, a$ is an integer factorization $[k_1, ..., k_m]$ of n.
- To obtain this partition, we simply apply the composition of all *Split* k_i in order: *Split* k_1 *Split* k_2 ... *Split* k_m Vec $n\, a$ = Vec k_1 Vec k_2 ...Vec k_m a.
- Application of of *Merge* to any vector type results in a 1-D vector type: *Merge* Vec k_1 Vec k_2 ...Vec k_m a = Vec $\prod_{i=1}^{m} k_i\, a$ = Vec $n\, a$.
- Thus, to transform v_1 into v_2 it is sufficient apply *Merge* to v_1 and then apply the *Split* operations corresponding to the partition of v_2.

□

Appendix 2: Program Transformations

In this Appendix we provide the theorems governing how transforming a top-level type impacts on the functions *map*, *fold*, *zipt* and *unzipt*.

We will denote type-transformed functions and variables using a prime, e.g. *s'*.

Theorem 2 (Derivation of *map*-based programs)**.**

- *Given the following program:*

$$\begin{aligned} &map\ f\ s \\ &where\ s\ ::\ Vec\, n\, a_1\ ,\ f\ ::\ a_1 \rightarrow a_2 \end{aligned}$$

- *The type of* s' *is given by the transformation of the type of* s*:*

$$s'\ ::\ F\, Vec\, n\, a_1$$

- *The transformed program derived from this type is given by:*

$$map'\ f\ s'$$

- *Then the following propositions hold:*
  1. F = Split ⇒ *map'* $=$ *map map*
  2. F = Merge ⇒ *map* $=$ *map' map'*

Theorem 3 (Derivation of *fold*-based programs)**.**

- *Given the following program:*

$$\begin{aligned} &fold\ f\ x\ s \\ &where\ s\ ::\ Vec\, n\, a_1\ ,\ x\ ::\ b\ ,\ f\ ::\ a_2 \rightarrow a_1 \rightarrow a_2 \end{aligned}$$

- *The type of* s' *is given by the transformation of the type of* s*:*

$$s'\ ::\ F\, Vec\, n\, a_1$$

- *The transformed program derived from this type is given by:*

$$fold'\ f\ x\ s'$$

- *Then the following propositions hold:*
  1. F = Split ⇒ *fold'* $=$ *fold fold*
  2. F = Merge ⇒ *fold* $=$ *fold' fold'*

Theorem 4 (Derivation of *zipt*-based programs)**.**

- *Given the following program:*

$$\begin{aligned} &zipt\ v \\ &where\ v\ ::\ (Vec\, n\, a_1\ ,\ Vec\, n\, a_2) \end{aligned}$$

- *The type of* v' *is given by the transformation of the type of* v*:*

$$v'\ ::\ (F\, Vec\, n\, a_1\ ,\ F\, Vec\, n\, a_2)$$

- *The transformed program derived from this type is given by:*

$$zipt'\ v'$$

- *Then the following propositions hold:*
  1. F = Split ⇒ *zipt'* $=$ *(map zipt) zipt*
  2. F = Merge ⇒ *zipt* $=$ *(map zipt') zipt'*

Theorem 5 (Derivation of *unzipt*-based programs)**.**

- *Given the following program:*

$$\begin{aligned} &unzipt\ v \\ &where\ v\ ::\ (Vec\, n\, (a_1\ ,\ a_2) \end{aligned}$$

- *The type of* v' *is given by the transformation of the type of* v*:*

$$v'\ ::\ (F\, Vec\, n\, (a_1\ ,\ a_2)$$

- *The transformed program derived from this type is given by:*

$$unzipt'\ v'$$

- *Then the following propositions hold:*
  1. F = Split ⇒ *unzipt'* $=$ *zipt (map zipt)*
  2. F = Merge ⇒ *unzipt* $=$ *zipt' (map zipt')*

The proofs for these theorems are very similar and quite straightforward.

- Proposition 1. is proven using direct inference through substitution on the type signatures.
- Proposition 2. is actually a correctness condition, with an associated lemma for each case that is straightforward to prove. The key observation is that for Proposition 2. to hold, the type of the input vector must be a nested vector type. But as the types of the opaque functions are per definition atomic, this implies a previous *Split* transformation.