Nabi, S. W., and Vanderbauwhede, W. Using Type Transformations to Generate Program Variants for FPGA Design Space Exploration. In: 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig), Mexico City, Mexico, 7-9 Dec 2015, pp. 1-6. ISBN 9781467394055.

There may be differences between this version and the published version. You are advised to consult the publisher’s version if you wish to cite from it.

http://eprints.gla.ac.uk/117319/

Deposited on: 09 March 2016
Abstract—We present preliminary results with the TyTra design flow. Our aim is to create a parallelising compiler for high-performance scientific code on heterogeneous platforms, with a focus on Field-Programmable Gate Arrays (FPGAs). Using the functional language Idris, we show how this programming paradigm facilitates generation of different correct-by-construction program variants through type transformations. We have developed a custom Intermediate Representation (IR) language, the TyTra-IR, which is similar to the LLVM IR, with extensions to express parallelism, allowing us to design variants associated with each program variant. The key innovation of the TyTra-IR is the ability to construct and cost design variants for FPGAs. Our prototype compiler generates Verilog code for FPGA synthesis from a given IR description. Using a real-world Successive Over-Relaxation (SOR) kernel, we illustrate generation of program variants in Idris, their representation in TyTra-IR, and evaluation of variants using our cost-model. We compare the estimates from the cost-model with results from synthesis and simulation of equivalent HDL.

I. INTRODUCTION

Platforms for High-Performance Computing (HPC) are becoming increasingly heterogeneous with the adoption of GPUs, many-core accelerators and FPGAs. Such platforms present a significant programming challenge, especially because the key users of HPC resources are scientists, not parallel programmers. The work we present in this paper aims to facilitate the use of Field-Programmable Gate Arrays (FPGAs). These devices are very promising in terms of energy efficiency, but programming them presents a major obstacle to their wider adoption in HPC. High-level programming tools for FPGAs have made significant contributions, but require considerable effort to find the best design variant from the complex design space offered by the fine-grained reconfigurability of FPGAs.

Our main contribution is raising the programming abstraction for FPGAs such that we can express the design in a functional language like Idris as shown in Figure 1. This functional abstraction enables type transformation that reshape the data to create a variants that are correct-by-construction. The transformation implies a reconfigured FPGA architecture for that reshaped data, effectively creating a new design variant. A light-weight cost-model for evaluation of multiple design variants opens the route to a fully automated design flow. Our aim is to create a parallelising compiler for FPGAs. Our prototype compiler generates Verilog code for FPGA synthesis from a given IR description. Using a real-world Successive Over-Relaxation (SOR) kernel, we illustrate generation of program variants in Idris, their representation in TyTra-IR, and evaluation of variants using our cost-model. We compare the estimates from the cost-model with results from synthesis and simulation of equivalent HDL.

Fig. 1. Design entry in the TyTra flow is in a functional language like Idris, variants are generated using type-transformations and converted to the TyTra-IR. The back-end compiler costs the variants and emits HDL code, which can be integrated with an HLS tool for working solution. The dotted line marks the stages in the flow that are currently automated.

II. PROGRAM GENERATION THROUGH TYPE TRANSFORMATIONS

We aim to demonstrate how a program can be rewritten in a high-level language that facilitates generation of differ-
ent, correct-by-construction instances of that program through type-transformations. Each program variant will have a different performance related to its degree and type of parallelism, and a different cost. Through our cost model we are able to select the best suited instance in a guided optimisation search.

A. Exemplar: Successive Over-Relaxation

Consider the following SOR kernel, taken verbatim from the code for the Large Eddy Simulator, an experimental weather simulator [1] written in Fortran.

```fortran
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)+cn3s(i)+p(i-1,j,k)) &
   + cn3l(j)+p(i,j+1,k)+cn4s(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
```

The kernel iteratively solves the Poisson equation for the pressure assuming periodic boundary conditions in the transverse (i) direction and open (Neumann) conditions in other directions. The main computations are a stencil over the neighbouring cells (which is inherently parallel), and a reduction to compute the remaining error. The boundary conditions are simple copy operations. As the boundary conditions do not involve any computation, they are not discussed in what follows. The SOR algorithm is iterative, when the error is smaller than a preset value or when a maximum number of steps is reached, the algorithm stops.

B. Idris, Higher-Order Functions map and fold, and Dependent Types

Functional languages can express higher-order functions i.e. functions that take functions as arguments and can return functions. They support partial application of a function, and have strong type safety. These features make them suitable as a high-level design-entry point, and for generating correct-by-construction or safe program variants through type transformations. We have chosen the Idris language [2] because it is dependently typed, which allows the size of the data to be expressed explicitly in the type. This feature is crucial for our purpose of generating program variants by reshaping data and ensuring correctness through type safety.

To describe computations on finite-size ordered sets of data we use a dependent vector type which encodes the size of the vector:

```idris
p1D : Vect (ip+3)*(jp+3)*(kp+2) Float
```

Here the type of p1D is the entire string after : , showing it is a vector of size equal to size of the 3D matrix, and of type Float. Multi-dimensional vectors are obtained through nesting:

```idris
p2D : Vect (jp+3)*(kp+2) (Vect ip+3 Float)
```

The main higher order functions we will use are map and fold, which capture the computation pattern at a higher abstraction than the more familiar for loops. The map operation simply applies a function to a vector, similar to a dependency-free loop. We can write its type as:

```idris
map : (t1 -> t2) -> (Vect sz t1) -> (Vect sz t2)
```

The foldl operation (which performs reduction) applies a function to a vector and an accumulator, similar to a for-loop that performs an accumulation. We can write its type similarly as

```idris
foldl : (t2 -> t1 -> t2) -> t2 -> (Vect sz t1) -> t2
```

These higher order functions allow us to transform programs through transformation of the types, as discussed in §II-D. Although expressing programs in terms of map and fold might seem restrictive, the paradigm is in fact expressive enough to express most scientific codes and is a very good starting point for stream-based FPGA programming.

C. SOR Kernel in Idris

The functions map and fold perform computations on the vector without explicit iterators. Therefore, to calculate e.g. the SOR expression as above, we need to define a set of vectors, one for every term in the expression. This means that we need a function that will take the original vectors p, rhs, cn* and return a new single vector of size im.jm.km, where each elements is a tuple consisting of all terms required to compute the SOR, i.e. the pressure at a given point an its 6 neighbouring cardinal points, the weight coefficients cn* and the rhs term for a given point. The implementation of this function is simply a copy operation and not our main concern.

```idris
pps = prepare_vectors p rhs cn1 cn2l ...
```

Given the new vector of tuples, we can define the actual SOR computation as

```idris
ps = map p_sor pps
```

where p_sor computes the new value for the pressure for a given input tuple from pps:

```idris
p_sor pt = reltmp + p_c
```

where

```idris
  (p1, p2, ..., p_c, rhs_c) = pt
  reltmp = omega * (cn1 * {
    cn2l_x * p_i_p1 + cn2s_x * p_i_m1 +
    cn3l_x * p_j_p1 + cn3s_x * p_j_m1 +
    cn4l_x * p_k_p1 + cn4s_x * p_k_m1
  } - rhs_c) - p_c
```

D. Type Transformations

Our main purpose is to generate many safe variants by transforming the type of the various functions making up the program and inferring the program transformations from the type transformation. The details and proofs of the type transformations and their safety are available in [3]. In brief, we reshape the vector in an order-preserving manner and infer the corresponding program that produces the same result. Each reshaped vector in a program variant translates to a different

---

1 Note that in Idris, arguments in a type signature or a function call are separated by a space.

2 Function types are specified by the type of each argument, separated by the arrow ->, with the last argument being the return type.
arrangement of streams. We then use our cost-model to choose the best design, as we will discuss in §V-A.

Let’s assume that the type of the 1D prepared vector is \( t \) and its size \( im \cdot jm \cdot km \), which we can turn into e.g. a 2D vector with sizes \( im \cdot jm \) and \( km \) using type-transformation:

\[
\begin{align*}
\text{pps} & : \text{Vect} \ (im \cdot jm \cdot km) \ t \quad --\text{1D vector} \\
\text{ppst} & : \text{Vect} \ km \ (\text{Vect} \ im \cdot jm \ t) \quad --\text{transformed 2D vector}
\end{align*}
\]

Resulting in a corresponding change in the program:

\[
\begin{align*}
\text{ps} & = \text{map} \ p_{\text{SOR}} \ \text{pps} \quad --\text{original program} \\
\text{ppst} & = \text{reshapeTo} \ km \ \text{pps} \quad --\text{reshaping data} \\
\text{pst} & = \text{map} \ (\text{map} \ p_{\text{SOR}}) \ \text{ppst} \quad --\text{new program}
\end{align*}
\]

where \( \text{map} \ p_{\text{SOR}} \) is an example of partial application.

Because \( \text{ppst} \) is a vector of vectors, the outer map takes a vector and applies the function \( \text{map} \ p_{\text{SOR}} \) to this vector.

E. Parallelism Transformations

As a \( \text{map} \) is by definition a dependency-free operation, it can in principle be executed in parallel on all elements of the vector. It can of course also be executed sequentially, or as a stream. This means that the original program has three variants, one for each type of map, i.e. parallel, sequential and pipelined. A transformed program with two nested maps has nine variants. Each program transformed by \text{reshaping} the data will have these nine variants. The number of transformed program types is equal to the number of possible integer divisions of the original type. In this fashion it is possible to generate large numbers of safe program variants. In §IV we discuss the cost-model we developed to evaluate the variants. As part of our future work, we aim to limit the number of program variants generated and evaluated, by developing heuristics for constrained variant generation.

F. Reductions

The \( \text{map} \) higher-order function is limited to dependency-free vector operations. To express dependencies in a reduction, we can use the \text{foldl} higher-order function. In order to incorporate the computation for the global SOR error we change the return value of the function \( p_{\text{SOR}} \) to a tuple that contains two values, i.e. the pressure and the error from that iteration:

\[
\begin{align*}
\text{p_{SOR}} \ pt & = (\text{p_c+reltmp, reltmp*reltmp}) \ 	ext{where} ...
\end{align*}
\]

We apply \( \text{map} \) and separate out the arrays using \text{unzip} (which transforms a vector of tuples into a tuple of vectors); to obtain the cumulative \text{SOR error} we sum \text{rtsqs}:

\[
\begin{align*}
(\text{ps, rtsqs}) & = \text{unzip} \ (\text{map} \ p_{\text{SOR}} \ \text{pps}) \\
\text{SOR error} & = \text{sum} \ \text{rtsqs}
\end{align*}
\]

Here, \( \text{sum} = \text{foldl} (+) 0 \). If we transform the type of \( \text{pps} \) as before, the type of \( \text{rtsqs} \) will be transformed and hence the fold operation will be transformed as explained in [3]:

\[
\begin{align*}
\text{SOR error} & = \text{foldl} \ (\text{foldl} (+) 0) \ \text{rtsqs}
\end{align*}
\]
Our throughput performance measure is \( EWGT \) (Effective Work-Group Throughput), defined as the number of times an entire work-group (i.e., all work-items) executes the kernel every second. Following is the generic expression which applies to the entire design space and expressions for configurations of interest can be derived from it.

\[
EWGT = \frac{L \cdot D_V}{N_R \cdot \{T_R + N_I \cdot N_{To} \cdot T \cdot (P + I)\}}
\]

Where: \( EWGT \) = Effective Workgroup Throughput; \( L \) = Number of parallel lanes or threads on the FPGA; \( D_V \) = Degree of vectorization per lane; \( N_R \) = Number of FPGA configurations needed to execute the complete kernel for the entire work-group; \( T_R \) = Time taken to reconfigure FPGA; \( N_I \) = Number of instructions per Processing Element\(^3\); \( N_{To} \) = Ticks taken by one instruction; \( T \) = FPGA clock period; \( P \) = Pipeline depth; \( I \) = Number of work-items. This generic expression can be reduced based on the FPGA configuration. For the typical case of using an FPGA for HPC applications where the FPGA implements one or more pipeline lanes of the kernel, \( D_V \), \( N_R \) and \( N_I \) would all reduce to 1.

For estimating resource utilization, we observed that the regularity of FPGA fabric allows simple cost expressions for most instructions. These are then used by our compiler to evaluate overall costs for the design. A certain amount of uncertainty is introduced as our models do not take into account the optimizations done by the synthesis tools, but the estimate remain accurate enough to achieve the purpose of making design choices. The novelty here is that through a well-defined syntax at a low abstraction, the TyTra-IR exposes the parameters in the cost-model expressions, which can be extracted by our compiler. If we were to use a higher-level language as an internal IR to represent the design variants, a more thorough and time-consuming build would be required (as used by e.g. the Maxeler tool flow [8]), which is not suitable for comparing a large number of variants.

The code-generator creates a dataflow architecture on the FPGA, with a pipeline of primitives as well as customized functional units, and exposes ILP by scheduling operations in parallel where possible. Streams are automatically created that connect these one or more pipelines to on-chip memories data for the streams. Our preliminary work on the cost model and code-generator is limited to working with on-chip memories of the FPGA.

V. USING THE COMPILER ON THE SOR EXAMPLE

For proof-of-concept of our cost model and prototype back-end compiler, we hand-coded in the TyTra-IR some design variants of the SOR kernel as discussed in §II. Figure 4 shows the translation of the SOR kernel to TyTra-IR configured as a single pipeline. 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 referred to in §II-C. This tuple is fed into the datapath pipeline described in lines 10–15. Figure 5 shows the kernel’s realization as a pipeline as described by this code.

The same SOR example can be expressed in the IR to represent thread-parallelism by adding multiple lanes, corresponding to a reshaped data along 4 rows, as shown in Figure 6. This is one of the many possible variants generated in the high-level Idris code through type transformations, as described in §II-E.

A. Evaluating TyTra-IR Design Variants using the Cost-Model

Figure 7 shows evaluation of variants generated by reshaping the input streams – implying a different configuration on

\[^3\text{For pipelined cores on the FPGA, a PE is always configured for one instruction.}\]
Fig. 4. Abbreviated TyTra-IR code for the SOR kernel configured as a single pipeline lane.

Fig. 5. Illustration of the pipelined dataflow 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 the top refer to on-chip memory for each data.

the FPGA – and costing the corresponding IR description. If data is transported between the CPU and device every time a new call to SOR is made, then beyond 4 lanes, we encounter the host communication wall; any increase in number of lanes will not improve performance unless the communication-to-computation ratio decreases by doing more kernel iterations per invocation of SOR. Alternatively, if the all the data is made available in the device’s global memory, i.e. the DRAM on the device board, then the communication wall moves to about 16 lanes. We encounter the computation-wall when we cross six lanes, where we run out of LUTs on the FPGA. However, we can see other resources are hugely underutilized, and some sort of resource-balancing can lead to further performance improvement.

We can also evaluate the impact of re-using data read from the host into the device on-chip memory. This re-use effects the extent to which we are exploiting the hardware parallelism of the configuration on the FPGA. As shown in Host-IO series of Figure 8, if the SOR kernel is repeated less than 16 times, we are in the IO-bound zone, not fully utilizing the eight lanes in the design. Further increase in the repetition of kernel brings us into the computation-bound zone, where we can get better performance by optimizing the design to use lesser or more balanced resources, or possibly by moving part of the kernel to a peer device. This transition from IO to compute bound performance happens at a smaller lane-count IO is from the device DRAM with a much higher bandwidth.

We have illustrated here how the TyBEC estimator can be used to: evaluate many design variants and the trade-offs involved, generate feedback for optimizations, and achieve a near-optimal design point. We would like to highlight that the estimator is very light-weight, and e.g. the evaluation of the five design variants in Figure 7 takes a few seconds. It is orders-of-magnitude quicker than e.g. the Maxeler flow that takes tens of minutes to give preliminary resource estimates for one variant.
The authors acknowledge the support of the EPSRC for the TyTra project (EP/L00058X/1).

REFERENCES


