# Towards Automated Design Space Exploration and Code Generation using Type Transformations

Syed Waqar Nabi
School of Computing Science
University of Glasgow, UK
syed.nabi@glasgow.ac.uk

Wim Vanderbauwhede
School of Computing Science
University of Glasgow, UK
wim.vanderbauwhede@glasgow.ac.uk

## ABSTRACT

Heterogeneous High-Performance Computing (HPC) platforms present a significant programming challenge, especially because the key users of HPC resources are scientists, not parallel programmers. It is our view that compiler technology has to evolve to automatically create the best compiled program variant by transforming a given original program. We describe our novel methodology based on *type transformations* and cost models, with current focus on FPGAs. We generate *correct-by-construction* program variants from a baseline representation of the kernel in a functional language like Idris. The variants are evaluated through our cost model which gives an estimate of performance and resource utilization on a specific FPGA device. The cost-model is built on top of a new intermediate language, the TyTra-IR. We use a simple kernel to illustrate multiple configurations on an FPGA using the semantics of TyTra-IR. We show preliminary results using a real-world Successive Over-Relaxation (SOR) kernel. We demonstrate the generation of program variants in Idris, their representation in TyTra-IR, their evaluation using our cost-model, and results of code-generation for a selected variant.

## 1. INTRODUCTION

High Performance Computing (HPC) platforms are increasingly adopting heterogeneous computing devices like many-core CPUs and GPUs. FPGAs are increasingly being used as well, with languages like OpenCL developing to provide a unified programming framework. Porting legacy scientific code for good performance still requires considerable effort.

The purpose of the *TyTra* project is to develop a compiler that will produce high-performance executables for heterogeneous platforms from a single code-base. The work we present in this paper aims specifically to facilitate the use of Field-Programmable Gate Arrays (FPGAs). These devices are very promising in terms of energy efficiency, but the relative difficulty in programming them is a major obstacle to their wider adoption in HPC. High-level programming tools for FPGAs have made a significant contribution in this respect, but still require considerable human input in finding the best design point in the design space exposed by the fine-grained reconfigurability of FPGAs. Our proposed flow (Figure 1) raises the programming abstraction for FPGAs such that we express the design in a functional language like Idris [1] or Haskell. This functional abstraction enables *type transformation* that reshape the data creating a new program variant that is *correct-by-construction*. This transformation effectively creates a new *design* variant. A light-weight cost-model allows evaluation of multiple design variants, opening the route to a fully automated compiler that can generate variants, evaluate them, choose the best option, and generate HDL code for FPGA targets.

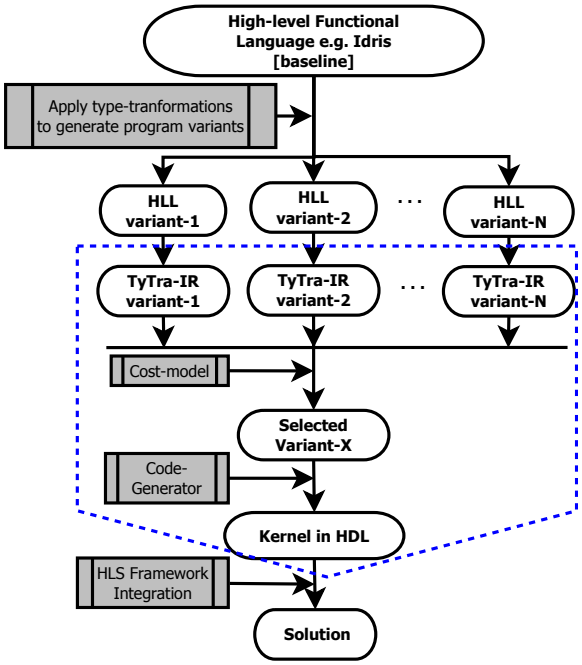


**Figure 1: The TyTra design flow. The dotted line marks stages that are currently automated.**

Our exemplar is a real kernel from the scientific computing domain, the successive over-relaxation (SOR) kernel used in a weather model [8] through which we illustrate generation of program variants in Idris, their representation in TyTra-IR, and evaluation of variants using our cost-model.

## 2. PROGRAM TRANSFORMATIONS

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

## 2.1 Exemplar: Successive Over-Relaxation

We consider a SOR kernel, taken from the code for the Large Eddy Simulator, an experimental weather simulator [8]. The kernel iteratively solves the Poisson equation for the pressure. The main computation is a stencil over the neighbouring cells (which is inherently parallel), and a reduction to compute the remaining error (not shown).

We express the algorithm in a functional language (Idris) using higher order functions to describe array operations. The baseline implementation will be:

```
ps = map  p_sor  pps
p_sor pt = reltmp + p_c
where
  (p_i_p1,...,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
```

where *pps* is a function that will take the original input vectors *p, rhs, cn** and return a single new vector of size *im.jm.km*, where each elements is a tuple consisting of all terms required to compute the SOR, including its 6 neighbouring cardinal points. *p_sor* computes the new value for the pressure for a given input tuple from *pps*.

Our main purpose is to generate variants by transforming the *type* of the functions making up the program and *inferring* the program transformations from the type transformation. The details and proofs of the type transformations are available in [13]. 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 variant translates to a different arrangement of streams. We then use our cost-model to choose the best design.

As an illustration, assume that the type of the 1D-vector is *t* and its size *im.jm.km*, which we can *transform* into e.g. a 2-D vector with sizes *im.jm* and *km*:

```
pps : Vect (im*jm*km) t      --1D vector
ppst: Vect km (Vect im*jm t) --transformed 2D vector
```

Resulting in a corresponding change in the program:

```
ps = map p_sor pps          --original program

ppst= reshapeTo km pps      --reshaping data
pst = map (map p_sor) ppst --new program
```

where *map p_sor* is an example of partial application. Because *ppst* is a vector of vectors, the outer map takes a vector and applies the function *map p_sor* to this vector. This transformation results in a reshaping of the original streams into parallel lanes of streams, implying a configuration of parallel kernel pipelines in the FPGA.

# 3. PLATFORM MODEL

The Tytra-FPGA platform model is similar to the platform model introduced by OpenCL [11], but also informed by our prior work on the MORA FPGA programming framework [3], and more nuanced than OpenCL's to incorporate FPGA-specific architectural features; Altera-OpenCL takes a similar approach [4]. The main departure from the OpenCL model is the *Core* block, and the *Compute-Cores*. Figure 2 is a block diagram of the model.

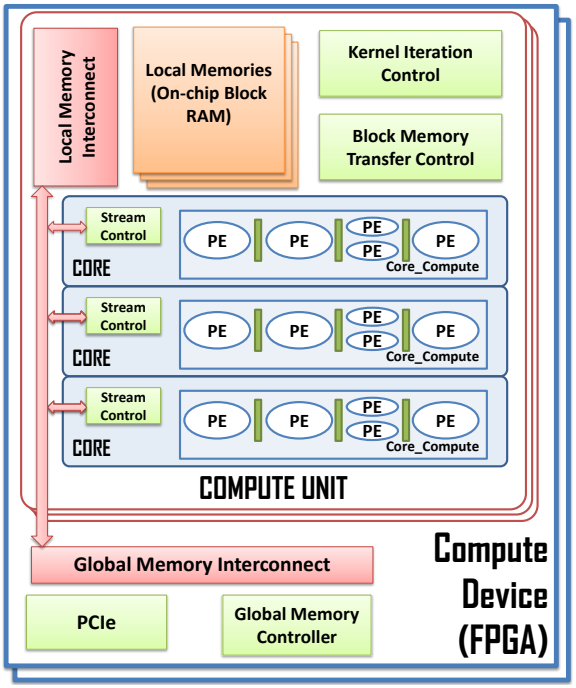


**Figure 2: The TyTra-FPGA Platform Model. The *compute-unit* is the unit of execution for a kernel. The *core* is the custom datapath unit created for a kernel, and may be considered equivalent to a pipeline *lane*, which may be replicated for thread-parallelism as shown here. It has logic for stream-control, and the inner *core-compute* is a pure dataflow block dealing exclusively with streams.**

# 4. DESIGN-SPACE ABSTRACTION

The available intrinsic parallelism in a kernel can be exposed by different configurations on an FPGA due to its fine-grained flexibility. Defining a *design-space* and a cost model to evaluate design points on it allows us to have a structured approach for mapping a kernel to a suitable configuration on the FPGA.

Our design-space abstraction (Figure 3) exposes the key differentiating feature of concern – the type and extent of parallelism. A `C2` configuration is a pipelined implementation of the kernel on the FPGA, its latency indicated along the x-axis. The other horizontal axis indicates the degree of parallelism achieved by creating multiple lanes of the pipeline. A configuration in the xy-plane (`C1`) will have multiple threads of execution, each of which will have pipeline parallelism as well. We expect this to be the preferable route for most small to medium sized kernels, and this is the focus of our prototype compiler. Note that we have not explicitly shown

the most fine-grained parallelism, i.e., Instruction-Level Parallelism (ILP). The assumption is that it will be exploited wherever possible in the pipeline, and our back-end compiler automatically extracts this parallelism. The design-space also encompasses resource re-use for cases where a kernel may have too many instructions to fit entirely on the available FPGA resources as a pipeline. Various configuration options for such situations are shown along the vertical axis, but these are outside the scope of our prototype compiler.

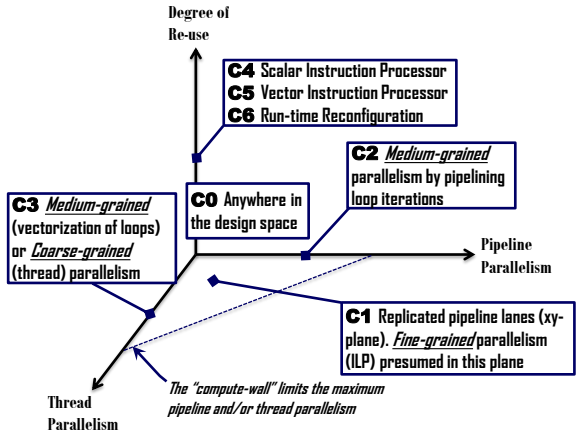


**Figure 3: The TyTra-FPGA Design Space Abstraction**

Various parameters will be relevant when making a selection from the design variants. In the context of HPC, we want to get the best performance (throughput) while staying within the FPGA resource and IO-bandwidth constraints. Also relevant would be *accuracy* as in FPGAs we can use non-standard, custom number representations and arithmetic operations.

# 5. A NEW INTERMEDIATE LANGUAGE

Our decision to use a new IR is a fundamental part of our proposed compilation flow. The development of the IR was guided by some specific requirements that arose out of its intended scope of use:

1. Should be intrinsically expressive enough to explore the entire design space of an FPGA (Figure 3), but with a particular focus on custom pipelines (The `C1` plane) because our prime target is HPC applications[12].
2. Should make a convenient target for a front-end compiler that would emit IR for each program variant (See Figure 1).
3. Should be able to express access operations in the entire communication hierarchy of the target device[1].
4. Should allow custom number representations to fully utilize the flexibility of FPGAs.
5. There should be a clear route to code generation of synthesizeable HDL code from the IR.
6. A core requirement is to have a light-weight cost-model for high-level estimates. We should be able to cost each configuration of interest in the design space.

[1]We have omitted the details in this paper, but the TyTra memory-model extends that of LLVM.

The above requirements necessitated the development of a custom intermediate language, as none of the existing HLS ("C-to-gates") tools meets all requirements. High-level FPGA programming languages like OpenCL or MaxJ[10] are designed to be programmer-friendly. They have coarse-grained, high-level datapath and control instructions and syntactic sugar, inappropriate as compiler targets. Moreover, even parallelism friendly high-level languages tend to be constrained to specific types of parallelism, and exploring the entire FPGA design-space would either be impossible, or protracted. The requirements of a lightweight cost-model also motivated us to work on a new language, which led to the TyTra-IR (TIR).

The TIR is a strongly and statically typed language, and all computations are expressed using Static Single Assignments (SSA). It is largely based on the LLVM-IR because it gives us a suitable point of departure for designing our language, where we can re-use the syntax of the LLVM-IR with little or no modification, and it will allow to explore LLVM optimizations to improve the code generation capabilities of our tool, as e.g. the LegUp [2] tool does. We use LLVM metadata syntax for describing FPGA-specific architectural features.

The TIR code for a design has two components:

**Manage-IR** deals with setting up the streaming data ports for the kernel. It corresponds to the logic in the *core* outside the *core-compute* (See Figure 2). All Manage-IR statements are wrapped inside the `launch()` method.

**Compute-IR** describes the datapath logic that maps to the core-compute unit inside the core. It mostly works with very limited data abstractions, namely, streaming and scalar ports. All Compute-IR statements are in the scope of the `main()` function or other functions "called" from it.

This division clearly separates the pure dataflow architecture — working with streaming variables and arithmetic datapath units — from the memory control and peripheral logic.

# 6. ILLUSTRATION OF IR USE

We use a trivial example and build various configurations for it, to demonstrate the expressiveness of the TIR for FPGAs. The following Fortran loop describes the kernel:

```
do n = 1,ntot
  y(n) =  K + ( (a(n)+b(n)) * (c(n)+c(n)) )
end do
```

## Sequential Processing

The baseline configuration, whose abbreviated TIR code is showed in Figure 4, is simply a sequential processing of all

```
1  ;****** Manage-IR ******
2  define void launch() {
3  @mem_a = addrspace(3) <NTOT x ui18>, ...
4  @strobj_a = addrspace(10),
5    !"source" , !"@mem_a", ...
6  @...[other memory and stream objects]
7  call @main()          }
8  ;****** Compute-IR ******
9  @main.a = addrSpace(12) ui18,
10   !"istream", !"CONT", !0, !"strobj_a"
11 @...[other ports]
12 define void @f1 (...args...) seq {
13   ui18 %1 = add ui18 %a, %b
14   ui18 %2 = add ui18 %c, %c
15   ui18 %3 = mul ui18 %1, %2
16   ui18 %y = add ui18 %3, @k       }
17 define void @main () {
18   call @f1(...args...) seq }
```

**Figure 4: TyTra-IR code for a sequential processing configuration of a simple kernel**

the operations in the loop. This corresponds to C4 configuration in Figure 3.

The manage-IR consists of the launch method which sets up the *memory-objects*, which are abstractions for any object that can be the source or destination of streaming data. In this case, the memory object (line 3) is a local-memory instance, indicated by the argument to *addrspace* qualifier. By changing the value of this qualifier, we can create memory-objects at different levels of the memory-hierarchy. The stream-objects connect to memory-objects to create streams of data, as shown in lines 4–5. The creation of streams from memory is equivalent to reading from an array in a loop, so the loop over work-items in Fortran disappears in the TIR. After setting up all stream and memory objects, the main function is called. Manage-IR code from further examples has been redacted, but looks very similar to what we have described here.

The compute-IR sets up the ports (lines 9-11), which are mapped to a stream-object, creating data streams for the compute-IR functions. The SSA datapath instructions in function *f1* are configured for sequential execution on the FPGA, indicated by the keyword *seq*.

## Single Kernel Execution Pipeline

This *C2* configuration is a fully pipelined version of the kernel, and the TIR code is shown in Figure 5.

```
1  @main.a = addrSpace(12) ui18,
2            !"istream", !"CONT", !0, !"strobj_a"
3  @...[other ports]
4  define void @f1 ( ...args...) pipe {
5    ui18 %1 = add ui18 %a, %b
6    ui18 %2 = add ui18 %c, %c
7    ui18 %3 = mul ui18 %1, %2
8    ui18 %y = add ui18 %3, @k }
9  define void @main () {
10   call @f1(...args...) pipe }
```

**Figure 5: TyTra-IR code for a pipelined configuration of a simple kernel.**

Note that the available ILP (the two add operations can be done in parallel) is detected by the compiler and automatically incorporated in the generated pipeline, without any need to express it in the IR explicitly. See Figure 6 for the block diagram of this configuration.

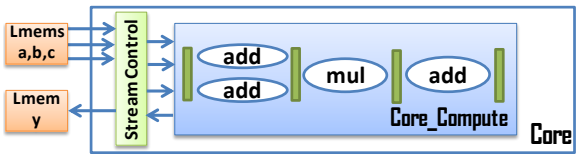


**Figure 6: Single Pipeline with ILP (C2 configuration) corresponding to IR code in Figure 5**

## Multiple Kernel Execution Pipelines

For a lot of kernels that aren't very complex, there may be enough resources on an FPGA to instantiate multiple identical pipeline *lanes*, i.e. a *C1* configuration. The code in Figure 7 illustrates how this can be specified in TIR. See Figure 8 for the block diagram of this configuration.

```
1  @main.a_01 = ...
2  @main.a_02 = ...
3  @...[other ports]
4  define void @f1 ( ...args...) pipe {...}
5  define void @f2 (...args...) par {
6    call @f1(...args...) pipe
7    call @f1(...args...) pipe
8    call @f1(...args...) pipe
9    call @f1(...args...) pipe }
10 define void @main () {
11   call @f2(...args...) par }
```

**Figure 7: TyTra-IR code for replicated pipeline *C1* configuration of a simple kernel**

Comparing with the previous single-pipeline configuration, note that we have a new *par* function *f2* calling the same *pipe* function four times, indicating replication. Similarly, there are now four separate ports for each array input, and there are four separate streaming objects for each of these ports (not shown), all of which connect to the same memory object, indicating a multi-port memory.

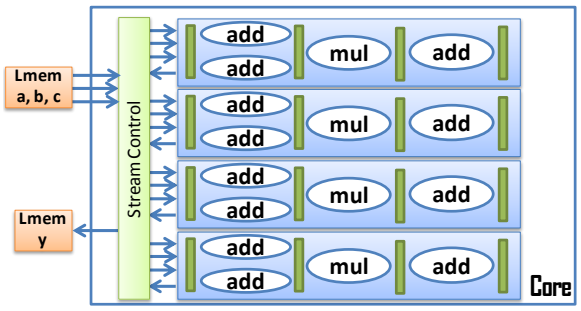


**Figure 8: Multiple Pipelines Lanes (C1 configuration) corresponding to IR code in Figure 7**

## Multiple Sequential Processing Elements - Vector Processing

There is one more useful configuration we can express in TIR by wrapping multiple calls to a `seq` function in a `par` function. This would represent a vectorized sequential processor (C5). We have omitted the IR and figure for this configuration in this paper.

# 7. THE TYTRA BACK-END COMPILER

As shown in Figure 1, we have already developed a back-end compiler that accepts a design-variant in TIR, costs it, and if needed, generates the HDL code for it. These steps are shown in more detail in Figure 9.

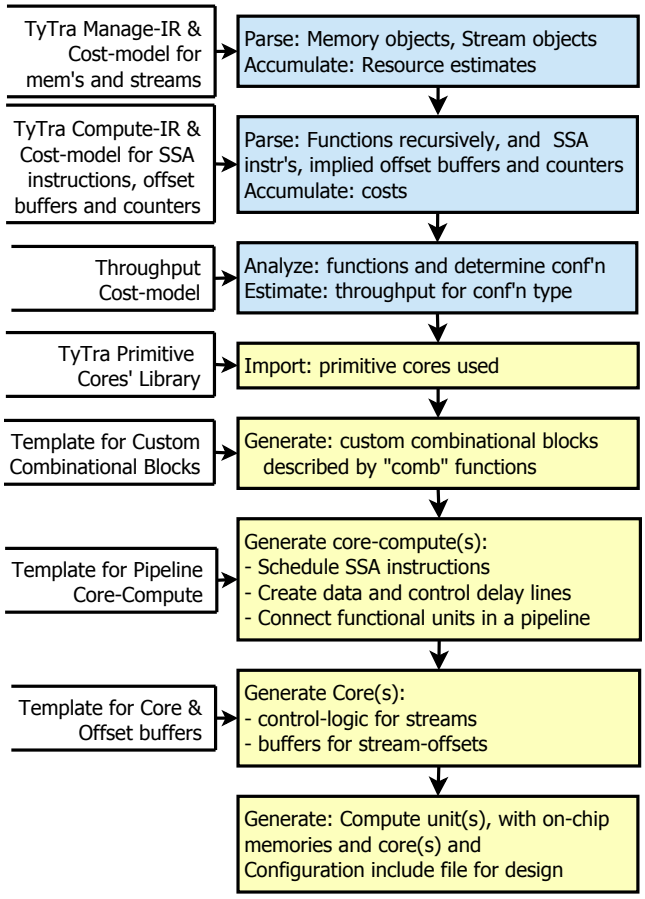


**Figure 9: The TyTra back-end compiler flow, showing the estimation flow (blue/first three stages) and code generation flow (yellow). The starting point for this subset of the entire flow (Figure 1) is TyTra-IR description representing a particular design variant, ending in the generation of synthesizeable HDL which can then be integrated with a HLS framework.**

## 7.1 The Cost-Model

We have designed the TIR specifically to allow generation of estimates of reasonable accuracy that allow us to judge the trade-offs of various design variants. The TyTra Back-End Compiler (TyBEC) can calculate estimates directly from the IR without any further synthesis. The estimates calculated by the TyBEC estimator are: the resource utilization for a specific Altera FPGA device (ALUTs, REGs, Block-RAM, DSPs), and the throughput estimate for the kernel under consideration.

### *7.1.1 Estimating Throughput*

We have described a performance measure called the *EWGT (Effective Work-Group Throughput)* defined as the number of times an entire work-group (the loop over entire index-space/work-items) of a kernel is executed every second. Measuring throughput at this coarse granularity allows us take into account parameters like dynamic reconfiguration penalty, and data transfer costs between successive kernel iterations. Following is the generic expression which applies to the entire design space (i.e. the `C0` root configuration), and expressions for configurations of interest can be derived from it.

$$EWGT = \frac{L.D_V}{N_R.\{T_R + N_I.N_{to}.T.(P+I)\}}$$

Where:

$EWGT$ = Effective Workgroup Throughput; $L$ = Number of identical *lanes*; $D_{V=}$ Degree of vectorization; $N_R$ = Number of FPGA configurations needed to execute the entire kernel; $T_R$ = Time taken to reconfigure FPGA; $N_I$ = Number of equivalent primitive operations per PE; $N_{TO}$ = Ticks taken by one primitive operation; $T$ = FPGA clock period; $P$ = Pipeline depth; $I$ = Total number of work-items. The expression is currently limited to on-chip memories and a more sophisticated cost expression incorporating DRAM access is being developed.

The key novelty here is that through its constrained syntax at a particular abstraction, the TIR *exposes* the parameters that make up the expression, and a simple parser can extract them from the TIR code. If we were to use a higher-abstraction HLS language as our front-end compiler target, we would not be able to use such an expression, and a more thorough and time-consuming synthesis would be required (as used by e.g. the Maxeler tool flow [9]).

### *7.1.2 Estimating Utilization of FPGA Resources*

Each instruction in the IR can be assigned a cost by either using a simple analytical expression or looking up a cost database for the specific IR instruction and data type. Both the analytical expressions and the cost-database are device-specific, which we create after doing a series of simple synthesis experiments on the device.

Our observation is that the regularity of FPGA fabric allows some very simple first or second order expressions to be built up for most instructions based on a few experiments. As an example, consider the trend-line for LUT requirements against bit-width for integer division shown in Figure 10. It was generated from three-data points (18, 32 and 64 bits) from synthesis experiment for a Stratix-V device. We can now use it for polynomial interpolation, e.g., for 24-bits, and get an estimate of 654 ALUTs, which compares favourably with the actual usage of 652 ALUTs.

Another example is shown in Figure 11, which illustrates how we derive the estimates for integer multiplication. A multiplier requires two different kinds of resources: DSP-elements and ALUTs. As we can see, both show a very different kind of trend, with the DSP-elements behaving like a step-function, and ALUTs showing a continuous trend. Either way, once we have the experimental results, incorporat-

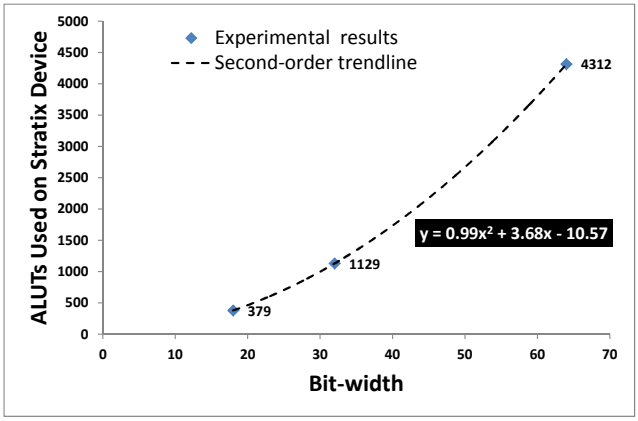


**Figure 10: Deriving analytical expression for used ALUTs for unsigned integer division on a Stratix-V device.**

ing the derived analytical expressions into the cost-model is straightforward. Other IR instructions have similar or simpler expressions that we can use to calculate their resource-utilization estimate. We thus calculate the overall resource-cost of the design by accumulating the cost of individual IR instructions and the structural information implied in the type of each IR function.

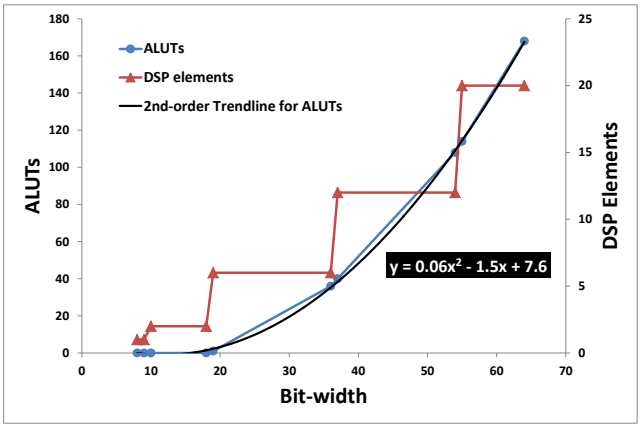


**Figure 11: Deriving analytical expression for utilization of ALUTs and DSP-elements for unsigned integer multiplication on a Stratix-V device.**

We thus calculate the overall resource-cost of the design by accumulating the cost of individual IR instructions and the structural information implied in the type of each IR function.

## 7.2 HDL Code Generation

We have developed a Verilog HDL code generator as part of the compiler, which can currently generate configurations in the C1 plane, that is, one or multiple identical pipelines for a kernel. Our code-generator creates designs that work with on-chip memories. For off-chip memory access and host communication, we integrate the generated code with a commercially available HLS framework by Maxeler, which has the ability to integrate custom HDL code as part of its high-level programming flow.

# 8. USING TYBEC ON THE SOR EXAMPLE

For proof-of-concept of our cost model and prototype compiler we hand-coded some design variants of the SOR kernel as discussed in §2.1. Figure 13 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 §2.1. This tuple is fed into the datapath pipeline described in lines `10-15`. Figure 12 shows the kernel's realization as a pipeline. The same SOR example can be expressed in the

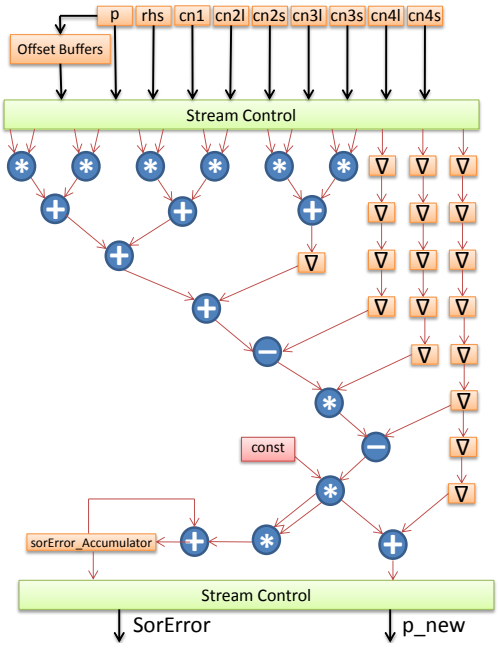


**Figure 12: Illustration of the pipelined dataflow of the SOR kernel generated by our compiler. Only pass-through pipeline buffers are shown; all functional units have pipeline buffers as well. The blocks at the top refer to on-chip memory for each data.**

IR to represent *thread-parallelism* by adding multiple *lanes*, corresponding to a reshaped data along 4 rows, similar to the simple illustration we showed in Figure 7.

## 8.1 Evaluating TyTra-IR Design Variants using the Cost-Model

We use the high-level *reshapeTo* function in Idris 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 14 shows evaluation of variants generated by reshaping the input streams and costing the corresponding IR description.

For maximum performance, we would like as many lanes of execution as the resources on the FPGA allow, or un-

```
; **** 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, %cn21
  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 }
```

**Figure 13: Abbreviated TyTra-IR code for the SOR kernel configured as a single pipeline lane.**

til we saturate the IO bandwidth. If data is transported between the host and device, then beyond 4 lanes, we encounter the *host communication wall*; further replication of kernel-pipeline will not improve performance unless the communication-to-computation ratio decreases by having more kernel iterations per invocation of SOR. 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.

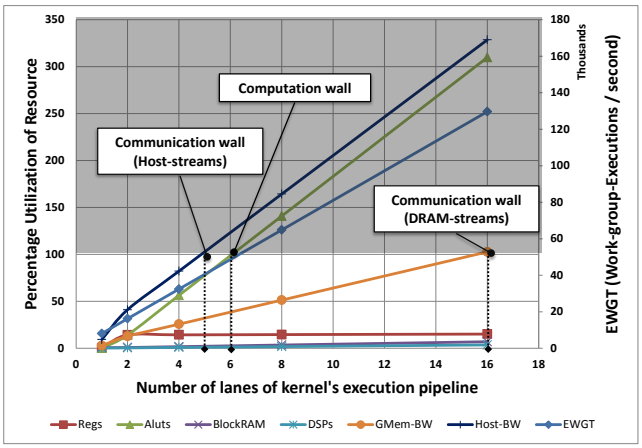


**Figure 14: Evaluation of variants for the SOR kernel generated by changing the number of kernel-pipelines (16 data points and 10 kernel iterations).**

The estimator can also be used to evaluate if we are fully utilizing the computation power that the design-configuration of the FPGA exposes, or if the overall performance has been limited by the IO bandwidth. As shown in *Host-IO* series of Figure 15, 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 or host. This transition from IO to compute bound performance comes much earlier in case the streams are connected to the global memory with a higher bandwidth.

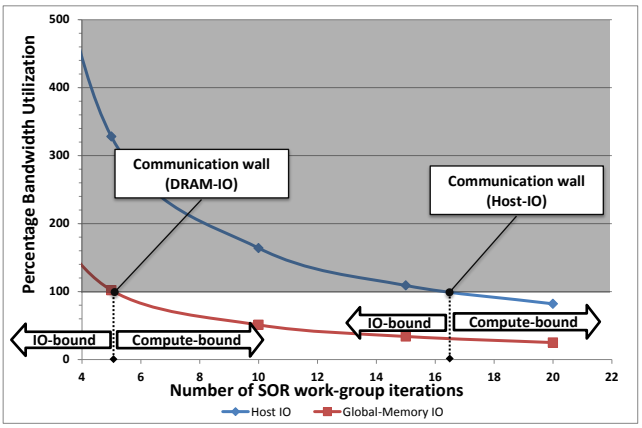


**Figure 15: Evaluating how the number of kernel repetitions (on the same data) effects whether performance is IO or compute bound, for both host and global-memory communication scenarios (lanes fixed at 8).**

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 here that the estimator is very light-weight, and e.g. the evaluation of the five design variants in Figure 14 takes a few seconds. That is orders-of-magnitude quicker than e.g. the Maxeler flow that takes tens of minutes to give preliminary resource estimates for one variant.

A light-weight cost-model is essential for our proposed approach of automatically evaluating many design variants as part of the TyTra compiler flow. Preliminary results show that there is a trade-off in the accuracy of our cost model, as shown in Table 1. The difference in the throughput (EWGT) estimate is due to deviation in the underlying frequency estimate, but it can be seen that the cycles/kernel estimate is much more accurate. 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.

## 9. RELATED 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 this high-level code 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 (mostly C-like) language, that nevertheless is not a pure software language and requires knowledge of target hardware and the programming framework [9, 4, 6], (2) compiler optimizations are limited to improving the overall architecture already specified by the programmer,

| Resource | 1-lane (E) | 1-lane (G) | 4-lane (E) | 4-lane (G) |
|---|---|---|---|---|
| ALUTs | 239 | 164 | 148$K$ | 146$K$ |
| REGs | 725 | 572 | 76,628 | 77,260 |
| BRAM(bits) | 186$K$ | 186$K$ | 449$K$ | 682$K$ |
| DSPs | 9 | 12 | 36 | 24 |
| Cycles/Kernel | 1,746 | 1,742 | 436 | 446 |
| EWGT | 190$K$ | 222$K$ | 763$K$ | 488$K$ |

**Table 1: Cost and throughput estimated from IR (E), compared with results from generated (G) Verilog code synthesized for a Stratix-5 Device, for *1-lane* and *4-lane* variants of SOR kernel.**

with no real *architectural* design-space exploration[9, 4, 6, 2], (3) solutions are based on a creating soft-microprocessors on the FPGA and are hence not necessarily optimized for HPC [2, 7], (4) the exploration requires evaluation of variants that take a prohibitively long amount of time[6], or (5) the flow is limited to very specific application domain e.g. for image-processing or DSP applications[5]. A high-level, pure software design-entry in the functional paradigm, that can automatically and quickly perform architectural optimizations using a light-weight cost-model, and apply *safe* transformations to ensure correct solutions, is to the best of our knowledge a completely novel proposition.

# 10. CONCLUSION AND FUTURE WORK

We have shown our approach to automated exploration of the design space of FPGA, and generating HDL code for programming the chosen design variant. Starting with a high-level functional language *Idris*, we demonstrated our method of creating program variants through the use of higher-order functions and type-transformations. These program variants map to design variants for the FPGA, expressed in our custom IR language, the TyTra-IR. It not only allows us to describe various design variants for the same problem, but also to directly associate each variant with a cost for cost-driven optimization. Using a Succesive Over-Relaxation kernel taken from a real-world weather simulator as a running example, we illustrated the generation of program variants using type-transformations in Idris, expressing design variants in the TyTra-IR, and costing different variants to evaluate trade-offs. We demonstrated the accuracy of the cost-model by comparing against synthesis figures. In future, the TyTra-IR will evolve to support different front-end languages, with the aim to support legacy scientific code. We will explore routes from Fortran and from a modern scientific language such as Julia, either via Idris or via LLVM. The compiler will also be extended to incorporate optimizations from the LLVM framework before emitting HDL code. Finally, we will experiment with a much wider selection of benchmark kernels beyond this SOR example, for a better qualification of this approach.

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

# 11. REFERENCES

[1] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 2013.

[2] 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.

[3] S. R. Chalamalasetti, S. Purohit, M. Margala, and W. Vanderbauwhede. Mora-an architecture and programming model for a resource efficient coarse grained reconfigurable processor. In *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, pages 389–396. IEEE, 2009.

[4] 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.

[5] 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.

[6] J. Keinert, M. Streub&uhorbar;hr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith. 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.

[7] 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.

[8] C.-H. Moeng. A large-eddy-simulation model for the study of planetary boundary-layer turbulence. *J. Atmos. Sci.*, 41:2052–2062, 1984.

[9] O. Pell and V. Averbukh. Maximum performance computing with dataflow engines. *Computing in Science Engineering*, 14(4):98–103, July 2012.

[10] A. Putnam, D. Bennett, E. Dellinger, J. Mason, P. Sundararajan, and S. Eggers. Chimps: A c-level compilation flow for hybrid cpu-fpga architectures. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 173–178, Sept 2008.

[11] J. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, May 2010.

[12] W. Vanderbauwhede. On the capability and achievable performance of fpgas for hpc applications. In *Emerging Technologies Conference*, Apr 2014.

[13] W. Vanderbauwhede. Inferring Program Transformations from Type Transformations for Partitioning of Ordered Sets, 2015.