
This is the author’s final accepted version.

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/117338/

Deposited on: 28 April 2017

Enlighten – Research publications by members of the University of Glasgow
http://eprints.gla.ac.uk
A Fast and Accurate Cost Model for FPGA Design Space Exploration in HPC Applications

Syed Waqar Nabi  
School of Computing Science  
University of Glasgow, Glasgow G12 8QQ  
syed.nabi@glasgow.ac.uk

Wim Vanderbauwhede  
School of Computing Science  
University of Glasgow, Glasgow G12 8QQ  
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. We contend that compiler technology has to evolve to automatically create the best program variant by transforming a given original program. We have developed a novel methodology based on type transformations for generating correct-by-construction design variants, and an associated light-weight cost model for evaluating these variants for implementation on FPGAs. In this paper we present a key enabler of our approach, the cost model. We discuss how we are able to quickly derive accurate estimates of performance and resource-utilization from the design’s representation in our intermediate language. We show results confirming the accuracy of our cost model by testing it on three different scientific kernels. We conclude with a case-study that compares a solution generated by our framework with one from a conventional high-level synthesis tool, showing better performance and power-efficiency using our cost model based approach.

I. INTRODUCTION

Higher logic capacity and maturing High-level Synthesis (HLS) tools are pushing FPGAs into the mainstream of heterogeneous High-Performance Computing (HPC) and Big Data. FPGAs allow configuration to a custom design at fine granularity. The advantage of being able to customize the circuit for the application comes with the challenge of finding and programming the best architecture for that kernel. HLS tools like Maxeler[1], Altera-OpenCL[2] and Xilinx SDAccel[3] have raised the abstraction of design entry considerably. Parallel programmers with domain 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 would 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[4]. SparkCL[5] brings increasingly diverse architectures, including FPGAs, into the familiar Apache Spark framework.

Our proposal however is to allow design-entry at a more fundamental and generic abstraction, inspired by functional languages with expressive type systems like Haskell\(^1\) or Idris\(^2\). The resultant flow, which we call the TyTra flow, is based on type-based program transformations, as shown in Figure 1. The design-entry is at a pure software, functional abstraction, and we leave the task of variant generation and tuning to the compiler. Program variants are generated using type transformations and translated to the TyTra intermediate language. The compiler costs the variants and emits HDL code. The HDL kernel-code is integrated with an existing HLS framework.

\(\text{Fig. 1. The TyTra design flow. Program variants are generated from a baseline functional description using type transformations and translated to the TyTra-IR.}\)

The focus of this paper is our cost model. It is a key enabler of design-flow, allowing us to quickly evaluate the large number of design variants that can be generated when we apply type transformations. We discuss how we move from an Intermediate Representation (IR) of a kernel’s design variant to an estimate of its performance, resource utilization and memory bandwidth, as shown in Figure 2. Our cost model also exposes the performance limiting parameter, allowing targeted optimization and opening the route to a feedback path in our compiler flow with automated, targeted tuning of designs.

\(^1\)http://www.haskell.org  
\(^2\)http://www.idris-lang.org/
By showing how quickly and accurately we can evaluate a design variant, we argue that our approach has tremendous potential to lead to a compiler that automatically creates and evaluates design variants for an HPC kernel, potentially saving days if not weeks of programming effort. We also show results of integrating HLS code generated by our compiler with a commercial HLS tool to show that we are able to create working solutions on an FPGA device, and that by exploring the design-space in our flow, we are able to exceed the performance achievable by the baseline HLS implementation.

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([1], [2], [6]), (2) compiler optimizations are limited to improving the overall architecture already specified by the programmer, with no real architectural exploration([1], [2], [6], [7]), (3) solutions are based on a creating soft-microprocessors on the FPGA and not optimized for HPC([7], [8]), (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[9]. The Geometry of Synthesis project[10] 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 [11] 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. A flow with high-level, pure software design-entry in the functional paradigm, that can apply safe transformations to generate variants automatically, and quickly evaluate them to achieve architectural optimizations, is to the best of our knowledge an entirely novel proposition.

II. Generating Variants through Type Transformations

We demonstrate how a program can be rewritten in a high-level functional 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 then select the best suited instance in a guided optimisation search.

Exemplar: Successive Over-Relaxation (SOR)

We consider an SOR kernel, taken from the code for the Large Eddy Simulator, an experimental weather simulator [12]. The kernel iteratively solves the Poisson equation for the pressure. The main computation is a stencil over the neighbouring cells (which is inherently parallel).

We express the algorithm in a functional language. 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 safe or correct-by-construction program variants through type transformations. We use a dependently-typed functional language Idris because the type transformations require dependent types which is intrinsically possible in Idris[13]. This feature is crucial for our purpose of generating program variants by reshaping data and ensuring correctness through type safety. The syntax is very similar to that of Haskell.

The baseline implementation of the SOR kernel in Idris is:

$$\text{ps} = \text{map p_sor pps}$$

$\text{pps}$ is a function that will take the original vectors $p$, $\text{rhs}$, $\text{cn}^*$ and return a single new vector equal to size of the 3D matrix $\text{im.jm.km}$. Each elements of this vector is a tuple consisting of all terms required to compute the SOR, i.e. the pressure $p$ at a given point, and its 6 neighbouring cardinal points, the weight coefficients $\text{cn}^*$ and the $\text{rhs}$ term for a given point.

$\text{p_sor}$ computes the new value for the pressure for a given input tuple from $\text{pps}$ against each input pressure point:

$$\text{p_sor pt} = \text{reltmp} + \text{p}$$

where

$$\text{pt = (p_i_pos, ..., p_rhs)}$$

$$\text{reltmp} = \omega \times (\text{cn1} \times (\text{cn2l} \times p_i_pos + \text{cn2s} \times p_i_neg$$
$$+ \text{cn3l} \times p_j_pos + \text{cn3s} \times p_j_neg$$
$$+ \text{cn4l} \times p_k_pos + \text{cn4s} \times p_k_neg) - \text{rhs}) - \text{p}$$

The function map performs computations on the vector without using explicit iterators. Here, map applies p_sor to every element of the vector pps in turn, resulting in the new pressure vector ps of size $\text{im.jm.km}$.

Our purpose is to generate variants by transforming the types 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 [14]. In brief, we reshape the vector in an order and size 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, over which
different parallelism patterns can be applied. We then use our cost model to choose the best design.

As an illustration, assume that the type of the 1D-vector is \( t \) (i.e. an arbitrary type) and its size \( \text{im} \times \text{jm} \times \text{km} \), which we can transform into e.g. a 2-D vector with sizes \( \text{im} \times \text{jm} \) and \( \text{km} \):

\[
\text{pps} : \text{Vect (} \text{im}\times\text{jm}\times\text{km} \text{)} \ t \quad \text{--1D vector}
\]

\[
\text{ppst} : \text{Vect (} \text{km} \text{) (} \text{Vect (} \text{im}\times\text{jm} \text{) } \text{t} \text{)} \quad \text{--transformed 2D vector}
\]

Resulting in a corresponding change in the program:

\[
\text{ps} = \text{map p_sor pps} \quad \text{--original program}
\]

\[
\text{ppst} = \text{reshapeTo km pps} \quad \text{--reshaping data}
\]

\[
\text{pst} = \text{map}(\text{par}) \text{ppst} \quad \text{--new program}
\]

where \( \text{map p_sor} \) is an example of partial application. Because \( \text{ppst} \) is a vector of vectors, the outer map takes a vector and applies the function \( \text{map p_sor} \) 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. Such a transformation is visualized in Figure 3.

![Fig. 3. Using type-transformations like reshapeTo, a baseline program which represents processing all \( \text{im} \times \text{jm} \times \text{km} \) items in a single pipeline fashion (top) is converted to a program that represents \( N \) concurrent pipelines, each now processing \( (\text{im} \times \text{jm} \times \text{km}) / N \) elements. The actual SOR kernel pipeline not shown for simplicity. It is illustrated in Figure 13.)](image)

![Fig. 4. The TyTra platform and memory model. Both these models map OpenCL abstractions to the FPGA architecture.](image)

Note the metadata information (pipe, par) in the map superscripts, indicating pipeline parallelism over the inner map, and thread-parallelism on the outer-map. By applying different combinations of parallelism keywords pipe, par and seq, and reshaping along different dimensions, it can be seen that the design-space grows very quickly even on the basis of a single basic reshape transformation. Developing a structured, accurate and fast evaluation approach is a key challenge of our approach.

### III. Models of Abstraction in the TyTra Framework

In general, we have adopted the models as defined in the OpenCL standard[15] wherever possible. While the TyTra flow is not intrinsically dependant on OpenCL, adopting its ecosystem is useful in creating a framework that is convenient for us and familiar to the community. We have defined the following models:

1) **Platform Model**: We have used OpenCL abstractions for FPGA-specific architectural elements. This is similar to the approach taken by the commercial OpenCL-FPGA solutions. The platform model, along with the memory model described next, is shown in Figure 4. The Compute Unit is the unit of execution for a kernel. The Processing Element 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. The stream-control block is transparent to the programmer and the compiler IR, but an integral part of the platform architecture as it translates between random memory access and a pure streaming domain.

2) **Memory Hierarchy Model**: As with the platform model, we adopt the OpenCL abstractions to describe the memory hierarchy on the FPGA, as shown in Figure 4. The number specified against each level provides us a convenient way of specifying the hierarchy in our TyTra-IR.

3) **Execution model**: The execution model too is adopted from the OpenCL standard, using terms like kernel, work-item, work-group, NDRange, global-size and kernel-instance. Interested readers are referred to the OpenCL standard[15] for definition of these terms. The term kernel-instance however is of special interest, as our throughput measure is defined with reference to it. It may be understood as the combination of a kernel (i.e. the function being executed on the device) and the entire index-space (i.e. NDRange) over which it executes. Hence execution of a kernel-instance means the execution of the kernel for all elements (work-items) of the index-space (NDRange).

4) **Design-Space Model**: FPGAs offer a much more flexible design-space than CPUs or GPUs. For an automatic design-space exploration framework like TyTra, defining it formally...
was a requirement. The key differentiating feature of concern is the type and extent of parallelism available in the design, based on which we developed the model as shown in Figure 5. E.g. a C2 configuration is a pipelined implementation of the kernel on the FPGA. The other horizontal axis indicates the degree of parallelism achieved by replicating the pipeline lane. A configuration in the xy-plane (C1) will thus have multiple threads of execution, each with pipeline parallelism. We expect C1 to be the preferable route for most small to medium sized kernels. For cases where a kernel may have too many instructions to fit entirely on the available FPGA resources as a pipeline, various configuration options are shown along the vertical axis.

![Diagram](image)

**Fig. 5.** The TyTra-FPGA Design Space Abstraction

5) Memory Execution Model: The need for defining a memory-execution model arose in response to the observation that a typical CPU-host–FPGA-device application can have different forms of execution with respect to data movement across the memory-hierarchy as multiple kernel-instances are executed. This type effects the performance significantly, and hence our cost model needs to take this into account.

We have defined three forms of memory-execution scenarios, which can be understood with reference to Figure 6. **Form-A** is where every kernel-instance requires all the data in the NDRange to be transported between the host and Device-DRAM. A **form-B** execution is where the data is moved to and from the global-memory only once by the host. The iterations in a kernel-instance then access the data from the global-memory. A **form-C** is a special case where the data needed for the NDRange is small enough to fit inside the local-memory, i.e. the on-chip block-RAMs of the FPGA. In such a case, all iteration of the kernel-instance will always access data from the on-chip local-memory. We expect this model to evolve to take into account tiling an index space such that it can lie on a finer-grained spectrum between these three main types.

6) Streaming Data Pattern Model: The TyTra Compute Units work with streams of data, and streaming from the global-memory is equivalent to looping over an array. Since the pattern of index-access has a significant impact on the sustained bandwidth (see V-C), there has to be a model that specifies this pattern explicitly, which can then be expressed in the IR description, and costed accordingly. Our prototype model currently considers two patterns: contiguous access, and strided access with constant strides.

IV. EXPRESSING DESIGNS IN THE TYTRA INTERMEDIATE REPRESENTATION LANGUAGE

The high-level description of the application kernel as described in section II is a pure software paradigm, and not directly cost-able. Generating and then costing HDL code on the other hand is too time-consuming. Our approach is to define an Intermediate Representation (IR) language, which we call the *TyTra-IR*. With reference to Figure 1, the TyTra-IR captures the design variants generated by the front-end type-transformations, and these variants are costed by parsing 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.

The TyTra-IR is currently used to express (and cost) the device-side code only, and 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 Assignments (SSA). It is based on the LLVM-IR, with extensions for parallelism suitable for an FPGA target. This gives us a baseline for designing our language, and leaves the route open to explore LLVM optimizations, as e.g. the LegUp [7] 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 instantiate *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 that was described in Figure 5.

The currently supported set of configurations are those suited for HPC applications, and are shown in Figure 7.

```
;1. Pipeline with combinatorial blocks
pipe {
  instr
  instr
  combA()
  ...
}

;2. Data-parallel pipelines
par {
  pipeA()
  pipeB()
  ...
}

;3. Coarse-grained
pipeTop {
  pipe {
    pipeA()
    pipeB()
    ...
  }
  ...
}

;4. Coarse-grained pipeline
pipeTop {
  pipeTop()
  ...
  ...
  ...
}

;where
pipeTop{
  pipeA()
  pipeB()
  ...
}
```

Fig. 7. Configurations currently supported by the TyTra compiler.

The TyTra compiler parses the IR description of a design-variant expressed using these parallelism constructs, and extracts the architecture from it. As an example, Figure 8 shows the configuration tree created for a coarse-grained pipeline where one of the peer kernels uses a combinatorial function (i.e. a single-cycle custom combinatorial block).

V. ANALYTICAL AND EMPIRICAL COST MODELS FOR EVALUATING DESIGN VARIANTS IN THE TYTRA FLOW

We have developed a prototype compiler that can parse TyTr-IR and emit various cost and performance estimates, as shown in Figure 2. Our cost models have both analytical and empirical elements, the latter requiring one to run a set of benchmark experiments for every new FPGA target platform. We will describe both elements of our cost models against the backdrop of the abstractions developed in section III.

A. Resource-Utilization Cost Model

Our observation is that the regularity of FPGA fabric allows some very simple first or second order expressions to be built up for most primitive 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 9. It was generated from three-data points (18, 32 and 64 bits) from synthesis experiment on 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.

A multiplier requires two different kinds of resources: DSP-elements and ALUTs. Both these resources show a piece-wise-linear behaviour with respect to the bit-size, with clearly identifiable points of discontinuity, also shown in Figure 9. This results in a relatively trivial expression that we use in the cost-model. Other IR instructions have similar or simpler expressions that we can use to estimate their resource-utilization. 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.

```
0 10 20 30 40 50 60 70
DSP Blocks
ALUTs

10 20 30 40 50 60 70 80
Pol. (div-ALUTs [x100])
mul-DSP elements
div-ALUTs [x100]
```

Fig. 8. A typical configuration generated by the TyTra compiler showing a coarse-grained pipeline where one of the peer kernels uses a custom combinatorial function

Fig. 9. Deriving cost expressions for ALUTs used in unsigned integer division (see polynomial trend-line), and ALUTs and DSP-elements used in unsigned integer multiplication, on a Stratix-V device.
B. Throughput Cost Model

An estimate of the performance of a design variant described in the TyTra-IR is essential to TyTra flow. This estimate can be expected to be the main differentiating parameter when choosing from multiple variants. The other estimates would typically only be used to confirm whether or not a particular design variant is valid, based on the limits of FPGA resources and IO bandwidth.

We have described a performance measure called the EKIT (Effective Kernel-Instance Throughput) for comparing how fast a kernel executes across different design points, with “kernel-instance” used as defined in the OpenCL standard. Measuring throughput at this granularity allows us to reason at a coarse enough level to take into account parameters like memory latencies and throughput for different kinds of data access patterns, as well as dynamic reconfiguration penalty if applicable. With the models developed as described in section III, a cost function for the throughput can now be developed.

Table I lists all the parameters that make up the expression for EKIT described later, their key dependence (program, target-hardware, design-variant), and with a description of how we expect to evaluate them in the TyTra compiler.

Table I. The Parameters that Make Up the Expressions for EKIT Given Along with Their Key Dependence and the Way They Are Evaluated in the TyTra Compiler.

<table>
<thead>
<tr>
<th>Param'</th>
<th>Description</th>
<th>Key Dependence</th>
<th>Evaluation Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>N(_{KI})</td>
<td>Number of times kernel-instance executed over all N(_{GS}) work-items</td>
<td>Kernel</td>
<td>Parsing IR</td>
</tr>
<tr>
<td>N(_{off})</td>
<td>Maximum offset in a stream</td>
<td>Kernel</td>
<td>Parsing IR</td>
</tr>
<tr>
<td>N(_{WPT})</td>
<td>Words per tuple per work-item</td>
<td>Kernel</td>
<td>Parsing IR</td>
</tr>
<tr>
<td>N(_{PD})</td>
<td>Pipeline depth of kernel</td>
<td>Design-variant</td>
<td>Parsing IR</td>
</tr>
<tr>
<td>F(_D)</td>
<td>Device’s operating frequency</td>
<td>Design-variant</td>
<td>Parsing IR</td>
</tr>
<tr>
<td>N(_{TO})</td>
<td>Cycles per instruction</td>
<td>Design-variant</td>
<td>Parsing IR</td>
</tr>
<tr>
<td>N(_{I})</td>
<td>Instructions per PE</td>
<td>Design-variant</td>
<td>Parsing IR</td>
</tr>
<tr>
<td>N(_{KL})</td>
<td>Number of parallel kernel lanes</td>
<td>Design-variant</td>
<td>Parsing IR</td>
</tr>
<tr>
<td>D(_V)</td>
<td>Degree of vectorization per lane</td>
<td>Design-variant</td>
<td>Parsing IR</td>
</tr>
</tbody>
</table>

The kernel-instance throughput is equal to the number of kernel-instance repetitions N\(_{KI}\), divided by the time taken to execute all the kernel-instances. The EKIT expressions are developed separately for the three types of implementations as described in the memory-execution model in III.

Form A: The total time taken to execute all kernel-instances, T\(_{NKI}\), is composed of four elements, i.e. the times to: 1) transfer data between host and device DRAM, 2) fill offset stream buffers until the first work-item can be processed, 3) fill the kernel pipeline with work-items and 4) execute all work-items on the device. The last element depends on either the external memory bandwidth, or the maximum throughput achievable on the device pipeline at its operating frequency. The smaller of the two becomes the limiting factor and determines the overall throughput. The resulting expression for form-A is Equation 1:

\[
EKITA = \frac{N_{GS} \cdot N_{WPT}}{H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \max(\frac{N_{GS} \cdot N_{WPT}}{G_{PB} \cdot \rho_G}, \frac{N_{GS} \cdot N_{WPT} \cdot N_{TO} \cdot N_I}{F_D \cdot K_{NL} \cdot D_V})
\]

Form B: With reference to Figure 6, in form-B memory-access scenarios the data, once available in the device DRAMs, is always accessed from the DRAM for all kernel-instance iterations. We expect this to be the form for most real scientific applications, which are generally too large to fit the entire kernel-instance on the device (so not form-C), and not so large that the kernel-instance cannot fit on the (increasingly large) DRAMs available on HPC PCIe boards (so not form-A). The expression for type-B implementations can be derived very simply from Equation 1 by scaling down the contribution of host-device transfer by a factor of N\(_{KI}\), which is the number of times the kernel-instance repeats.

\[
EKITB = \frac{N_{GS} \cdot N_{WPT}}{N_{KI} \cdot H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \max(\frac{N_{GS} \cdot N_{WPT}}{G_{PB} \cdot \rho_G}, \frac{N_{GS} \cdot N_{WPT} \cdot N_{TO} \cdot N_I}{F_D \cdot K_{NL} \cdot D_V})
\]

Form C: Since form-C programs are those where the kernel-instance data remains on the device throughout the N\(_{KI}\) iterations, the overall performance is always compute-bound. This implies that the expression for form-C is simply a specialized form of the one for form-B with the max function replaced by its second argument only, that is, the limiting factor in a compute-bound scenario.

\[
EKITC = \frac{N_{GS} \cdot N_{WPT}}{N_{WU} \cdot H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \max(\frac{N_{GS} \cdot N_{WPT} \cdot N_{TO} \cdot N_I}{F_D \cdot K_{NL} \cdot D_V})
\]

C. Sustained Stream Bandwidth

A significant variable in the throughput expressions is the bandwidth to the host or the device-DRAM. While the peak bandwidth can easily be read off the data-sheets, the sustained bandwidth for various streams in a particular design varies with the access-pattern and size. We performed a set of experiments by extending the stream benchmark[16] to OpenCL, based on the work done in [17] for GPUs, but in our case using SDAccel[3] for FPGAs. Specifically, we tested the effect of...
having the data streams access data contiguously and in a strided manner, and changing the size of the streams and the strides. We have currently only tested fixed-stride for non-contiguous patterns, but our experiments have shown that there is little difference in sustained bandwidth between fixed-stride and true random access. The results are shown in Figure 10. Note how the contiguity of data access has an up to two-orders-of-magnitude impact on the sustained bandwidth. Also note the considerable effect of stream-size on sustained bandwidth for contiguous access especially up to 1000×1000 elements, after which it plateaus. These trends highlight the importance of taking into account the factors effecting the sustained bandwidth for any realistic cost models. We have incorporated this empirical model into our compiler, and expect to further develop this streaming benchmark for FPGAs.

Fig. 10. Developing an empirical model of sustained bandwidth’s dependency on data size and contiguity of data. The horizontal axis represents the size of one dimension of a square 2D array. In case of strided access, this is equal to the stride as well. Results are based on Alpha-Data’s ADM-PCIe-7V3 board with a Xilinx Virtex 7 device. It is important to note that these are baseline figures for the target device, without using any vendor-recommended optimizations.

VI. USING THE COST MODEL

We have developed a back-end compiler that accepts a design variant in TyTra-IR, costs it and, if needed, generates the HDL code for it, as shown in Figure 11. We hand-coded some design variants of the SOR kernel generated by type-transformations as discussed in II. Figure 12 shows the TyTra-IR for the a baseline configuration which is a single kernel-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 that is fed into the datapath pipeline described in lines 10–15. Figure 13 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 Figure 12 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 8.

Fig. 11. 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 synthesizable HDL which can then be integrated with a HLS framework.

A. Evaluating TyTra-IR Design Variants using the cost model

We use the high-level reshapeTo function to generate variants of the program by reshaping the data, which means you 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 15 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 (form-A), then beyond 4 lanes, we encounter the host communication wall. If all the data is made available in the device’s global (on-board)
We tested the cost model by evaluating the integer version of kernels 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[18], 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 Since these kernels were compute-bound, so the off-chip stream cost model developed in V-C did not come into play.

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.

VII. 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 a wrapper kernel written in its kernel language MaxJ for the custom HDL module. Currently, we create the MaxJ wrapper kernel manually for each design, but generating them in our compiler is expected to be a relatively trivial engineering task. Figure 16 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 17). The baseline is a simple Fortran-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). We fixed the value of nmaxp, the number of times the SOR kernel repeats – and changes only with changing grid-size.

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

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

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 18, 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.

VIII. 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, we have shown that 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 shown an original flow that has a high-level design entry in a functional language, generates and evaluates design variants using a cost model on an intermediate description of the kernel, and then emits HDL code. We developed abstractions used to create a structured cost model, and then described our empirical and analytical cost models to estimate the utilization of various resources on the FPGA, the sustained bandwidth to the FPGA for a specific data patterns, and the overall throughput achievable. 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 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.

We are currently in the process of automating the generation of design variants from the high-level code. We are also testing the cost-model and code-generator with larger and more complex kernels. Once complete, this will provide us with a solution which has a high abstraction design-entry, and in addition will automatically converge on the best design variant from a single high-level description of the algorithm in a functional language. Eventually, we plan to evolve our flow to include legacy code written in languages typically used for scientific computing like Fortran or C.

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

References