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.

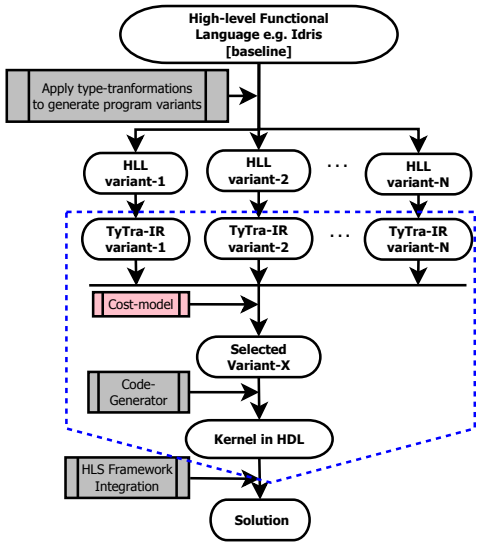

Fig. 1. The TyTra design flow. Program variants are generated from a baseline functional description using type transformations and translated to the TyTra-IR. The compiler costs the variants and emits HDL code. The HDL kernel-code is integrated with an HLS framework. The dotted line marks the stages that are currently automated.

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/

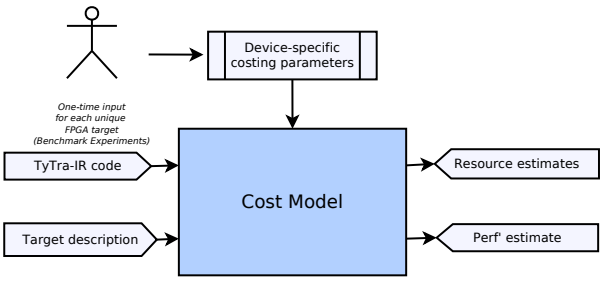

Fig. 2. The cost model use-case. A one-time set of benchmark experiments are carried out for each FPGA target. The cost model requires target description and the IR for the design, emitting estimates.

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 HDL 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:

```
ps = map  p_sor  pps
```

pps is a function that will take the original vectors *p, rhs, cn** and return a single new vector equal to size of the 3D matrix *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 *cn** and the *rhs* term for a given point.

p_sor computes the new value for the pressure for a given input tuple from *pps* against each input pressure point:

```
p_sor pt = reltmp + p
where
(p_i_pos,...,p,rhs) = pt
reltmp = omega * (cn1 * (
    cn2l * p_i_pos + cn2s * p_i_neg
  + cn3l * p_j_pos + cn3s * p_j_neg
  + cn4l * p_k_pos + cn4s * p_k_neg ) - rhs) - 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 *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 *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^par (map^pipe 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. Such a transformation is visualized in Figure 3.

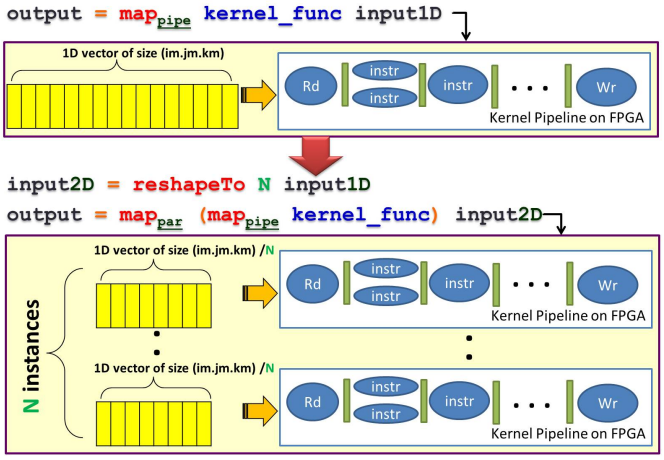

Fig. 3. Using type-transformations like *reshapeTo*, a baseline program which represents processing all `im.jm.km` items in a single pipeline fashion (top) is converted to a program that represents `N` concurrent pipelines, each now processing `(im.jm.km)/N` elements. (The actual SOR kernel pipeline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

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.

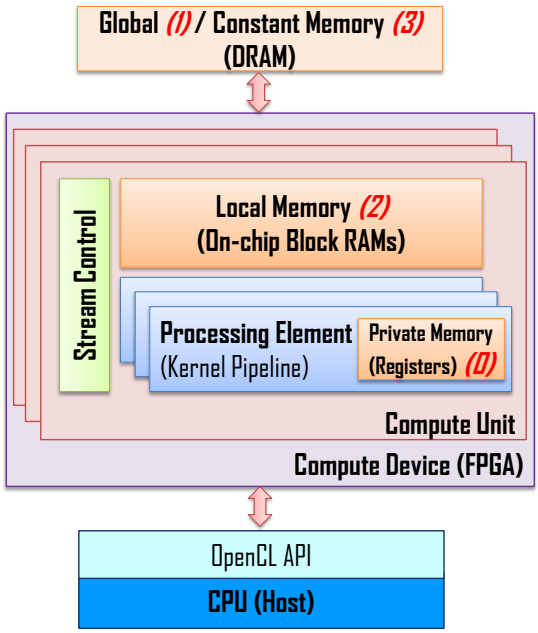

Fig. 4. The TyTra platform and memory model. Both these models map OpenCL abstractions to the FPGA architecture.

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.

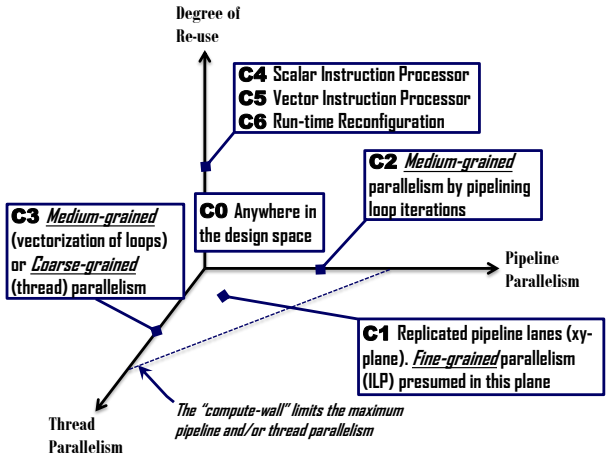

Fig. 5. The TyTra-FPGA Design Space Abstraction

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 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

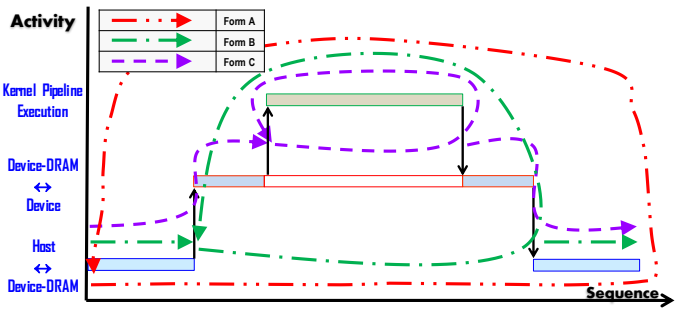

Fig. 6. The three forms of execution defined based on how the memory-hierarchy is traversed across multiple kernel-instance iterations. The through-put expressions developed later are different for each of the three forms.

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 instantiates *memory-objects* which is abstraction for any entity that can be the source or sink for a stream. In most cases, a memory object's

equivalent in a software description would be an array in main memory. It also has *stream-objects*, connecting a streaming port in the processing element to a memory-object.

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

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

;4. Data-parallel
;   Coarse-grained pipeline
par {
  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 TyTra-IR and emit various cost and performance estimates, as shown in Figure 2[3]. 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

[3]The compiler can also emit synthesizeable HDL code for the kernel pipeline.

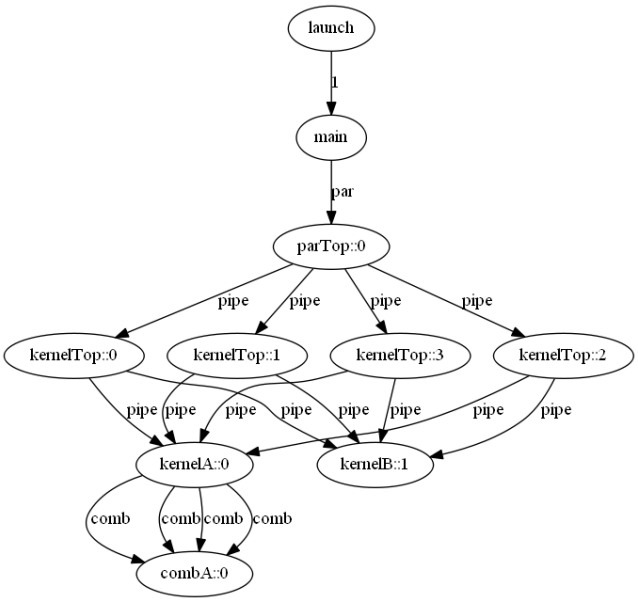

Fig. 8. A typical configuration generated by the TyTra compiler showing a coarse-grained pipeline where one of the peer kernels uses a custom combinatorial function

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.

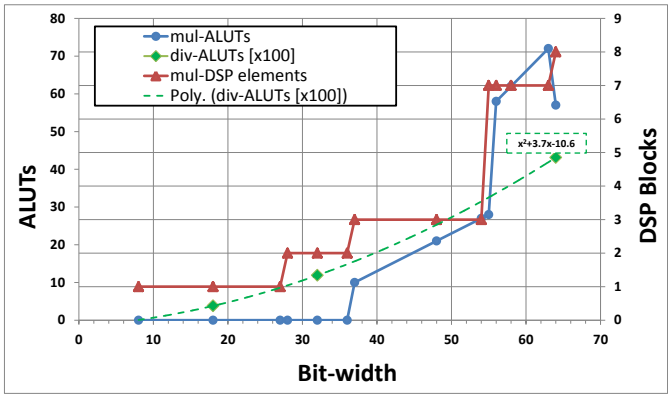

Fig. 9. Deriving cost expressions for ALUTs used in unsigned integer division (see polynomial trend-line), and ALUTs and DSP-elements used in unsigned integer multiplication, on a Stratix-V device.

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 kind 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.

Param'	Description	Key Dependence	Evaluation Method
H_{PB}	The host-device peak bandwidth	Target device	Architecture description
ρ_H	Scaling factor, host-device bandwidth	Target device & design-variant	Empirical data
G_{PB}	The device DRAM peak bandwidth.	Target device	Architecture description
ρ_H	Scaling factor, host-device bandwidth	Target device & design-variant	Empirical data
N_{GS}	Global-size of work-items in NDRange	Kernel	Parsing IR
N_{WPT}	Words per tuple per work-item	Kernel	Parsing IR
N_{KI}	Number of times kernel-instance executed over all N_{GS} work-items	Kernel	Parsing IR
N_{off}	Maximum offset in a stream	Kernel	Parsing IR
K_{PD}	Pipeline depth of kernel	Design-variant	Parsing IR
F_D	Device's operating frequency	Design-variant	Parsing IR
N_{TO}	Cycles per instruction	Design-variant	Parsing IR
N_I	Instructions per PE	Design-variant	Parsing IR
K_{NL}	Number of parallel kernel lanes	Design-variant	Parsing IR
D_V	Degree of vectorization per lane	Design-variant	Parsing IR

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.

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:

$$EKIT_A = 1 \div (\frac{N_{GS} \cdot N_{WPT}}{H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + max(\frac{N_{GS} \cdot N_{WPT}}{G_{PB} \cdot \rho_G}, \frac{N_{GS} \cdot N_{WPT} \cdot N_{TO} \cdot N_I}{F_D \cdot K_{NL} \cdot D_V})) \quad (1)$$

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.

$$EKIT_B = 1 \div (\frac{N_{GS} \cdot N_{WPT}}{N_{KI} \cdot H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + max(\frac{N_{GS} \cdot N_{WPT}}{G_{PB} \cdot \rho_G}, \frac{N_{GS} \cdot N_{WPT} \cdot N_{TO} \cdot N_I}{F_D \cdot K_{NL} \cdot D_V})) \quad (2)$$

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.

$$EKIT_C = 1 \div (\frac{N_{GS} \cdot N_{WPT}}{N_{WU} \cdot H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \frac{N_{GS} \cdot N_{WPT} \cdot N_{TO} \cdot N_I}{F_D \cdot K_{NL} \cdot D_V}) \quad (3)$$

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.

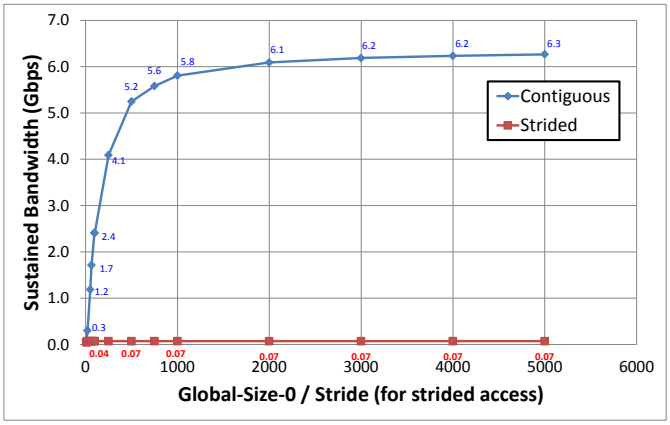

Fig. 10. Developing an empirical model of sustained bandwidth's dependency on data size and contiguity of data. The horizontal axis represents the size of one dimension of a square 2D array. In case of strided access, this is equal to the stride as well. Results are based on Alpha-Data's ADM-PCIE-7V3 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 a the 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.

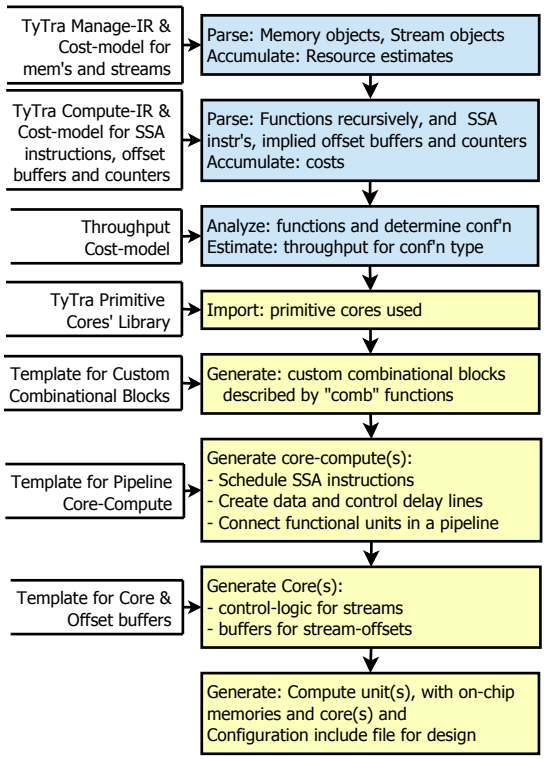

Fig. 11. The TyTra back-end compiler flow, showing the estimation flow (blue/first three stages) and code generation flow (yellow). The starting point for this subset of the entire flow is the TyTra-IR description representing a particular design variant, ending in the generation of synthesizeable HDL which can then be integrated with a HLS framework.

```
1  ; **** COMPUTE-IR ****
2  @main.p   = addrSpace(12) ui18,
3            !"istream", !"CONT", !0, !"strobj_p"
4  ;...[more inputs]...
5  define void @f0(...args...) pipe {
6    ;stream offsets
7    ui18 %pip1=ui18 %p, !offset, !+1
8    ui18 %pkn1=ui18 %p, !offset, !-ND1*ND2
9      ;...[more stream offsets]...
10   ;datapath instructions
11   ui18 %1 = mul ui18 %p_i_p1, %cn21
12   ui18 %2 = mul ui18 %p_i_n1, %cn2s
13     ;..[more instructions]...
14   ;reduction operation on global variable
15   ui18 @sorErrAcc=add ui18 %sorErr, %sorErrAcc
16 }
17 define void @main () {
18   call @f0(..args...) pipe }
```

Fig. 12. Abbreviated TyTra-IR code for the SOR kernel configured as a single pipeline lane.

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 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 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)

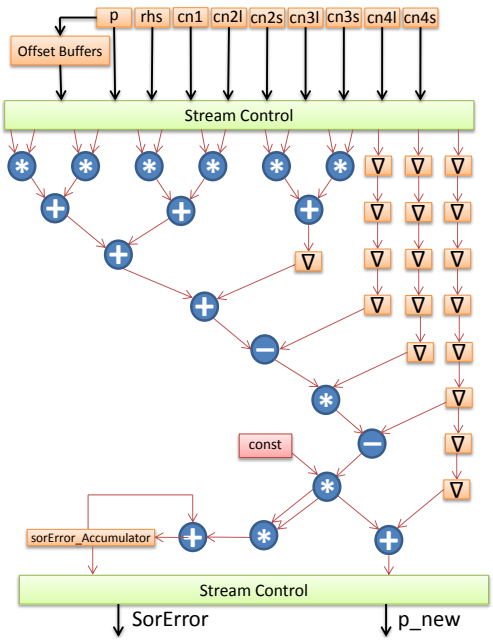

Fig. 13. Illustration of the pipelined datapath of the SOR kernel generated by our compiler. Only pass-through pipeline buffers are shown; all functional units have pipeline buffers as well. The blocks at edges refer to on-chip memory for each data.

```
1  ; **** COMPUTE-IR ****
2  @main.p0  = addrSpace(12) ui18,
3            !"istream", !"CONT", !0, !"strobj_p"
4  @main.p1  = ...
5  @main.p2  = ...
6  @main.p3  = ...
7  ;...[other inputs]...
8  define void @f0(...args...) pipe {...}
9  define void @f1 (...args...) par {
10   call @f0(...args...) pipe
11   call @f0(...args...) pipe
12   call @f0(...args...) pipe
13   call @f0(...args...) pipe }
14 define void @main () {
15   call @f1(..args...) par }
```

Fig. 14. Abbreviated TyTra-IR code for the SOR kernel configured with multiple pipelines lanes corresponding to reshaped data.

memory (form-B) then the communication wall moves to about 16 lanes. We encounter the *computation-wall* at six lanes, where we run out of LUTs on the FPGA. However, we can see other resources are underutilized, and some sort of resource-balancing can lead to further performance improvement.

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

B. Accuracy of the cost model

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

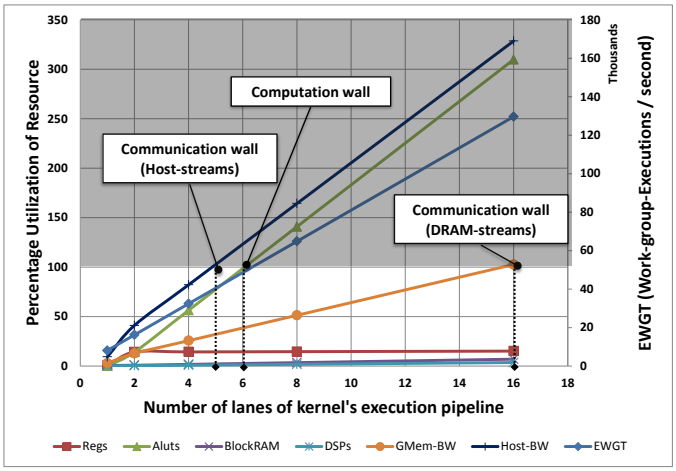

Fig. 15. Evaluation of variants for the SOR kernel generated by changing the number of kernel-pipelines (16 data points and 10 kernel iterations).

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

TABLE II. THE ESTIMATED VS ACTUAL PERFORMANCE AND UTILIZATION OF RESOURCES, THE FORMER MEASURED IN TERMS OF CYCLES-PER-KERNEL-INSTANCE (CPKI), FOR THREE SCIENTIFIC KERNELS. PERCENTAGE ERRORS ALSO SHOWN.

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.

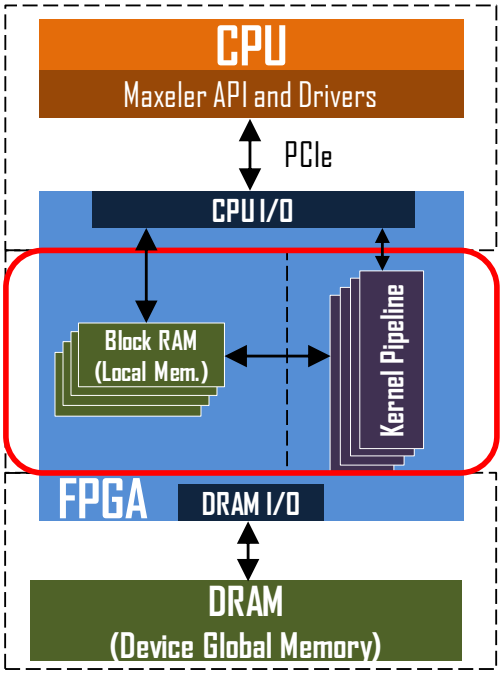

Fig. 16. The Maxeler-TyBEC integrated solution. The black dotted line identifies what is programmed using the Maxeler HLS tool. The solid/red line identifies the logic programmed with TyTra generated code. The overlap indicates that stream generation from on-chip Block-RAMs may be done by either in our flow.

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 Fortan-based CPU implementation (*cpu*) compiled with `gcc -O2`. The first FPGA implementation is using only the Maxeler flow (*fpga-maxJ*), which incorporates pipeline parallelism automatically extracted by the Maxeler compiler. The second FPGA implementation (*fpga-tytra*) is the design variant generated by the TyTra back-end compiler, based on a high type-transformation that introduced thread-parallelism (4 lanes) in addition to pipeline parallelism. We collected results for different dimensions of the input 3D arrays, i.e. `im, jm, km`, ranging from 24 elements along each dimension (55 KB) to 194 elements (57 MB). We fixed the value of `nmaxp`, the number of times the SOR kernel

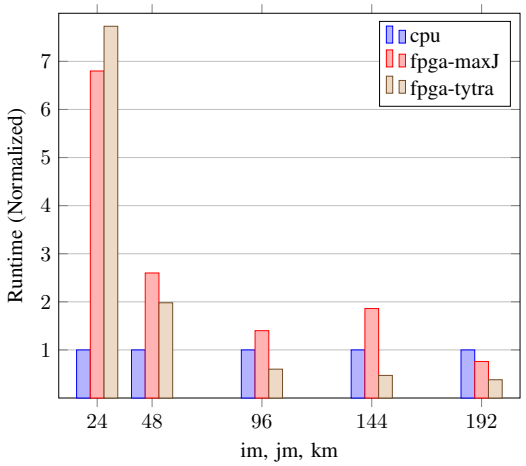

Fig. 17. Runtime of the SOR kernel for different sizes of grid, normalized against the CPU-only solution. The figures are for 1000 iterations of the kernel.

is called for each run, at 1000. [4]

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

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

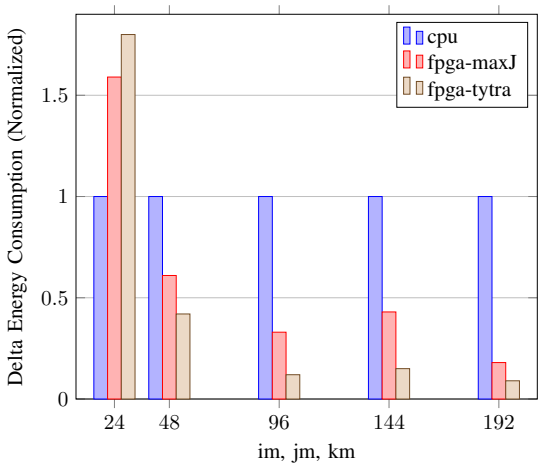

Fig. 18. *Increase* from idle energy-consumption, for calculating the SOR kernel for different sizes of grid, normalized against the CPU-only solution. The figures are for 1000 iterations of the kernel.

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

[1] O. Pell and V. Averbukh, "Maximum performance computing with dataflow engines," *Computing in Science Engineering*, vol. 14, no. 4, pp. 98–103, July 2012.

[2] 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*, Aug 2012, pp. 531–534.

[3] "The Xilinx SDAccel Development Environment," 2014. [Online]. Available: http://www.xilinx.com/publications/prod_mktg/sdx/sdaccel-backgrounder.pdf

[4] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," *Parallel Computing*, vol. 30, no. 3, pp. 389 – 406, 2004.

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

[6] J. e. a. Keinert, "Systemcodesigner;an automatic esl synthesis approach by design space exploration and behavioral synthesis for streaming applications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, no. 1, pp. 1:1–1:23, Jan. 2009.

[7] 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*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36.

[8] 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*, Sept 2005, pp. 273–278.

[9] 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*, ser. DAC '99. New York, NY, USA: ACM, 1999, pp. 616–622.

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

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

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

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

[14] W. Vanderbauwhede, "Inferring Program Transformations from Type Transformations for Partitioning of Ordered Sets," 2015. [Online]. Available: http://arxiv.org/abs/1504.05372

[15] "The OpenCL Specification," 2015. [Online]. Available: https://www.khronos.org/registry/cl/

[16] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[17] T. Deakin and S. McIntosh-Smith, "Gpu-stream: Benchmarking the achievable memory bandwidth of graphics processing units," in *IEEE/ACM SuperComputing, Austin, United States*, 2015.

[18] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct 2009, pp. 44–54.