FPGA Design Space Exploration for Scientific HPC Applications Using a Fast and Accurate Cost Model Based on Roofline Analysis

Syed Waqar Nabi[a,*], Wim Vanderbauhede[a]

[a]*School of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK. T:+44 (0) 141 330 2074.*

Abstract

High-performance computing on heterogeneous platforms in general and those with FPGAs in particular presents a significant programming challenge. We contend that compiler technology has to evolve to automatically optimized applications by transforming a given original program. We are developing a novel methodology based on *type transformations* on a *functional* description of a given scientific kernel, for generating *correct-by-construction* design variants. An associated lightweight costing mechanism for evaluating these variants is a cornerstone of our methodology, and the focus of this paper. We discuss our use of the *roofline model* to work with our optimizing compiler to enable us to quickly derive accurate estimates of performance from the design's representation in our custom intermediate language. We show results confirming the accuracy of our cost model by validating it on different scientific kernels. A case study is presented to demonstrate that a solution

*Corresponding author

Email addresses: `syed.nabi@glasgow.ac.uk` (Syed Waqar Nabi), `wim.vanderbauwhede@glasgow.ac.uk` (Wim Vanderbauhede)

created from our optimizing framework outperforms commercial high-level synthesis tools both in terms of throughput and power efficiency.
Keywords: performance model, cost model, high-performance computing, FPGA, high-level programming, high-level synthesis, roofline model

1. Introduction

Higher logic capacity and maturing high-level synthesis (HLS) tools are drivers to mainstream adoption of FPGAs in high-performance computing (HPC) and big data. The fine-grained flexibility of an FPGA comes with the challenge of figuring out and programming the best architecture for a given scientific kernel. HLS tools like Maxeler[1], Altera OpenCL[2], Xilinx SDAccel[3] and LegUp[4] have raised the abstraction of design entry considerably and made it easier to program FPGAs. Parallel programmers with domain expertise are however still needed to fine-tune the application for performance and efficiency. "Portable" heterogeneous frameworks like OpenCL are playing an important role in making heterogeneous computing more accessible, but they are not *performance*-portable across devices [5]. The performance portability issue is all the more acute with FPGAs. We contend that the design flow for HPC needs to evolve beyond current HLS approaches to address the productivity gap between the capacity of modern devices and our ability to efficiently program them. Our proposition is that for true performance portability, the design entry should be at a higher level of abstraction, and that the task of generating architecture-specific parallel code should be done by the compilers

Our proposal is to allow design entry at a 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* (or *type transformations* for short)

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

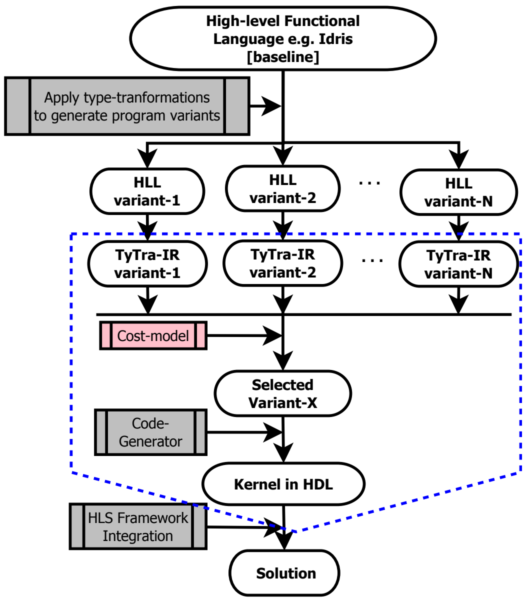


Figure 1: The TyTra design flow, showing design entry in a functional language, to an optimized FPGA solution. The dotted line marks the stages that are currently automated.

as shown in Figure 1. The design entry is at a pure software, *functional* abstraction, with no explicit parallel programming required by the user. We transfer the task of variant generation, search space exploration and converging on the optimal solution to the compiler. Program variants are generated using *type transformations* and translated to the TyTra intermediate language (IR). The compiler internally analyses the variants and emits code in a hardware description language (HDL), which is integrated with an existing HLS programming framework.

A key enabler of our approach is the performance and cost model embedded inside our flow, based on *roofline analysis* [6]. An automated search space explorer based on the roofline model is an entirely novel proposition to the best of our knowledge, and is the main contribution of this paper.

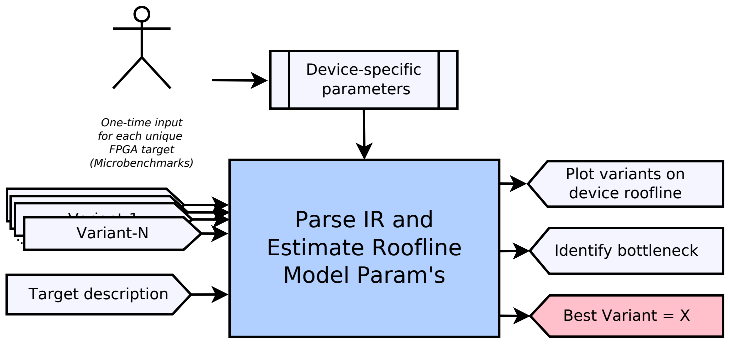

One-time input for each unique FPGA target (Microbenchmarks)
Device-specific parameters
Variant-1
Variant-N
Target description
Parse IR and Estimate Roofline Model Param's
Plot variants on device roofline
Identify bottleneck
Best Variant = X

Figure 2: The use case of the cost model that is integrated inside the TyTra design flow.

The use case of our cost model is shown in Figure 2. A one-time set of synthetic micro-benchmark experiments are required for each new FPGA target. Then, given the IR descriptions of multiple, functionally equivalent design variants of a given kernel, we obtain estimates of their cost and performance on a roofline model, and pick the best performing variant from the search space. Note that the performance estimate on the roofline requires estimates of FPGA resource utilization and achievable memory bandwidth for each variant, and our methodology for calculating these estimates forms an important contribution of this work.

Our work is oriented towards the general area of high-performance scientific computing. Such applications are generally amenable to streaming, leading to pipelined implementations on FPGAs, and our framework is optimized for this pattern. However, in principle, our approach is meant to be generic and comprehensive.

The rest of the paper is organized as follows: We first present the models of abstraction that we have developed or adopted in our framework. We then show how design variants are generated in the search space using *type* *transformations*, and how these variants are represented in our custom IR. Next we present the *roofline analysis* model, followed by a section on how our IR based compilation approach allows us to quickly and accurately estimate the parameters required for the roofline analysis. We give an illustration of a comparative roofline analysis in the TyTra compiler, and an optimization case study using our approach. We finally present some related prior work before concluding the paper.

2. Models of Abstraction in the TyTra Framework

In general, we have adopted the models as defined in the OpenCL standard [7] wherever possible, as this provides us with a familiar anchor, and suits our aim of eventually making our compiler work not just for FPGAs but for truly heterogeneous platforms.

2.1. Platform and Memory Hierarchy Model

The platform model, based on the OpenCL model, along with the memory model described later, is shown in Figure 3. The *Compute Unit* is the unit of execution for a kernel. The *Processing Element* (PE) is the custom datapath unit created for a given kernel, and may be considered equivalent to a pipeline *lane*, which may be *scaled* (i.e. replicated) for parallelism if there are enough resources on the FPGA.

As with the platform model, we adopt the OpenCL abstractions to describe the memory hierarchy on the FPGA, also shown in Figure 3.

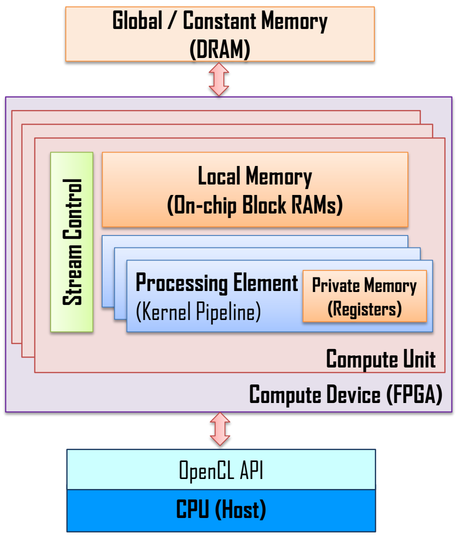


Figure 3: The TyTra platform and memory model. Both these models map OpenCL abstractions to the FPGA architecture.

2.2. Kernel Execution model

The execution model too is adopted from the OpenCL standard, using terms like *kernel, work-item, work-group, NDRange, global-size, kernel-instance*. Readers are referred to the OpenCL standard [7] for definition of these terms.

2.3. Memory Execution Model

A typical host–device partitioned application can have different *forms* of execution with respect to data movement across the memory hierarchy, as multiple *kernel-instances*[3] are executed. This *form* effects the achievable performance significantly. Hence, our framework requires a structured way of taking this into account when estimating performance.

[3]A kernel-instance is execution of the kernel for all *work-items* in the *NDRange*.

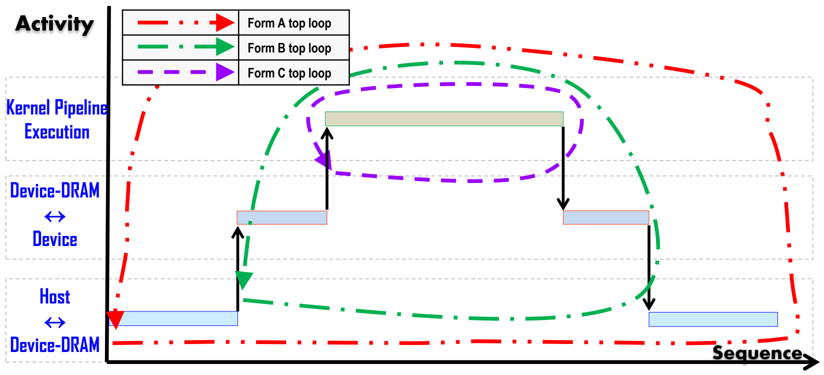


Figure 4: The three forms of execution based on how the memory hierarchy is traversed across multiple iterations of the top loop.

We have defined three forms of memory execution scenarios, which can be understood with reference the memory hierarchy (host ↔ device-DRAM ↔ device) and to Figure 4. *Form A* is where the top loop – the one that repeatedly executes the kernel over the entire index-space (generally the *time* loop in scientific applications) – is *outside* the host↔device-DRAM memory transfers, so these transfers take place on every iteration. A *Form B* execution is where the time loop is inside the host↔device-DRAM transfer, but outside the device-DRAM↔device transfer, so the host↔device-DRAM transfer happens only once. The iterations in a kernel-instance then access the data from the device-DRAM, i.e. the global memory. *Form C* is where the arrays for all work-items are small enough to fit inside the *local memory*. In such a case, the time loop over the kernel-instance is inside both the host↔device-DRAM and device-DRAM↔device transfers[4].

[4]We expect this model to evolve to take into account tiling the NDRange such that we can have a finer-grained spectrum between these three main forms.

2.4. Data Pattern Model

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 §6.3), this needs to be modelled. Our prototype model currently considers two patterns: contiguous access and strided access with constant strides. We plan to explore more sophisticated models in future versions.

3. Generating Variants in the Search Space through Type Transformations

A defining feature of our compiler is the generation of the search space by creating *correct-by-construction* variants from a functional, high-level, baseline description of a kernel through *type transformations*. Each program variant will have a different cost and performance related to its degree of parallelism. Using our roofline based cost model we can then select the best suited instance in a guided optimisation search.

Exemplar: Successive Over-Relaxation (SOR)

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).

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 it supports dependent types which the type transformations require [9]. This feature is crucial for our purpose of generating program variants by reshaping data and ensuring correctness through type safety.

The baseline implementation of the SOR kernel in Idris is:

```
ps = map  p_sor  pps  p rhs cn
```

p, rhs, and *cn* are the original vectors in the application, which are passed to the function *pps* that returns a *single* new vector equal to size of the 3D matrix *im.jm.km*. Each element 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.

Each tuple from this 3D matrix is passed to the computation kernel *p_sor* which computes the new value for the pressure:

```
p_sor pt = reltmp + p                   --'pt' is the tuple passed to p_sor
                                        --which then returns new pressure
                                        --based on original and delta
where
(p_i_pos,...,p,rhs) = pt                --extract scalars from tuple 'pt'
reltmp = omega * (cn1 * (               --compute pressure delta 'reltmp'
    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 high-level function *map* performs computations on a vector without using explicit iterators. So *map* applies *p_sor* to every element – which is a tuple – of the 3D matrix returned by *pps*, 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 [10]. 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 maps over 1D
ppst= reshapeTo km pps                 --reshaping data
pst  = map-par (map-pipe p_sor) ppst --new program with nested map over 2D
                                       --with parallelism annotation
```

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 processing elements (pipelines) in the FPGA. Such a transformation is visualized in Figure 5.

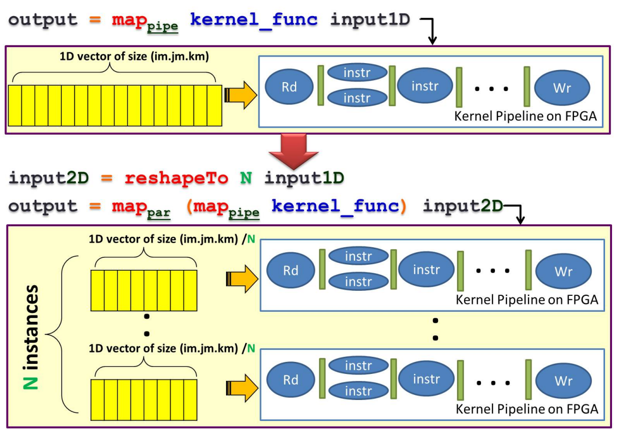

output = map_pipe kernel_func input1D
1D vector of size (im.jm.km)
Rd
instr
instr
instr
Wr
Kernel Pipeline on FPGA
input2D = reshapeTo N input1D
output = map_par (map_pipe kernel_func) input2D
N instances
1D vector of size (im.jm.km) /N
Kernel Pipeline on FPGA

Figure 5: 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.

By applying different combinations of parallelism keywords *pipe, par* and *seq*, and reshaping along different dimensions, the search space very quickly explodes 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.

4. Expressing Designs in the TyTra Intermediate Representation Language

The high-level description of the application kernel as described in section 3 and not directly costable. Generating and then costing HDL code through synthesis and place-and-route 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, which can then be costed. 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 used to express the device-side code only, and models all computations on a dataflow machine rather than a von-Neumann architecture. The host-device interactions are managed by using the *shell* of a commercial HLS tool around the TyTra generated *kernel*.

The TyTra-IR is strongly and statically typed, and all computations are expressed using static single assignments (SSA). The compute language component and syntax are based on the LLVM IR [11], with extensions for coordination, memory access and parallelism. It has two components: the *Manage-IR* and the *Compute-IR*. The *Manage-IR* has semantics to instantiate *memory objects*, entities that can be the source or sink for a stream. Typically, a memory object's equivalent in software would be an array. *Stream objects* are used to express the connection between a processing element and a *memory object*.

```
;1. Pipeline with          | ;3. Coarse-grained
;   combinatorial blocks   | ;   pipeline
pipe {                     | pipe {
  instr                    |   pipeA()
  instr                    |   pipeB()
  combA()                  |   ... }
  ... }                    |
                           |
;2. Data-parallel          | ;4. Data-parallel
;   pipelines              | ;   Coarse-grained pipeline
par {                      | par {
  pipeA()                  |   pipeTop()
  pipeA()                  |   pipeTop()
  ... }                    |   ... }
                           |   ;where
                           |   pipeTop{
                           |     pipeA()
                           |     pipeB()
                           |     ... }
```

Figure 6: Design configurations in the IR currently supported by the TyTra compiler.

The *Compute-IR* describes the PE(s), which typically is a pipelined implementation of the kernel datapath. The PEs are constructed by creating a hierarchy of *functions*, which may be considered equivalent to *modules* in an HDL like Verilog. However, these *functions* are described at a higher abstraction than HDL, with a keyword specifying the parallelism for the 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 search space for an FPGA target. The currently supported set of configurations shown in Figure 6 are those suitable for our application use case, i.e. HPC applications amenable to streaming, pipelined implementation on FPGAs. As an illustration, Figure 7

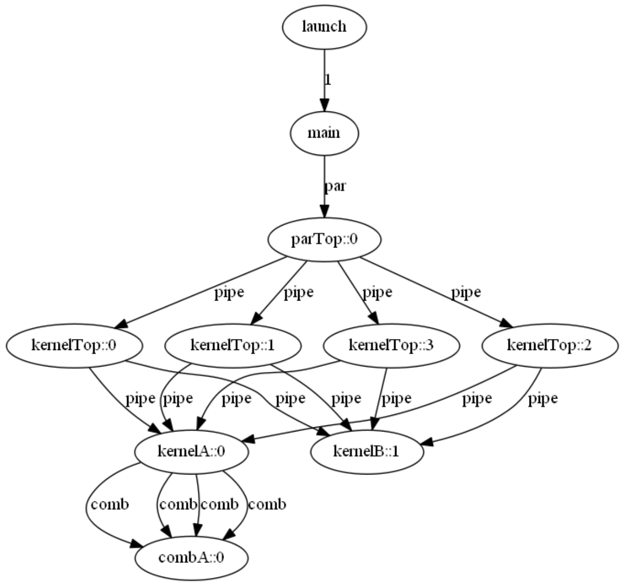


Figure 7: A typical configuration generated by the TyTra compiler showing a coarse-grained pipeline where one of the peer kernels uses a custom combinatorial function

shows the configuration tree created for multiple lanes of a coarse-grained pipeline where one of the peer kernels uses a custom combinatorial block.

5. Roofline Analysis for Evaluating Variants in the Search Space

Given such a framework as just described for representing design variants, the crucial requirement is being able to evaluate these variants for cost and performance. We use the *roofline* model inside our automated compiler as a systematic framework for evaluating design variants in the search space. Also, as it is a very visual performance model, it is useful for manual optimizations.

The roofline analysis [6] was introduced as a model to estimate the performance of a particular application on a multicore architectures. Since then, it has been been adopted for GPUs [12] as well as FPGAs [13]. The model was based on the observation that "For the foreseeable future, off-chip memory bandwidth will often be the constraining resource in system performance". The architectural constraints are captured by two *rooflines*, one representing the achievable *Computational Performance* (CP), and the other representing the reachable memory bandwidth (BW). These rooflines are plotted on a plot of performance (GFLOPS/sec) vs operational or *computational intensity*(CI), which is defined as FLOPS per byte of *DRAM* traffic. The CI captures an algorithmic feature, that is, it predicts the DRAM bandwidth needed by a kernel. The proposed model brings together two architectural features – computational performance and memory performance – and one algorithmic feature, the computational intensity. The performance of a kernel is defined in the model as follows (shown visually in Figure 8):

$Attainable\ Performance(FLOPS/sec) = min(CP, BW \cdot CI)$

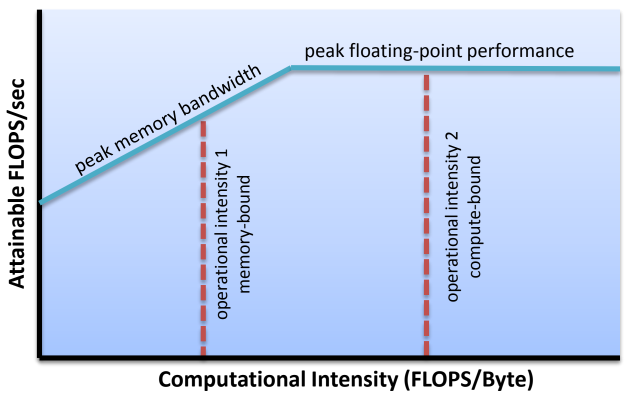


Figure 8: The original roofline model [6], showing a memory-bound and a compute-bound kernel.

5.1. *Roofline Model for FPGAs*

Silva et. al. [13] presented their extended roofline model for FPGAs. They observed that the straightforward approach of fixed rooflines that are hardware dependant no longer works for FPGAs. On FPGAs, the algorithm itself defines the architecture, and hence the rooflines for both the computation and the memory bandwidth have to be adapted for each algorithm. Alsp, instead of floating-point operations, they use byte operations (BOPS) as more suitable for FPGA targets.

They also add the *scalability* parameter (SC), which captures the replication of the PE. This scalability is determined by the available resources on the FPGA. The scalability (SC) is defined as:

$SC = \textit{Available-resources} \ / \ \textit{Resource-consumption-per-PE}$

Hence the performance roof becomes:

$CP_{FPGA} = CP_{PE} \cdot SC$

And the attainable performance:

$$Attainable\ Performance(BOPS) = min(CP_{PE} \cdot SC, CI \cdot BW)$$

6. Roofline Analysis in the TyTra Flow

We have developed a prototype compiler that can parse the TyTra-IR of a design variant, and plot its performance on the roofline model. The compiler can also emit synthesizable HDL code for the kernel pipeline (see use case diagram in Figure 2). Our cost model for estimating the variables in the roofline model is primarily an empirical model, described as follows in the context of the roofline analysis model and the models developed in section 2.

6.1. Estimating Roofline Analysis Parameters in the TyTra Flow

Our starting point was the expression developed in [13] for the attainable performance on FPGAs:

$$Attainable\ Performance(BOPS) = min(CP_{PE} \cdot SC, CI \cdot BW) \qquad (1)$$

Our main contribution is in how we estimate the four variables in Equation 1. In our framework, they are calculated on the basis of a set of parameters that depend on the target device, the kernel, and its design variant. Table 1 lists all these parameters, their key dependence (program, target hardware, design variant) and how we evaluate them in the TyTra compiler framework.

Device Peak Computational Performance: CP_{PEAK}

The horizontal *roof* in the roofline model refers to the peak computational capacity of the device, which is typically provided by the vendors. However, the FPGA adaptation of the model presented in [13] replaces this fixed roof

Param'	**Description**	**Key Dependence**	**Evaluation Method**
N_{GS}	Global-size of work-items in NDRange	Kernel	Parsing IR
N_{WOPK}	Word operations per kernel	Kernel	Parsing IR
N_{BPW}	Bytes per word	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 and device	Parsing IR
L_{PI}	Latency per instruction	Design-variant	Parsing IR
N_I	Instructions per PE	Design-variant	Parsing IR
D_V	Degree of pipeline vectorization	Design-variant	Parsing IR
N_{WPT}	Words per memory I/O tuple	Kernel	Parsing IR
Max_X	Maximum available resource of type X	Target device	Architecture description
$Util_X$	Utilization of resource of type X	Design-variant	Parsing IR

Table 1: The parameters required to calculate the four main variables of the roofline model, along with their key dependence, and the way they are evaluated in the TyTra compiler.

with a dynamic one that depends on the algorithm in addition to the device. The *roof* in their work is determined by the CP of one PE, scaled to the maximum possible in that device, and we work with their definition:

$$CP_{PEAK} = CP_{PE} \cdot SC \tag{2}$$

Computational Performance of one PE: CP_{PE}

We have developed an expression for CP_{PE} that is generic enough to accommodate the various configurations that can be created on an FPGA.

We started from the basic definition of the computational performance for one PE, CP_{PE}, which is:

$$CP_{PE} = \frac{\textit{total bytes executed per kernel instance}}{\textit{time taken}} \tag{3}$$

This expression can be expanded based on the parameters described in Table 1:

$$CP_{PE} = \frac{N_{GS} \cdot N_{WOPK} \cdot N_{BPW}}{\frac{N_{OFF}+K_{PD}}{F_D} + \frac{N_{GS} \cdot L_{PI} \cdot N_I}{F_D \cdot D_V}} \tag{4}$$

The numerator is the total number of byte operations in the kernel-instance. The first term of the denominator is the time taken to fill offset buffers and the kernel pipeline. Then the second term accounts for the time taken to execute all work-items given that the offset buffers and pipelines were full.

For most applications where $N_{GS} \gg N_{OFF} + K_{PD}$, we can use the asymptotic performance by ignoring the time taken to fill offset buffers and kernel pipeline in Equation 4. This gives us the simplified expression:

$$CP_{PE} = \frac{F_D \cdot N_{WOPK} \cdot N_{BPW} \cdot D_V}{L_{PI} \cdot N_I} \tag{5}$$

Processing Element Scaling: SC

We restrict the scaling SC of the PE up to 80% utilization of any of the four FPGA resources, as 100% utilization of FPGAs is generally not possible [14]. We need an estimate of device utilization by the kernel PE as well as the *base platform* peripheral logic. This is taken up in §6.2. Once we have these estimates, the scaling is determined by whichever one of the four resources on the FPGA runs out first:

$$SC = min(\frac{Max_{LE}}{Util_{LE}}, \frac{Max_{FF}}{Util_{FF}}, \frac{Max_{DSP}}{Util_{DSP}}, \frac{Max_{RAM}}{Util_{RAM}}) \tag{6}$$

Maximum Attainable Bandwidth: BW

The *peak* bandwidth to host and global memory is typically available from vendor datasheets. However, an estimate of *achievable* memory bandwidth is more relevant, and a must for the roofline model. This is especially relevant in FPGAs where the performance tends to be *memory-bound* (see Figure 8). The distinction of forms of memory execution as presented in §2.3 is relevant here. *Form B* is the most common pattern, where the bandwidth of concern would be the DRAM (global memory) bandwidth. Our approach for making this estimate is discussed in §6.3.

Computational Intensity: CI

This is a critical parameter of the model that ties characteristics of the algorithm to the architecture. In our framework, it is straightforward to calculate it from the parameters available to us inside our framework:

$$CI = \frac{N_{WOPK}}{N_{WPT}} \tag{7}$$

We have now presented our approach to calculating all four terms in Equation 1 needed to estimate the *attainable performance*. Two aspects however need further elaboration and are discussed following: estimating device utilization (to calculate the scaling factor SC), and achievable bandwidth to the DRAM.

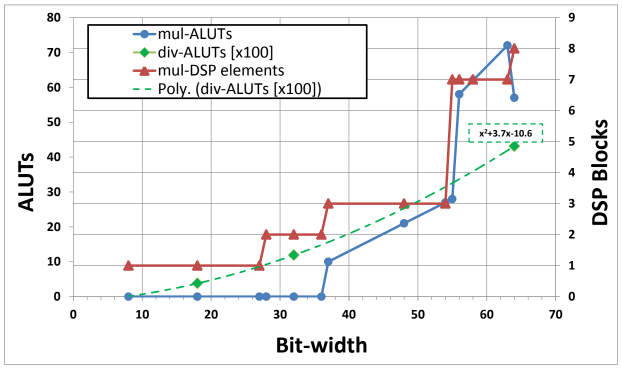


Figure 9: ALUTs used in unsigned integer division (see polynomial regression), and ALUTs and DSP-elements used in unsigned integer multiplication, on a Stratix-V device.

6.2. Resource-Utilization Cost Model

The scaling of a PE, and even the question of whether or not a single PE can fit in the FPGA, is determined by the available on-chip logic and memory resources. The resources are taken up by (1) the kernel PE, and (2) the peripheral logic or *base platform* that connects this PE to the memories via data streams.

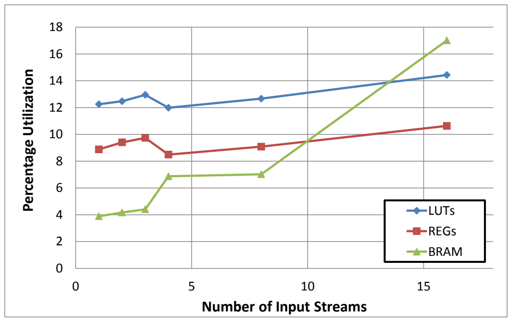
6.2.1. Estimating Kernel PE Resource Utilization

Our observation is that the regularity of FPGA fabric allows a simple empirical model to be built for most primitive instructions. As an example, consider the trend-line for LUT requirements against bit-width for integer division shown in Figure 9. It was generated from 3 data points (18, 32 and 64 bits) from micro-benchmark experiments on an Altera Stratix-V device. We can now use it for polynomial regression and interpolation, e.g., for 24 bits, and get an estimate of 654 LUTs, which compares favourably with the actual figure of 652 LUTs. A multiplier requires two different kinds of resources: DSP-elements and LUTs. Both these resources show a piece-wise-linear behaviour with respect to the bit-width, with clearly identifiable points of discontinuity, also shown in Figure 9. This results in a relatively trivial empirical model. Other primitive 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 kernel by accumulating the cost of individual IR instructions and the structural information implied in the *type* of each IR *function.* With reference to Figure 7, if a kernel is identified as a *pipe*, then each of its instruction requires dedicated resource on the FPGA, along with pipeline registers. If we have a *par* function with child *pipe* functions, then each of the children requires dedicated logic resources.

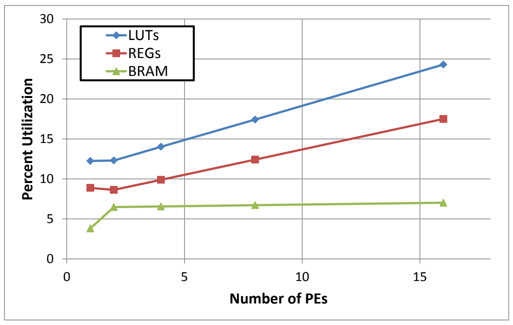
6.2.2. Estimating Base Platform Resource Utilization

An estimate of the overhead of the peripheral logic or *shell* provided by a vendor *base platform* is crucial if we want to estimate the maximum scaling. For small kernels especially, this will dominate the resource consumption. This estimate is somewhat tricky however as the internal structure of the base platform is not visible to us.

Our use-case of primarily interfacing with the host or DRAM via data streams simplifies this estimate. We have designed a simple micro-benchmark the Altera OpenCL base platform where we instantiate a minimalist pass-through kernel, and synthesize the shell with varying the number of streams or varying number of PEs. Linear regression gives us the expressions we can insert into our compiler for generating the resource estimates of the shell, which are then added to the estimates of the kernel. The total resource utilization estimates are then used by our compiler for calculating the maximum



(a) For different number of input streams.



(b) For different values of PE scaling (1 input stream in all cases).

Figure 10: The resource cost of the base-platform that creates streams feeding the TyTra-generated PE. One output stream in all cases, and word-size is 32 bits. Target is a Stratix-V device programmed with Altera-OpenCL.

possible PE scaling for a given kernel.

Our solution for estimating the base platform resources is based on two observations: first, changing the number of input streams has a marginal impact on the utilization of registers and LUTs, but significant impact on the utilization of BRAM (See Figure 10a); second, scaling the PE has significant impact on the utilization of registers and LUTs, but marginal impact on that of the on-chip BRAM (See Figure 10b). We hence use simplifying assumptions[5] to avoid the complete matrix of synthesis experiments (all possible combinations of number of PEs and streams). Since the kernel compilation is independent of the host array (stream) sizes, this has no impact on the

[5]Even with a ±2% margin of error, we can assume the register and LUT usage to be independent of number of input streams, and the BRAM usage to be independent of PE scaling (apart from one outlier).

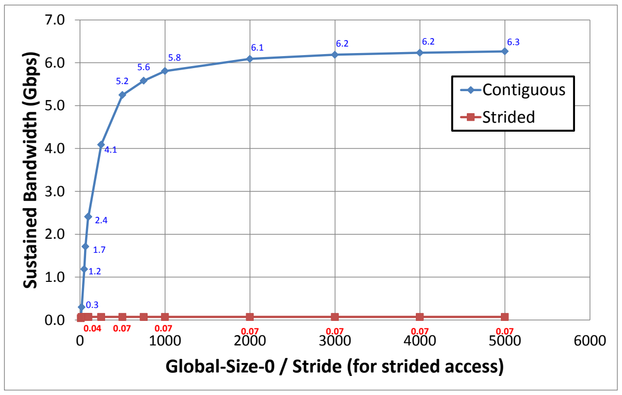


Figure 11: An empirical model of the dependency of sustained bandwidth on data size and contiguity of data, based on 32-bit integers. The horizontal axis represents one dimension of a square array, so it is also the stride in case of strided access. Results are based on Alpha-Data's ADM-PCIE-7V3 board with a Xilinx Virtex 7 device. The performance can be improved with optimizations.

resource utilization of the base platform.

6.3. Estimating Sustained Memory 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 datasheets, the *sustained* bandwidth for various streams in a particular design varies with the access pattern, size, and other parameters as well. We performed a set of experiments by extending the STREAM benchmark [15] to OpenCL, based on the work done in [16] for GPUs. 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. The results are shown in Figure 11. They highlight the importance of taking into account the factors effecting the sustained bandwidth for any realistic cost models.

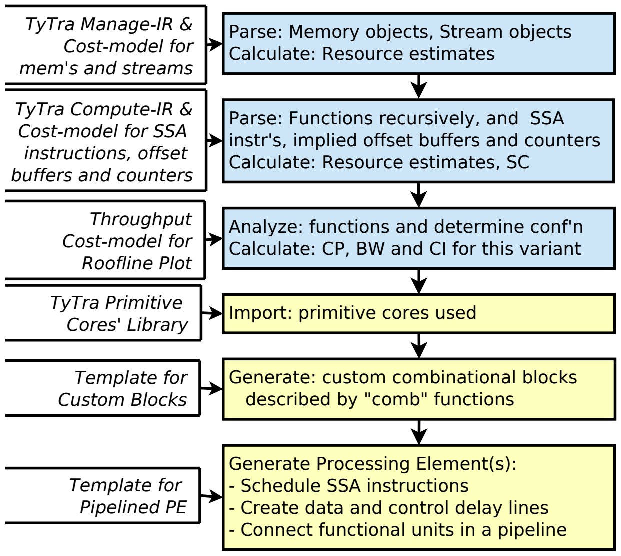

TyTra Manage-IR & Cost-model for mem's and streams
Parse: Memory objects, Stream objects
Calculate: Resource estimates
TyTra Compute-IR & Cost-model for SSA instructions, offset buffers and counters
Parse: Functions recursively, and SSA instr's, implied offset buffers and counters
Calculate: Resource estimates, SC
Throughput Cost-model for Roofline Plot
Analyze: functions and determine conf'n
Calculate: CP, BW and CI for this variant
TyTra Primitive Cores' Library
Import: primitive cores used
Template for Custom Blocks
Generate: custom combinational blocks described by "comb" functions
Template for Pipelined PE
Generate Processing Element(s):
- Schedule SSA instructions
- Create data and control delay lines
- Connect functional units in a pipeline

Figure 12: 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.

We have incorporated this empirical model into our compiler, and continue developing the stream benchmark.

7. Using the Roofline Cost Model in the TyTra Compiler – an Illustration

We have developed a prototype compiler that accepts a design variant in TyTra-IR, estimates its cost and performance and plots it on the roofline model, and if needed, generates the HDL code for it. The flow is shown in Figure 12.

We created some design variants of the SOR kernel generated by type

```
 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, %cn2l
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 }
```

(a) Single PE.

```
 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 }
```

(b) Scaled (x4) PEs.

Figure 13: Abbreviated TyTra-IR code for the two variants of the SOR kernel.

transformations as discussed in 3. Figure 13a 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 14 shows the realization of the kernel as a pipeline in the FPGA as generated by the TyTra compiler. The same SOR example can be expressed in the IR to represent *data parallelism* by adding multiple PE *lanes*, corresponding to a reshaped data along e.g four rows, by encapsulating multiple instances of the kernel-pipeline function shown in Figure 13a into a top-level function of type `par`, and creating multiple stream objects to service each of these parallel kernel-pipelines. This variant's IR is shown in Figure 13b.

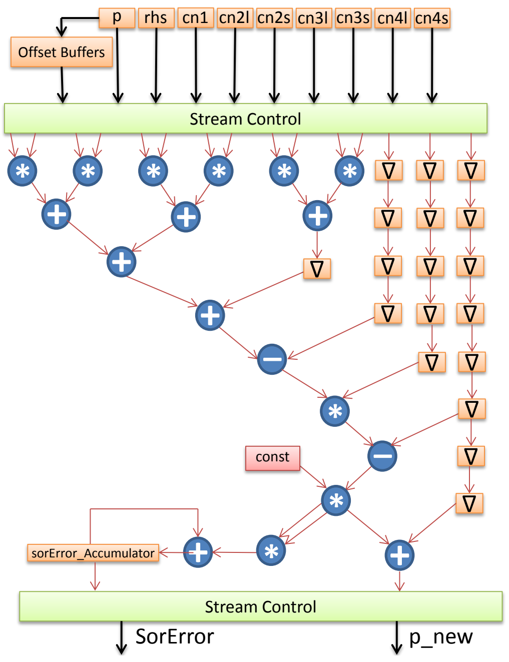


Figure 14: Pipelined datapath of the SOR kernel as generated by the TyTra compiler. Only pass-through pipeline buffers are shown; all functional units have pipeline buffers as well. The blocks at edges refer to on-chip memory.

7.1. Roofline Analysis of the Design Variants using our 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. This high level translation transforms to scaling the number of PEs.

Figure 15 shows the roofline plots of 5 variants thus generated. The first variant has a single PE, and the computational intensity CI intersects the computational roof CP_{PE} of the design. So this variant is *compute-bound*. With the performance well below the peak computational capacity of the device for *this algorithm* (the blue dotted line), there is clearly room for improvement.

In the next variant, there are now 2 PEs, and since the design is still compute-bound, we see a proportional increase in performance. For the next variant, with PEs scaled to 4, the computational roof moves even further upwards, and we are almost at the intersection of compute and bandwidth roofs. This is the ideal situation as we are making the best use of both the available computation and bandwidth capabilities. One can predict that further scaling will not yield any improvement as the design moves into the memory-bound region. This is confirmed by the two more roofline graphs

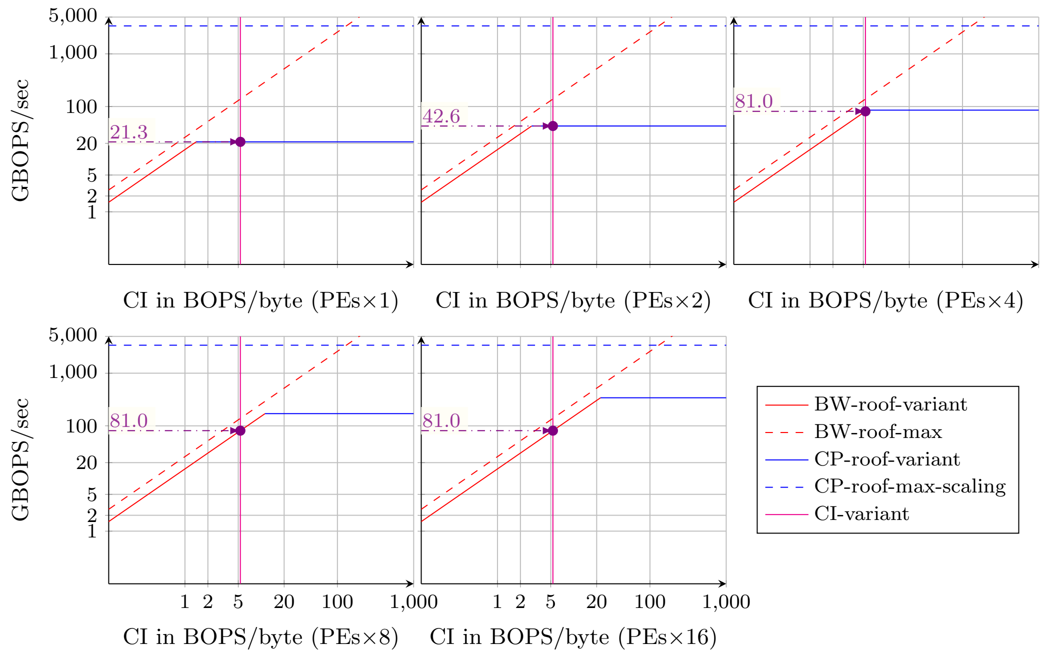


Figure 15: Evaluation of variants for the SOR kernel generated by applying the PE-scaling transformations. We get dividends from scaling until we hit the memory-wall at a scaling of 4.

showing the PEs scaled to 8 and 16 respectively, yet the performance constrained at what was estimated for a scaling of 4.

The limit of PE scaling itself is determined by our device utilization model, and is used to determine the CP_{PEAK} (blue-dotted line). If we do not encounter the memory roof first, we can in theory scale the PE until we reach CP_{PEAK}.

We can see from the roofline plots that we can improve the performance of this design by either improving the sustained bandwidth to the memory which is still below peak, or by carrying out an algorithmic or memory buffering optimization that increases the computational intensity. If, for another design, the limiting factor was CP_{PEAK}, then the optimization focus would be on reducing device utilization, e.g. by using reduced precision arithmetic.

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 would also like to point out that while the cost-model does facilitate the evaluation of the search-space by providing a light-weight high-level route to estimates, on its own it does not *simplify* the search-space. The issue of simplifying the design search-space is important in its own right, as it can very quickly explode to an unmanageable number even for relatively small applications. We are currently working on an approach to simplify the search-space, but that is outside the scope of this paper.

7.2. Accuracy of the cost model

Preliminary results on relatively small but realistic scientific kernels have been very encouraging. We evaluated the estimated vs actual[6] utilization of resources for the kernel pipelines, and throughput measured in terms of cycles-per-kernel-instance in Table 2. 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 [17], used to estimate processor temperature based on an architectural floorplan and simulated power measurements; (3) The *lavaMD* molecular dynamics application also from Rodinia, which calculates particle potential and relocation due to mutual forces between particles within a large 3D space.

These results confirm that an IR defined at an appropriate abstraction will allow quick estimates of cost and performance that are accurate enough for relative comparison of design variants in the search space.

Currently our cost model has two limitations which we are investigating. First, it does not anticipate the optimizations done in the relevant synthesis tool. We will however require better visibility into the behaviour of synthesis tools like Quartus if we are to have a realistic, nuanced model for such synthesis optimizations, and we will explore this in the future. Secondly, the accuracy we see in Table 2 is for the *kernel* estimates only. When we compare

[6] The *actual* resource utilization figures are based on full synthesis, and *actual* cycle-counts are from RTL simulations. The design entry for these experiments is in Verilog RTL. The RTL is generated from TyTra-IR description using our back-end compiler.

Kernel		LUT	REG	BRAM	DSP	CPKI
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 2: The estimated vs actual[6] performance and utilization of resources, the former measured in terms of cycles-per-kernel-instance (CPKI), for the kernel of three scientific applications. Percentage errors also shown. All are executed as *Form-C* implementations (see §2.3).

the performance of a complete solution (kernel logic generated by tytra, shell logic based commercial HLS tool) with the TyTra estimate, the accuracy is relatively lower (see Figure 17), though we still achieve the primary purpose, i.e., finding the best design variant.

8. Case Study: Comparison of a TyTra-generated solutions against HLS tools

A working solution using an FPGA accelerator requires a base platform or *shell* on the FPGA to deal with off-chip IO and other peripheral functions, along with an API for accessing the FPGA accelerator. We use a commercially available frameworks to implement this shell, and the TyTra-generated HDL code for the kernel. We will now compare this *hybrid* approach against the *baseline* using only the HLS tool, for two different frameworks.

Maxeler's MaxJ [1] is an HLS design tool for FPGAs, and provides a Java meta-programming model for describing computation kernels and connecting data streams between them. The experimental setup for the experiments on the Maxeler framework is shown in Figure 16.

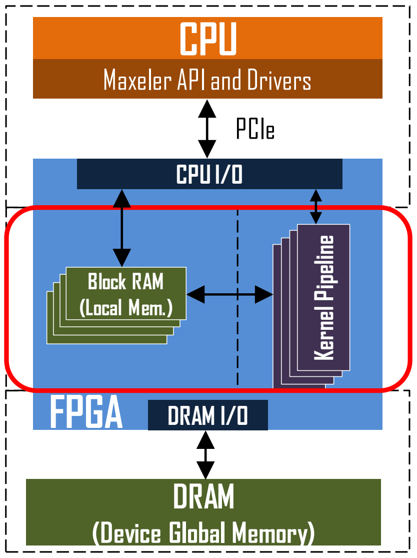


Figure 16: The Maxeler-TyBEC hybrid solution. The dotted line identifies what is programmed using the Maxeler HLS tool (shell). The solid/red line identifies the logic programmed with TyTra generated code (kernel). The overlap indicates that stream generation from on-chip Block-RAMs can be done by either.

The CPU implementation (*cpu*) is compiled with `gfortran -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-level type transformation that introduced parallelism ($4\times$PEs) 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). .

8.1. Performance Comparison

The performance comparison of Maxeler-only (*fpga-maxJ*) and Maxeler-TyTra hybrid (*fpga-tytra*) is shown in Figure 17. Note that *fpga-maxJ* could in principle be optimized manually to achieve a similar performance as *fpga-tytra*, but we deliberately use an unoptimized baseline for *fpga-maxJ*. Our contention is that by using our approach, one can obviate the need to carry out manual optimizations in an HLS tool like Maxeler. Hence our competition is an unoptimized HLS solution.

Apart from the smallest grid-size, *fpga-tytra* consistently outperforms *fpga-maxJ* as well as *cpu*, showing up to 3.9× and 2.6× 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 point 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 2.75× faster. These performance results clearly indicate that a straightforward implementation on an HLS tool will not be optimal and manual effort would be required; the TyTra flow can automate this.

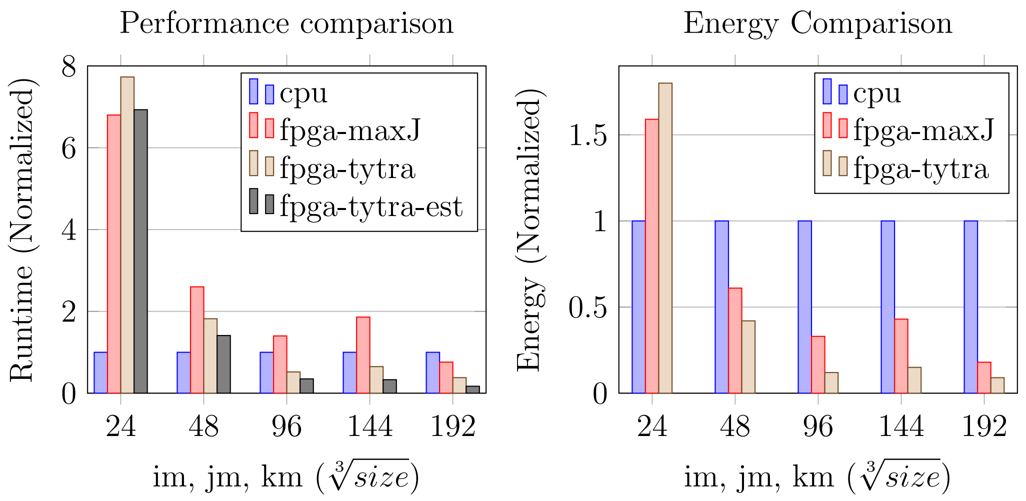


Figure 17: Comparing performance and energy differential of the SOR kernel for different sizes of grid, normalized against the CPU-only solution. The figures are for 1000 iterations of the kernel. Setup: Intel-i7 quad-core processor at 1.6GHz, 32 GB RAM, and an Altera Stratix-V-GSD8 FPGA.

If we look at the (the *fpga-tytra-est* column) showing the performance predicted by our cost model, we can see they are not accurate as the kernel-only estimates in Table 2. The introduction of the shell adds a degree of inaccuracy to the performance estimate, which in the worst case in this particular example is off by 2.35×. However, the use-case of finding the best variant from a search-space is still very much applicable as these results show.

To further qualify our approach, we compared it against another commercial HLS tool, Altera-OpenCL (AOCL), using a 2D SOR kernel. The run-time, normalized against a baseline CPU implementation[7], is shown in

[7]A faster Xeon CPU along with higher data locality for 2D stencil would explain why the CPU performs much better than the FPGA in this experiment, as compared to the

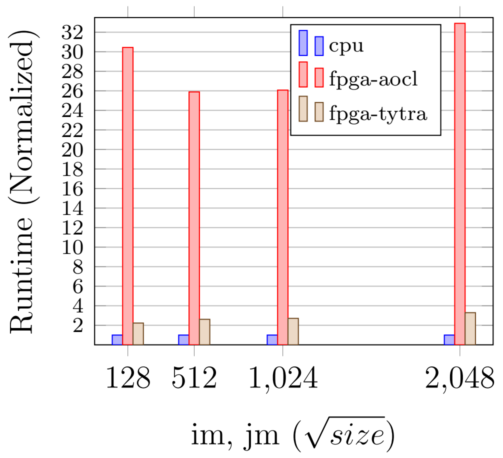


Figure 18: Runtime of the 2D-SOR kernel for AOCL-only and AOCL-TyTra hybrid for different sizes of grid, normalized against the CPU-only solution. Setup: Intel Xeon E5 quad-core at 2.4 GHz, 64GB RAM, and an Altera Stratix-V-GSD5 FPGA.

Figure 18.

The advantage of our approach to FPGA programming is starkly demonstrated in this experiment, where – using an approach similar to the one described for the Maxeler-TyTra experiment – the Tytra solution (with an AOCL shell) yields an order of magnitude better performance than the AOCL-only solution on the same FPGA[8].

one on Maxeler.

[8]As opposed to the *fpga-tytra* or the *fpga-maxj* solution, we were unable to use on-chip buffers for stencil data in the *fpga-aocl* solution, as AOCL failed to synthesize within available resources (see Figure 19. Hence the *fpga-aocl* solution accesses the main memory for every data-point in the stencil, which affects its performance.

8.2. Energy Comparison

For the energy figures, we used the actual power consumption of the host+device measured at the node's power socket using a WattsUp 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 **??**, FPGAs very quickly overtake CPU-only solutions, and *fpga-tytra* solution shows up to 11× and 2.9× 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.

8.3. Resource Utilization Comparison

Previous results show the optimized hybrid TyTra solution compared with baseline, unoptimized solutions using HLS tools. Here we compare what happens when we compare like-for-like variants, that is the *same* optimization using the two approaches, with the AOCL tool as the baseline. The results are shown in Figure 19 [9] We also compare the effect of varying the array sizes for both cases, which effect the size of internal buffers for stencil data, and hence effect resource utilization.

We can see that the resource utilization is comparable for the baseline solution with one PE. However, when we optimize the design by replicating the PEs, then the AOCL-only solution – which implements the optimization by

[9] Full synthesis results are used, apart from cases where design could not synthesize because required resources exceeded availability, in which case we used estimated resources emitted by AOCL.

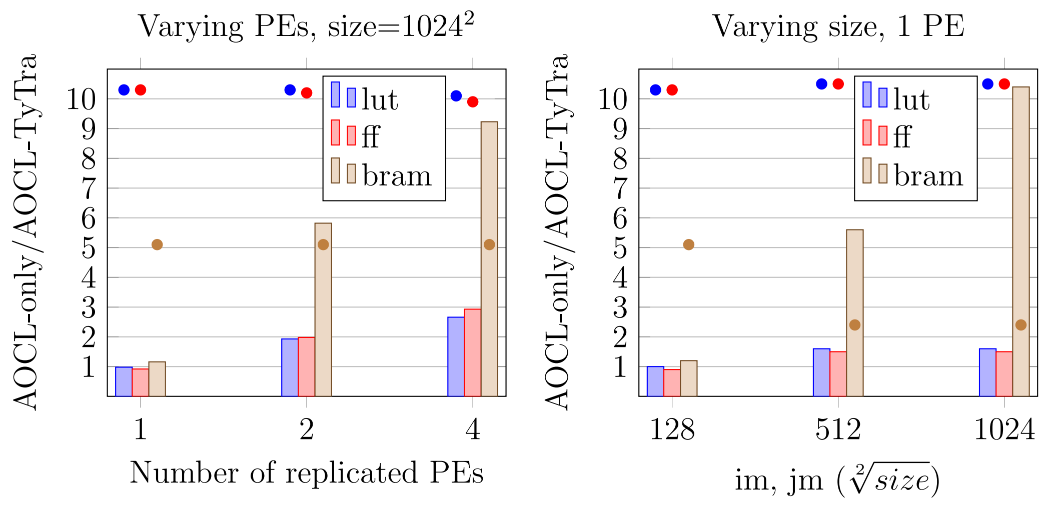


Figure 19: Normalized FPGA resource utilization, AOCL-only against AOCL-TyTra, for equivalent design variants. The dots represent maximum available resource, so if the dot is inside a plot, it means the required resource exceeds availability and did not synthesize.

changing the number of *compute-units* in the OpenCL code – takes up much more resources than the AOCL-TyTra solution, especially in the utilization of BRAMs. A similar observation is made for the case when we fix the design to one PE, and change the size of data[10] . In fact, the AOCL-only solution do not even synthesize in most cases as the available BRAM resources are exceeded. These results make a strong case for using our flow not just to generate or evaluate the variants, but also to *implement* them based on our generated HDL code.

[10]We have found that the Maxeler framework is more efficient at using resources even when the PE-replication optimization is implemented entirely in Maxeler. However, that requires more programming effort as compared to simple pragma based optimization in AOCL.

9. Related Work

We can discuss related work from three different perspectives, i.e., in relation to: raising the design-entry abstraction above conventional high-level languages in general, high-level programming approaches specific to FPGAs, and cost/performance models developed for FPGAs.

Our observation that there is a requirement for a higher abstraction design entry than conventional high-level languages is not novel in itself. For example, researchers have proposed algorithmic skeletons to separate algorithm from architecture-specific parallel programming [18]. SparkCL [19] brings increasingly diverse architectures, including FPGAs, into the familiar Apache Spark framework. Domain-specific languages (DSL) are another way to raise the design abstraction within the scope of a particular application domain, and numerous examples can be found for FPGAs. For example, *FSMLanguage* for desiging FSMs [20], and *CLICK* for networking applications [21].

There is considerable work that deals specifically with Programming FPGAs using conventional high-level programming. 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, 22], (2) compiler optimizations are limited to improving the overall architecture already specified by the programmer, with no real *architectural* exploration [1, 2, 22, 4],

(3) solutions are based on creating a soft microprocessors on the FPGA and are not optimized for HPC [4, 23], (4) the exploration requires evaluation of variants that take a prohibitively long amount of time [22], or (5) the flow is limited to very specific application domain e.g. for image processing or DSP applications [24]. The *Geometry of Synthesis* project [25] is more similar than others, 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. 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.

Comparison with work related to cost models is another dimension. We have used the work described in [26] on extending the roofline analysis for FPGAs. However, our work is fundamentally different as we are only using the abstractions offered in the roofline analysis model; the manner in which we actually generate and represent variants, and estimate the cost parameters are entirely novel contributions of our work. Kerr et. al. [27] have developed a performance model for CUDA kernels on GPUs based on empirical evaluation of a number of existing applications. Park et. al. [28] create a performance model for estimating the effects of loop transformation on FPGA designs. Since our *type transformations* can be viewed as a different abstraction for achieving similar outcomes, so there is a strong parallel between their work and ours. However, their work seems to focus on loop-unrolling, and is fundamentally different in terms of design entry and variant generation. Dent et. al. presented cost models for area, time and power [29] which is based on the MATLAB-based *FANTOM* tool. Their approach of using an empirical model based on actual synthesis experiments resonates with our approach. However, their estimation model works on generated HDL whereas we make estimates at a higher abstraction, and our approach of generating and evaluating variants is fundamentally different. Reference [30] presents another cost estimation approach comparable to ours, where regression analysis is done on empirical data from synthesis tools to create resource estimation models, which are then used at compile-time on FPGA designs programmed at a high-level using the SA-C language. Their work does not estimate performance however, and the overall context is very different from the TyTra flow. More recently, [31] have presented an analytical model, but their focus is on estimating dynamic and static power of various architectures.

10. Conclusion

FPGAs are increasingly being used in HPC for accelerating scientific computations. While the typical route to implementation is the use of HLS frameworks like Maxeler or OpenCL, they may not necessarily expose the parallelism opportunities on an FPGA in a straightforward manner. Tuning designs to exploit the available FPGA resources on these HLS tools is possible but still requires considerable effort and expertise. We have presented an original flow that 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 have developed abstractions used to create a structured cost model, and discussed our use of the roofline model to automatically cost and evaluate design variants. We have shown how our empirical approach to costing designs represented at an intermediate level allows us to calculate all parameters required to plot the performance of a design variant on the roofline model.

We have illustrated the use of our roofline-based cost model to evaluate different variants of the SOR kernel. The accuracy of the cost model was shown across three different kernels. A case study based on the SOR kernel from a real-world weather model 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 give better performance than the baseline solutions on both Maxeler and Altera-OpenCL. In addition, we showed that even if the design optimizations were to be programmed in the HLS tools, our approach gives much more efficient resource utilization on the basis of its generated HDL code.

We are currently in the process of automating the generation of design variants from high-level code. Also, we are working to extend our cost model code generator to floating point as well more complex arithmetic operations. We are also validating the cost model and code generator with larger and more complex kernels, while expanding the set of available transformations. Our approach to extending the cost-model is to keep evolving the TyTra-IR as we experiment with more applications and transformations, and ensuring the cost-model remains complete by providing an analytical or an empirical model for all valid TyTra-IR instructions. Ultimately, our work aims to provide 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. We are also working on evolving our flow to include legacy code written in languages typically used for scientific computing like Fortran or C.

Funding. This work was supported by EPSRC [grant number EP/L00058X/1].

[1] O. Pell, V. Averbukh, Maximum performance computing with dataflow engines, Computing in Science Engineering 14 (4) (2012) 98–103. `doi:10.1109/MCSE.2012.78`.

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

[3] The Xilinx SDAccel Development Environment (2014).
URL `http://www.xilinx.com/publications/prod_mktg/sdx/sdaccel-backgrounder.pdf`

[4] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, 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, ACM, New York, NY, USA, 2011, pp. 33–36.

[5] S. Rul, H. Vandierendonck, J. DHaene, K. De Bosschere, An experimental study on performance portability of opencl kernels, in: Application

Accelerators in High Performance Computing, 2010 Symposium, Papers, 2010.

[6] S. Williams, A. Waterman, D. Patterson, Roofline: An insightful visual performance model for multicore architectures, Commun. ACM 52 (4) (2009) 65–76. doi:10.1145/1498765.1498785.
URL http://doi.acm.org/10.1145/1498765.1498785

[7] The OpenCL Specification (2015).
URL https://www.khronos.org/registry/cl/

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

[9] E. Brady, Idris, a general-purpose dependently typed programming language: Design and implementation, Journal of Functional Programming 23 (2013) 552–593. doi:10.1017/S095679681300018X.

[10] W. Vanderbauwhede, Inferring Program Transformations from Type Transformations for Partitioning of Ordered Sets (2015). arXiv:arXiv:1504.05372.
URL http://arxiv.org/abs/1504.05372

[11] Chris Lattner and Vikram Adve, The LLVM Instruction Set and Compilation Strategy, Tech. Report UIUCDCS-R-2002-2292, CS Dept., Univ. of Illinois at Urbana-Champaign (Aug 2002).

[12] K.-H. Kim, K. Kim, Q.-H. Park, Performance analysis and optimization of three-dimensional FDTD on GPU using roofline model,

Computer Physics Communications 182 (6) (2011) 1201 – 1207. doi:http://dx.doi.org/10.1016/j.cpc.2011.01.025.
URL http://www.sciencedirect.com/science/article/pii/S0010465511000452

[13] B. da Silva, A. Braeken, E. H. D'Hollander, A. Touhafi, Performance modeling for fpgas: Extending the roofline model with high-level synthesis tools, Int. J. Reconfig. Comput. 2013 (2013) 7:7–7:7. doi:10.1155/2013/428078.
URL http://dx.doi.org/10.1155/2013/428078

[14] R. Tessier, H. Giza, Balancing logic utilization and area efficiency in fpgas, in: Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications, FPL '00, Springer-Verlag, London, UK, UK, 2000, pp. 535–544.
URL http://dl.acm.org/citation.cfm?id=647927.739548

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

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

[17] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: Workload

Characterization, 2009. IISWC 2009. IEEE International Symposium on, 2009, pp. 44–54. `doi:10.1109/IISWC.2009.5306797`.

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

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

[20] J. Agron, Domain-Specific Language for HW/SW Co-design for FPGAs, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 262–284. `doi:10.1007/978-3-642-03034-5_13`.
URL `http://dx.doi.org/10.1007/978-3-642-03034-5_13`

[21] C. Kulkarni, G. Brebner, G. Schelle, Mapping a domain specific language to a platform fpga, in: Proceedings of the 41st Annual Design Automation Conference, DAC '04, ACM, New York, NY, USA, 2004, pp. 924–927. `doi:10.1145/996566.996811`.
URL `http://doi.acm.org/10.1145/996566.996811`

[22] J. e. a. Keinert, Systemcodesigner;an automatic esl synthesis approach by design space exploration and behavioral synthesis for streaming applications, ACM Trans. Des. Autom. Electron. Syst. 14 (1) (2009) 1:1–1:23.

[23] K. Keutzer, K. Ravindran, N. Satish, 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, 2005, pp. 273–278. doi:10.1145/1084834.1084903.

[24] M. Kaul, R. Vemuri, S. Govindarajan, 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, ACM, New York, NY, USA, 1999, pp. 616–622. doi:10.1145/309847.310010.

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

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

[27] A. Kerr, G. Diamos, S. Yalamanchili, Modeling gpu-cpu workloads and systems, in: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU 10, ACM, New York, NY, USA, 2010, p. 3142. doi:http://doi.acm.org.prx.library.gatech.edu/10.1145/1735688.1735696.
URL http://doi.acm.org.prx.library.gatech.edu/10.1145/1735688.1735696

[28] J. Park, P. C. Diniz, K. R. S. Shayee, Performance and area modeling

of complete fpga designs in the presence of loop transformations, IEEE Transactions on Computers 53 (11) (2004) 1420–1435. doi:10.1109/TC.2004.101.

[29] L. Deng, K. Sobti, Y. Zhang, C. Chakrabarti, Accurate area, time and power models for fpga-based implementations, Journal of Signal Processing Systems 63 (1) (2011) 39–50.

[30] D. Kulkarni, W. A. Najjar, R. Rinker, F. J. Kurdahi, Compile-time area estimation for lut-based fpgas, ACM Transactions on Design Automation of Electronic Systems (TODAES) 11 (1) (2006) 104–122.

[31] H. Mehri, B. Alizadeh, Analytical performance model for fpga-based reconfigurable computing, Microprocessors and Microsystems 39 (8) (2015) 796 – 806. doi:http://dx.doi.org/10.1016/j.micpro.2015.09.009.