2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

Automatic Generation of Efficient Accelerators for Reconfigurable Hardware

David Koeplinger
Stanford University
dkoeplin@stanford.edu

Raghu Prabhakar
Stanford University
raghup17@stanford.edu

Yaqi Zhang
Stanford University
yaqiz@stanford.edu

Christina Delimitrou
Stanford University
Cornell University
cdel@stanford.edu

Christos Kozyrakis
Stanford University
EPFL
kozyraki@stanford.edu

Kunle Olukotun
Stanford University
kunle@stanford.edu

***Abstract*—Acceleration in the form of customized datapaths offer large performance and energy improvements over general purpose processors. Reconfigurable fabrics such as FPGAs are gaining popularity for use in implementing application-specific accelerators, thereby increasing the importance of having good high-level FPGA design tools. However, current tools for targeting FPGAs offer inadequate support for high-level programming, resource estimation, and rapid and automatic design space exploration.**

We describe a design framework that addresses these challenges. We introduce a new representation of hardware using parameterized templates that captures locality and parallelism information at multiple levels of nesting. This representation is designed to be automatically generated from high-level languages based on parallel patterns. We describe a hybrid area estimation technique which uses template-level models and design-level artificial neural networks to account for effects from hardware place-and-route tools, including routing overheads, register and block RAM duplication, and LUT packing. Our runtime estimation accounts for off-chip memory accesses. We use our estimation capabilities to rapidly explore a large space of designs across tile sizes, parallelization factors, and optional coarse-grained pipelining, all at multiple loop levels. We show that estimates average 4.8% error for logic resources, 6.1% error for runtimes, and are 279 to 6533 times faster than a commercial high-level synthesis tool. We compare the best-performing designs to optimized CPU code running on a server-grade 6 core processor and show speedups of up to 16.7×.

I. Introduction

Over the past few years, the computing landscape has seen a paradigm shift towards specialized architectures [1, 2, 3, 4]. Customized accelerators, implemented as application specific integrated circuits (ASICs) efficiently perform key kernel computations within larger applications to achieve orders of magnitude improvements in performance and energy efficiency compared to programmable processors [5]. However, such improvements typically require sacrificing flexibility. Once fabricated, an ASIC's custom datapath can no longer be modified to meet new requirements. ASICs also have high non-recurring engineering (NRE) costs associated with manufacturing,

Reconfigurable fabrics such as field-programmable gate arrays (FPGAs) offer a promising alternative to ASIC-based accelerators due to their reconfigurability and customizability, even if these benefits come at a price [6]. FPGAs are increasingly gaining traction in the industry as mainstream accelerators. Microsoft [7] and Baidu [8] have successfully deployed FPGA-based accelerators in a commercial setting to accelerate web search and deep neural networks. Intel [9] is actively working on integrating an FPGA with a processor on a heterogeneous motherboard. The acquisition of Altera by Intel, and new startups [10] working on using FPGAs as accelerators for datacenters suggest that future systems will incorporate reconfigurable logic into their design. As a result, FPGAs will play a crucial role in the space of customizable accelerators over the next few years. This places a greater importance on FPGA programmability and automated tools to generate efficient designs that maximize application performance on a given reconfigurable substrate.

Designing an efficient accelerator architecture involves balancing compute with on-chip and off-chip memory bandwidth requirements to avoid resource bottlenecks. This is irrespective of whether the accelerator is implemented as an ASIC or on an FPGA. This process involves navigating a large multi-dimensional design space with application-level, architectural and microarchitectural parameters. The optimal set of parameters depends on the inherent parallelism and data locality in the application, as well as the available hardware resources. The design space for FPGAs is further complicated by the heterogeneous nature of available FPGA resources, which include units such as look-up tables (LUTs), block RAMs (BRAMs), flip flops (FFs) and digital signal processing units (DSPs). As a result, hardware accelerator design is an inherently iterative process which involves exploring a large design space for even moderately complex accelerators. Exhaustive or manual exploration of this space would be impractical for all but the simplest of designs, suggesting that efficient FPGA accelerator design requires support from high-level tools for rapid modeling and design space exploration.

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.20

Unfortunately, current FPGA design tools support relatively primitive programming interfaces that require extensive manual effort [11]. Designing an FPGA accelerator typically involves an architecture description in RTL or a C-based high-level language coupled with ad-hoc tools to explore the design space, potentially involving multiple logic synthesis runs. The long turn-around times from logic synthesis tools, which are on the order of several hours per design, makes it infeasible for them to be included in the iterative design process. High-level synthesis tools [12, 13, 14] raise the level of abstraction and provide some support for pre-place-and-route analysis. However, high-level synthesis tools do not capture many important design points, such as coarse-grained pipelining and nested parallelism, and generally use simple memory models that do not involve modeling off-chip memory accesses [15].

This paper presents a practical framework for automatic generation of efficient FPGA accelerators. Figure 1 describes our overall system architecture. The input to our system is an application described in a high-level language using high-level parallel patterns like *map*, *reduce*, *filter*, and *groupBy* [16]. Parallel patterns serve the dual purpose of raising the level of abstraction for the programmer [17, 19], and providing richer semantic information to the compiler [20]. These constructs are then automatically lowered into our hardware definition language (HDL) that explicitly captures information on parallelism, locality, and memory access pattern at all levels of nesting using parameterizable architectural templates (Step 1 in Figure 1). Step 1 performs high-level optimizations like loop fusion and tiling transformations. The output of step 1 is a tiled representation of the input design expressed in our HDL. Note that tiling here includes both loop and data tiling.

We characterize each template and construct resource models to provide quick and accurate estimates of cycle count and FPGA resource utilization for a given set of design parameters, including modeling off-chip memory transfers. The estimators guide a design space exploration phase (Steps 2–4 in Figure 1) which navigates a large space to produce a set of optimized parameters. Finally, we integrate hardware generation into the design flow so that optimized designs can be automatically generated, synthesized and run on a real FPGA (Steps 5–7 in Figure 1). We synthesize hardware by automatically generating MaxJ, a low-level Java-based hardware generation language from Maxeler Technologies [21]. Step 1 in Figure 1 has been described in previous work [22]. Steps 2–7 are the focus of this paper.

We make the following contributions in this paper:

- We define an intermediate representation called *Delite Hardware Definition Language*, or *DHDL*. DHDL defines a set of parameterizable architectural templates to describe hardware. Templates capture specific types of memory accesses and parallel compute patterns, and can be composed to naturally express parallelism at multiple levels in the design. The templates are designed such that applications expressed using high-level parallel patterns can be mapped to these templates in a straightforward way.
- We provide quick estimates of cycle count and FPGA area usage for designs expressed in DHDL. Estimates take into account available off-chip memory bandwidth and on-chip resources for datapath and routing, as well as effects from low-level optimizations like LUT packing and logic duplication.
- We study the space of designs described by tiling sizes, parallelization factors, and coarse-grained pipelining. This space is larger than previous work because we study more design dimensions than what is possible using state-of-the-art HLS tools. This in turn allows our system to find better design points in terms of performance and performance-per-area than previously possible.
- We evaluate the quality of our estimators and generated designs on a variety of compute-bound and memory-bound benchmarks from the machine learning and data analytics domain. We show that our run time estimates are within 6.1% and area estimates are within 4.8% of post place-and-route reports provided by FPGA vendor tools, making this a useful tool for design space exploration. We evaluate the performance of the generated designs compared to optimized multi-core CPU implementations running on a server-grade processor and achieve speedups of up to 16.7×.

The remainder of this paper is structured as follows. Section II outlines the requirements of good automated FPGA design tools and reviews related work in this domain. Section III describes the DHDL language and provides insights into how this representation enables larger design space exploration and accurate estimation. Section IV describes our modeling methodology for DHDL templates. Section V discusses the evaluation of our approach for absolute accuracy, design exploration efficiency, and performance of generated designs compared to a multi-core processor.

II. Background and Related Work

A primary requirement for good accelerator design tools is the ability to capture and represent design points along all important dimensions. Specifically, design tools must be able to capture application-level parameters (e.g., input sizes, bitwidth, data layout), architectural parameters (parallelism factors, buffer sizes, banking factors pipelining levels, off-chip memory streams) and microarchitectural parameters (e.g., on-chip memory word width). Having a representation rich in parallelism information allows for more accurate estimations, thorough design space exploration, and efficient code generation.

In addition to application characteristics, both heterogeneity within FPGAs and low-level optimizations done by

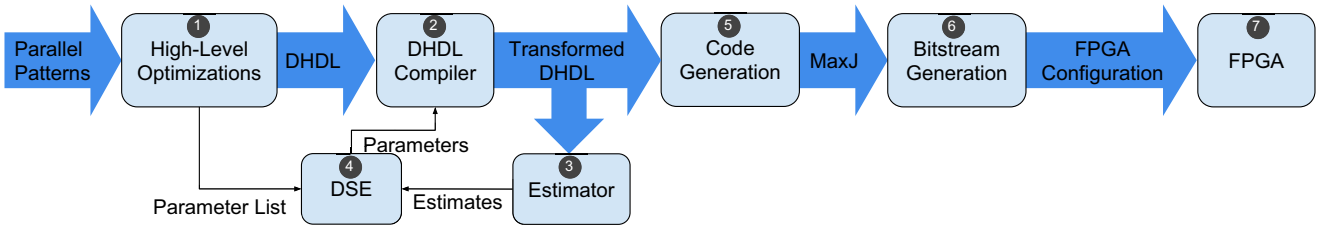

Figure 1. System Overview

logic synthesis tools have significant impact on required design resources. FPGA resource utilization does not just depend on the compute and memory operations in a given design; a non-trivial amount of resources are typically used to establish static routing connections to move data between two points, often rendering them unavailable for "real" operations. In addition, low-level logic synthesis tools often perform optimizations like LUT-packing or logic duplication for signal fanout reduction that alter resource usage. Off-chip memory communication requires FPGA resources to implement various queues and control logic. Such effects from low-level tools must be factored into the design tools to provide accurate estimates of design resource requirements.

A good FPGA design tool should have the following features:

- *Representation*: The tool must internally represent hardware using a general and parameterizable representation. This representation must preserve information regarding data locality, memory access pattern and parallelism in the input at all levels of nesting. Such a representation must be target-agnostic and should be targetable from high-level language constructs.
- *Estimation*: The tool must quickly analyze a design in the above representation and estimate metrics such as cycle counts and FPGA resource requirements for a target FPGA.
- *DSE*: The tool must be able to leverage the estimators to prune the large design search space, walk the space of designs, and find the Pareto-optimal surface.
- *Generation*: The tool must be able to automatically generate hardware which can then be synthesized and run on the target FPGA. Without this feature, hardware would typically be generated using separate toolchains for estimation and generation, which makes accurate estimation much harder.

Previous work on generating FPGA accelerators has focused on various aspects of the points mentioned above. Here we provide an overview of this work.

High-level synthesis (HLS) tools such as LegUp [13] and Vivado HLS [23] (previously AutoPilot) [12] synthesize hardware from C. These tools provide estimates of the cycle count, area and power consumption along with hardware generation. However, imperative design descriptions place greater burden on the compiler to discover parallelism, pipeline structure and memory access patterns. The absence of explicit parallelism often leads to conservative compiler analyses producing sub-optimal designs. While some tools allow users to provide compiler hints in the form of directives or pragmas in the source code, this approach fails to capture key points in the design space. For example,

```
L1: for (int i=0; i<R; i++) {
  #pragma HLS PIPELINE II=1
  L11: for (int j=0; j<C; j++) {
    sub[j] = y[i] ? x[i][j]-mu0[j] : x[i][j]-mu1[j];
  }
  L121: for (int j1=0; j1<C; j1++) {
    L122: for (int j2=0; j2<C; j2++) {
      sigma[j1][j2] += sub[j1]*sub[j2];
  }
}
```

Figure 2. GDA for high-level synthesis.

consider Figure 2 which represents the gaussian discriminant analysis (GDA) kernel. All loops in this kernel are parallel loops. One set of valid design points would be to implement $L1$ as a coarse-grained pipeline with $L11$ and $L121$ as its stages. Commercial HLS tools support limited coarse-grained pipelining, but with serveral restrictions. For example, the *DATAFLOW* directive in Vivado HLS enables users to describe coarse-grained pipelines. However, the directive does not support arbitrarily nested coarse-grained pipelines, multiple producers and consumers between stages, or coarse-grain pipelining within a finite loop scope [24], as required in the outer loop in Figure 2. In addition, compile times for HLS can be long for large designs due to the complications that arise during scheduling. Previous studies [15] point out other similar issues. Such limitations restrict the capability of HLS tools to explore more complex design spaces.

Pouchet et al. [25] explore combining HLS with polyhedral analysis to optimize input designs for locality and use estimates from HLS tools to drive design space exploration. While this captures a larger design space than previous work by including tile sizes, this approach is limited to the capabilities of the HLS tools and to benchmarks that have strictly affine data accesses. This paper improves upon previous work by modeling tiling parameters in addition

to other design points like coarse-grained pipelining of imperfectly nested loops which are not supported by HLS tools, as well as data-dependent accesses which are not supported by polyhedral analysis. Chen et al. [26] describe a simultaneous resource allocation and binding algorithm and perform design space exploration using a high-level power estimator. They characterize area usage of primitives and fit linear models to derive estimation functions. However, this study does not consider higher level design parameters or nested parallelism as part of the design space. We perform characterization of primitive operations as well as other coarse-grained templates, which enables us to estimate resource usage for much more complex accelerators. CMOST [18] is a C-to-FPGA framework that uses task-level modeling to exploit multi-level parallelism. While CMOST uses simple analytical models, this paper uses a mixture of analytical and machine learning models that enables much more fine-grained and accurate estimation of FPGA resource utilization.

Aladdin[15] is a pre-RTL estimation tool for ASIC accelerator design. Aladdin uses a dynamic data dependence graph (DDDG) as input and estimates the latency, area, and power for a variety of designs. However, using a DDDG limits the tool's ability to discover nested parallelism and infer coarse-grained pipeline structures that require double buffering, especially with complex memory accesses in patterns like *filters* or *groupBy*s. Also, Aladdin is focused on ASIC designs while our work focuses on FPGA accelerators which have a different set of challenges, as outlined above.

Other related work [27, 28, 29, 30, 31] explore various ideas from analytical to empirical models for estimating latency and area of designs in high-level languages. However, these approaches do not consider complex applications with nested parallelism. Also, previous work either ignores memory or has a relatively simple model for memory. This paper handles both on-chip and off-chip memory accesses with varying, data-dependent memory access patterns.

III. DHDL

In this section, we describe the Delite Hardware Definition Language, or *DHDL*. DHDL is an intermediate language for describing hardware datapaths. A DHDL program describes a dataflow graph consisting of various kinds of nodes connected to each other by data dependencies. Each node in a DHDL program corresponds to one of the supported architectural templates listed in Table I. DHDL is represented in-memory as a parameterized, hierarchical dataflow graph.

DHDL is a good hardware representation to aid in design space exploration for the following reasons:

- Templates in DHDL capture parallelism, locality, and access pattern information at multiple levels. This dramatically simplifies coarse-grained pipelining and enables us to explicitly capture and represent a large space of designs which other tools cannot capture, as shown in Figure 2.
- Every template is parameterized. A specific hardware design point is instantiated from a DHDL description by instantiating all the templates in the design with concrete parameter values passed to the program. DHDL heavily uses metaprogramming, so these values are passed in as arguments to the DHDL program. The generated design instance is represented internally as a graph that can be analyzed to provide estimates of metrics such as area and cycle count. The parameters used to create the design instance can be automatically generated by a design space exploration tool.

DHDL is implemented as an embedded domain-specific language in Scala, thereby leveraging Scala's language features and type system.

A. Generating DHDL from parallel patterns

A fundamental design goal of DHDL is that it should be automatically generated from high-level languages which express the computation using parallel patterns such as *map*, *reduce*, *zip*, *groupBy* and *filter*. Previous work has shown that parallel patterns can be used to improve both productivity and performance by using patterns as a basis for high-level languages [16, 19] and sophisticated compiler frameworks [20]. The templates in DHDL are inspired from these well-known parallel patterns. This makes it possible to define explicit rules to generate DHDL for each parallel pattern mentioned above. Previous work [22] has proposed compilation techniques to automatically generate hardware designs from parallel patterns using a similar template-based approach. We use these techniques to automatically generate DHDL from parallel patterns.

B. Language constructs

A hardware datapath is described in DHDL using various nodes connected to each other by their data dependencies. DHDL also supports variable bit-width fixed-point types, variable precision floating point types, and associated type checking. Every node that either produces or stores data has an associated type. Table I describes the hardware templates and associated parameters supported in DHDL. There are four types of nodes:

1) Primitive Nodes: Primitive nodes correspond to basic operations, such as arithmetic and logic tasks, and multiplexers. Some complex multi-cycle operations such as *abs*, *sqrt* and *log* are also supported as primitive nodes. Every primitive node represents a vector computation; a "vector width" parameter defines the number of parallel instances of each node. Scalar operations are thus special cases where the associated vector width is 1.

2) Memories: DHDL distinguishes between on-chip buffers and off-chip memory regions by representing them explicitly using separate nodes. This is used to capture

Template		Description	Design Parameters
Primitive Nodes	+, -, *, /, <. >, mux	Basic arithmetic, logic, and control operations	Vector width, Type
	Ld, St	Load and store from on-chip memory	Vector width, Bank stride
Memories	OffChipMem	N-dimensional off-chip memory array	Dimensions, Type
	BRAM	On-chip scratchpad memory	Dimensions, Word width, Double buffering, Vector width, Banks, Interleaving scheme, Type
	Priority Queue	Hardware sorting queue	Double buffering, Depth, Type
	Reg	Non-pipeline register	Double buffering, Vector width
Controllers	Counter	Counter chain used to produce loop iterators	Vector width
	Pipe	Hardware pipeline of primitive operations. Typically used to represent bodies of innermost loops.	Parallelization factor, Pattern
	Sequential	Non-pipelined, sequential execution of multiple stages.	Parallelization factor, Pattern
	Parallel	Fork-join style parallel container with synchronizing barrier.	
	MetaPipe	Coarse-grained pipeline with asynchronous handshaking signals across stages.	Parallelization factor, Pattern
Memory Command Generators	TileLd	Load a tile of data from an off-chip array	Tile dimensions, Word width, Parallelization factor
	TileSt	Store a tile of data to an off-chip array	Tile dimensions, Word width, Parallelization factor

Table I
DESCRIPTION OF TEMPLATES IN DHDL AND SUPPORTED PARAMETERS FOR EACH TEMPLATE

on-chip and off-chip accesses which have different access times resource requirements. *OffChipMem* represents an N-dimensional region of memory stored on off-chip DRAM. *BRAM*, *Priority Queue* and *Reg* correspond to different types of on-chip buffers specialized for different kinds of computation. *OffChipMem*s are accessed using nodes called *memory command generators*, while on-chip buffers are accessed using primitive *Ld* (load) and *St* (store) nodes. The banking factor for a *BRAM* node is automatically calculated using the vector widths and access patterns of all the *Ld* and *St* nodes accessing it such that the required memory bandwidth can be met.

3) Controllers: Several controller templates are supported in DHDL to capture imperfectly nested loops and parallelism at multiple nesting levels. Parallel patterns in input designs are represented using one of the *Pipe*, *MetaPipe*, or *Sequential* controllers with an associated *Counter* node. Each of these controllers is associated with a parallelization factor and the parallel pattern from which it was generated, which is used in replicating the nodes for parallelization. For example, nodes associated with the *map* pattern are replicated and connected in parallel, whereas nodes associated with the *reduce* pattern are replicated and connected as a balanced tree. *Pipe* is a dataflow pipeline which consists of purely primitive nodes. This typically represents innermost bodies of parallel loops that are traditionally converted to pipelines using software pipelining techniques. *MetaPipe* represents a coarse-grained pipeline where each of its stages are other controller nodes. *MetaPipe* orchestrates the execution of its stages in a pipelined fashion using asynchronous handshaking signals, thereby being able to tolerate variations in the execution times of each stage. Communication buffers used in between stages are converted to double buffers. *Sequential* represents unpipelined execution of a chain of controller nodes. *Parallel* is a container to execute multiple controller nodes in parallel with an implicit barrier at the end of execution. *Counter* is a simple chain of counters required to generate loop iterators. *Counter* has an associated vector width so that multiple successive iterators can be produced in parallel. This vector width is typically equal to the parallelization factor of the *Pipe*, *MetaPipe*, or *Sequential* it is associated with.

*4) Memory Command Generators: OffChipMem*s in DHDL are accessed at the granularity of *tiles*, where *tile* is an regular N-dimensional region of memory. Previous work [25, 22] has shown the importance of tiling transformations to maximize locality and generate efficient hardware. Accesses to *OffChipMem*s are explicitly captured in DHDL using special *TileLd* (tile load) and *TileSt* (tile store) controllers. Each *TileLd* and *TileSt* node instantiates data and command queues to interface with the memory controller, and contains control logic to generate memory commands.

C. Code Example: GDA in DHDL

Figure 4 shows GDA written in DHDL, complete with off-chip memory transfers. The hardware described is depicted pictorially in Figure 3. Note that the design captures nested parallelism with two levels of *MetaPipe*s with stages separated by double buffers. Each bubble denotes parameters that apply to the template it points to. Some of the parameters, like number of banks for *BRAM*, is omitted as they are automatically inferred based on parallelization factors. The design is parameterized using three kinds of parameters:

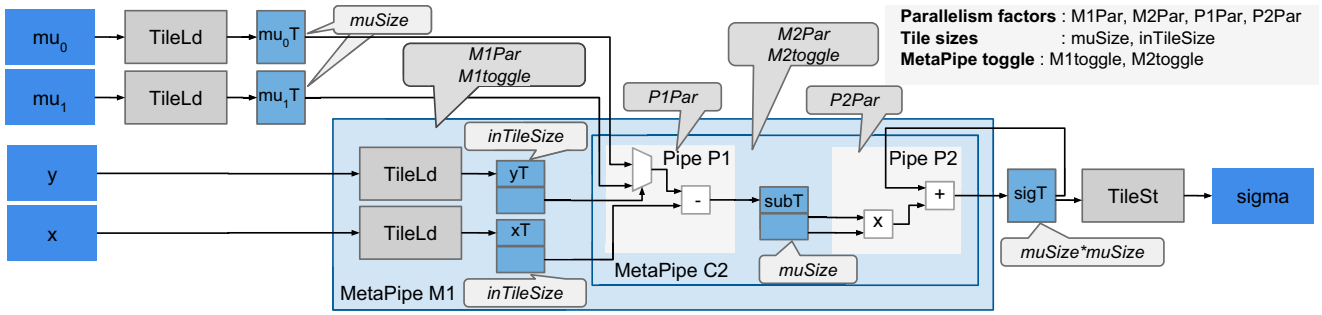

Figure 3. Parameterized GDA design described in Figure 4. Bubbles denote parameters that apply to the template it points to. Note that some parameters (e.g. *M1toggle*) apply to more than one template (e.g. *M1*, *xT*, and *yT*) but have been not been shown for clarity.

```
1  val x = OffChipMem[Float](R, C)
2  val y = OffChipMem[Bit](R)
3  val mu0 = OffChipMem[Float](C)
4  val mu1 = OffChipMem[Float](C)
5  val sigma = OffChipMem[Float](C, C)
6
7  Sequential {
8    val mu0T = BRAM[Float](muSize)
9    val mu1T = BRAM[Float](muSize)
10   Parallel {
11     mu0T := mu0(0::muSize) // Load mu0
12     mu1T := mu1(0::muSize) // Load mu1
13   }
14
15   val sigT = BRAM[Float](muSize, muSize)
16   MetaPipe(rows by inTileSize, sigT) { r =>
17     val yT = BRAM[Bit](inTileSize)
18     val xT = BRAM[Float](inTileSize, muSize)
19     Parallel {
20       // Load one tile of x and y
21       xT := x(r::r+inTileSize, 0::muSize)
22       yT := y(r::r+inTileSize)
23     }
24
25     val sigmaBlk = BRAM[Float](muSize, muSize)
26     MetaPipe(inTileSize by 1, sigmaBlk) { rr =>
27       val subT = BRAM[Float](muSize)
28       val sigmaTile = BRAM[Float](muSize, muSize)
29       Pipe(muSize by 1){ cc =>
30         val sub = yT(rr) ? mu1T(cc) :: mu0T(cc)
31         sigmaTile(cc) = xT(rr,cc) - sub
32       }
33       Pipe(muSize by 1, muSize by 1) { (ii,jj) =>
34         sigmaTile(ii,jj) = subT(ii) * subT(jj)
35       }
36       sigmaTile
37     }{_+_}
38     sigmaBlk
39   }{_+_}
40
41   sigma(0::muSize, 0::muSize) := sigT
42 }
```

Figure 4. GDA in DHDL

parallelism factors controlling number of parallel iterations, *tile sizes* corresponding to on-chip buffer sizes and *MetaPipe toggles* which controls whether an outer loop should be implemented as a *Sequential* or a *MetaPipe*. The *MetaPipe toggle* parameters also control whether the buffers internal to the *MetaPipe* should be double-buffered. Note that by supplying different parameters, different design points implementing GDA can be automatically generated from the same DHDL source code. In comparison to the design in Figure 3, the high-level synthesis specification in Figure 2 cannot capture the design points where either *M1toggle* or *M2toggle* are set to *true*. Also, it is challenging to generate multiple design points using the input in Figure 2 without extensively modifying the source code. We explore these design spaces with various parallelism factors, tile sizes, and toggles in detail in Section V. We generate parameters to the design in Figure 3 automatically using a design-space exploration tool, and each proposed design is analyzed to estimate FPGA resource utiliztion and cycle counts.

IV. Modeling and Estimation

In this section, we describe our modeling methodology. Our models account for the various design parameters for each DHDL template, as listed in Table I, as well as optimizations done by low-level logic synthesis tools in order to accurately estimate resource usage.

A. Modeling Considerations

The resource requirements of a given application implemented on an FPGA depend both on the target device and on the toolchain. Heterogeneity in the FPGA fabric, use of FPGA resources for routing, and other low-level optimizations performed by logic synthesis tools often have a significant impact on the total resource consumption of a design. Since these factors reflect the physical layout of computation on the device after placement and routing, they are not captured directly in the application's dataflow graph. We identify and account for the following factors:

LUT and register packing: Basic compute units in FPGAs are typically composed of a lookup table (LUT), and a small number of single bit multiplexers, registers, and full adders. Modern FPGA LUTs support up to 8-input binary functions but are often implemented using a pair of smaller LUTs [32, 33]. When these LUTs can be configured and used independently, vendor placement tools

attempt to "pack" multiple small functions into a single 8-input unit. LUT packing can have a significant impact on design resource requirements. In our experiments, we are able to pack about 80% of the functions in each design in pairs, decreasing the number of used LUTs by about 40%.

Routing Resources: Logic synthesis tools require a significant amount of resources to establish static routing connections between two design points (e.g., a multiplier and a block RAM) which fit in the path's clock period. While FPGAs have dedicated routing resources, logic synthesis tools may have the option to use LUTs for routing. These LUTs may then be unavailable to be used for "real" compute. In our designs, "route-through" LUTs typically account for about 10% of the total number of used LUTs.

Logic duplication: Logic synthesis tools often duplicate resources such as block RAMs and registers to avoid routing congestion and to decrease fan out. While duplicated registers typically encompass around 5% of the total number of registers required in our designs, we found that block RAM duplication can increase RAM utilization by 10 to 100%, depending on the complexity of the design.

Unavailable resources: FPGA resources are typically organized in a hierarchy, such as Altera's Logic Array Block structure (10 LUTs) and Xilinx's Slice structure (4 LUTs). Such organizations impose mapping constraints which can lead to resources that are rendered unusable. In our experiments, the number of unusable LUTs made up only about 4% of the design's total LUT usage.

B. Methodology

In order to model runtime and resource requirements of DHDL designs, we first need an estimate of the area requirements and propagation delay of every DHDL template. Area requirements include the number of digital signal processing units (DSPs), device block RAMs, LUTs, and registers that each template requires. To facilitate LUT packing estimation, we split template LUT resource requirements into the number of "packable" and "unpackable" LUTs required. We obtain characterization data by synthesizing multiple instances of each template instantiated for combinations of its parameters as given in Table I. Using this data, we create analytical models of each DHDL template's resource requirements and cycle counts for a predefined fabric clock. The area and cycle count of controller templates are modeled as functions of the latencies of the nodes contained within them. The total cycle count for a *MetaPipe*, for example, is modeled using the recursive function

$$(N-1)\max(cycles(n)|n \in nodes) + \sum_{n \in nodes} cycles(n)$$

where N is the number of iterations of the *MetaPipe* and *nodes* is the set of nodes contained in the *MetaPipe*.

Most templates require about six synthesized designs to characterize their resource and area usage as a function of their parameters. Note that these models include estimates of off-chip memory access latency as a function of the number and length of memory commands, as well as contention due to competing accessors. Since template models are application-independent, each needs only be characterized once for a given target device and logic synthesis toolchain. The synthesis times required to model templates can therefore be amortized over many applications.

Using these models, we run a pair of analysis passes over the application's DHDL intermediate representation to estimate design cycle counts and area requirements.

1) Cycle Count Estimation: In the first analysis pass, we estimate the total runtime of the design on the FPGA. Since the DHDL intermediate representation is hierarchical in nature, this pass is done recursively. The total runtime of *MetaPipe* and *Sequential* nodes is calculated first by determining the runtime of all controller nodes contained within them. The total propagation delay of a single iteration of a *Pipe* is the length of the body's critical path, calculated using a depth first search of the body's subgraph and the propagation delay of all primitive nodes within the graph. Input dataset sizes, given as user annotations in the high-level program, are used by the analysis pass along with tiling factors to determine the iteration counts for each controller template. Iteration counts are then used to calculate the total runtime of the respective controller nodes.

2) Area Estimation: Since the FPGA resource utilization of a design is sensitive to factors that are not directly captured in the design's dataflow graph, we adopt a hybrid approach in our area analysis.

We first estimate the area of the DHDL design by counting the resource requirements of each node using their pre-characterized area models. In *Pipe* bodies, we also estimate the resources required for delaying signals. This is done by recursively calculating the propagation delay of every path to each node using depth first search. Paths with slack relative to the critical path to that node require their width (in bits) multiplied by the slack delay resources. Delays over a synthesis tool-specific threshold are modeled as block RAMs. Otherwise, they are modeled as registers. Note that this estimation assumes ASAP scheduling.

We model LUT routing usage, register duplication, and unavailable LUTs using a set of small artificial neural networks implemented using the Encog machine learning library [34]. Each network has three fully connected layers with eleven input nodes, six hidden layer nodes, and a single output node. We chose to use three layer neural networks as they have been proven to be capable of fitting a wide number of function classes with arbitrary precision, including polynomial functions of any order [35]. One network is trained for each factor on a common set of 200 design samples with varying levels of resource usage to give a representative sampling of the space. Choosing the correct network parameters to obtain the lowest model error

is typically challenging, but in our experiments we found that above four nodes in the hidden layer, the exact number of hidden layer nodes made little difference. Duplicated block RAMs are estimated as a linear function of the number of routing LUTs, as we found that this gave the best estimate of design routing complexity in practice. This linear function was fit using the same data used to train the neural networks. Like the template models, these neural networks are application independent and only need to be trained once for a given target device and toolchain.

We use the raw resource counts as an input to each of our neural networks to obtain global estimates for routing LUTs, duplicated registers, and unavailable LUTs. We estimate the number of duplicated block RAMs using the routing LUTs. These estimates are then added to the raw resource counts to obtain a pre-packing resource estimate. For the purposes of LUT packing, we assume routing LUTs are always packable.

Lastly, we model LUT packing using the simple assumption that all packable LUTs will be packed. The target device in our experiments supports pairwise LUT packing, so we estimate the number of compute units used for logic as the number of unpackable LUTs plus the number of packable LUTs divided by two. We assume that each compute unit will use two registers on average. We model any registers unaccounted for by logic compute units as requiring compute units with two registers each. This gives us the final estimation for LUTs, DSPs, and BRAM.

C. Design space exploration

Our design space exploration tool uses the resource and cycle count estimates to explore the space of designs described by the parameters in Table I. As we are dealing with large design spaces on the order of millions of points even for small benchmarks, we prune invalid and suboptimal points in the search space using a few simple heuristics:

- Parallelization factors considered are integer divisors of the respective iteration counts. We use this pruning strategy because non-divisor factors create edge cases which require additional modulus operations. These operations can significantly increase the latency and area of address calculation, typically making them poor design parameter choices [36].
- Tile sizes considered are divisors of the dimensions of the annotated data size. Similar to parallelization factors, tile sizes with edge cases are usually suboptimal as they increase load and store area and latency with additional indexing logic.
- Automatic banking of on-chip memories eliminates the memory banks as an independent variable. This prunes a large set of suboptimal design points where on-chip memory bandwidth requirements do not match the amount of parallelization.
- The total size of each local memory is limited to a fixed maximum value.

Benchmark	Description	Dataset Size
dotproduct	Vector dot product	187,200,000
outerprod	Vector outer product	$38,400$ $38,400$
gemm	Tiled matrix multiplication	1536×1536
tpchq6	TPC-H Query 6	N=18,720,000
blackscholes	Black-Scholes-Merton model	N=9,995,328
gda	Gaussian discriminant analysis	R=360,000 D=96
kmeans	*k*-Means clustering	#points=960,000, k=8, dim=384

Table II
EVALUATION BENCHMARKS.

These heuristics defines a "legal" subspace of the total design space. In our experiments, we randomly generate estimates for up to $75,000$ legal points to give a representative view of the entire design space. We immediately discard illegal points.

V. EVALUATION

We evaluate the accuracy of the estimations described in Section IV. We use our models to study the space of designs on benchmarks from the data analytics, machine learning, and financial analytics domains. We then evaluate the speed of our design space exploration against a commercial high-level synthesis tool. Finally, we evaluate the performance of our Pareto-optimal points by comparing the execution times with an optimized multicore CPU implementation on a server-grade processor.

A. Experimental Setup

Table II lists the benchmarks we use in our evaluation along with the corresponding input dataset sizes used. *Dotproduct*, *outerprod*, and *gemm* are common linear algebra kernels. *Tpchq6* is a data analytics application that streams through a collection of records and performs a reduction on records filtered by a condition. *BlackScholes* is a financial analytics application that implements Black-Scholes option pricing. *Gda* and *kmeans* are commonly used machine learning kernels used for data classification and clustering, respectively. All benchmarks operate on single-precision floating point numbers, except in certain cases where the benchmark requires integer or boolean values as inputs. For the purposes of this paper, all benchmarks were written in DHDL by hand but are equivalent to what could be generated automatically from higher level DSLs.

We implement the DHDL compiler framework in Scala. The DHDL compiler generates hardware by emitting MaxJ, which is a low-level Java-based hardware generation language. Each generated design is synthesized and run on an Altera 28nm Stratix V FPGA on a Max4 MAIA board at a fabric clock frequency of 150MHz. The MAIA board

Benchmark	ALMs	DSPs	BRAM	Runtime
dotproduct	1.7%	0.0%	13.1%	2.8%
outerprod	4.4%	29.7%	12.8%	1.3%
gemm	12.7%	11.4%	17.4%	18.4%
tpchq6	2.3%	0.0%	5.4%	3.1%
blackscholes	5.3%	5.3%	7.0%	3.4%
gda	5.2%	6.2%	8.4%	6.7%
kmeans	2.0%	0.0%	21.9%	7.0%
Average	4.8%	7.5%	12.3%	6.1%

Table III
AVERAGE ABSOLUTE ERROR FOR RESOURCE USAGE AND RUNTIME.

interfaces with an Intel CPU via PCIe. The board has 48GB of dedicated off-chip DDR3 DRAM with a peak bandwidth of 76.8GB/s. In practice, our maximum memory bandwidth is 37.5 GB/s, as our on-chip memory clock is limited to 400MHz. We leverage Maxeler's runtime to manage communication and data movement between the host CPU and the MAIA board. Execution time is measured starting from when the FPGA design is started (after input has been copied to FPGA DRAM) and stopped after the design finishes execution (before output is copied to CPU DRAM). We report execution time as the average of 20 runs to eliminate noise from design initialization time and off-chip memory latencies. The FPGA resource utilization numbers reported are from the post place-and-route report generated by Altera's logic synthesis tools.

B. Evaluation of estimator

We first evaluate the absolute accuracy of our modeling approach. We select five Pareto points generated from our design space exploration for each of our benchmarks. We then generate and synthesize hardware for each design and run it on the FPGA. We compare our area estimates to post place-and-route reports generated by Altera's toolchain. We then run the design on the FPGA and compare the estimated runtime to observed runtime. Note that runtime includes off-chip memory accesses from the FPGA to its DRAM. Table III summarizes the errors averaged across all selected Pareto points for each benchmark.

Our area estimates have an average error of 4.8% for ALMs, 7.5% for DSPs, and 12.3% for BRAMs, while our runtime estimation error averages 6.1%. Our highest error occurs in predicting DSPs for *outerprod*, where we over-predict by 29.7% DSP usage on average. However, we found that errors above 10% for DSP usage only occur for designs which use less than 2% of the total DSPs available on the device. As our benchmarks are limited by other resources (typically ALMs or BRAM), the relative error for DSPs is more sensitive to low-level fluctuations and noise. We observe that our DSP estimates preserve absolute ordering of resource utilization. Hence, this error does not affect the quality of the designs found during design space exploration, and improves with increased resource utilization.

Of our estimated metrics, BRAM estimates have the highest average error over all benchmarks. These errors are primarily from block RAM duplication done by the placement and routing tool. In designing our models, we found that BRAM duplication is inherently noisy, as more complex machine learning models failed to achieve better estimates than a simple linear fit. Our linear model provides a rough estimate of design complexity and routing requirements, but it does not provide a complete picture for when and how often the synthesis tool will decide to duplicate BRAMs. However, like DSPs, we find that our BRAM estimates track actual usage and preserve ordering across designs, making it usable for design space exploration and relative design comparisons.

Gemm has the highest overall error of any benchmark. We found that this is due to low-level hardware optimizations like floating point multiply-add fusion, fusion of floating point reduction trees, and BRAM coalescing that Maxeler's compiler performs automatically and that we use heuristics to predict. Since we do not have explicit control over these optimizations, it is possible to mispredict when they will occur. The *gemm* benchmark is exceptionally sensitive to these errors. However, as with the other errors, we found that this error does not detract from the model's ability to guide design space exploration as long as the possibility of this error is accounted for.

C. Design space exploration

1) Pareto-optimality analysis: In this section we show the Pareto-optimal curves of each benchmark derived from estimators. Figure 5 shows the design space scatter plots for all benchmarks in Table II. A design point is considered invalid if its resource requirement for at least one type of resource exceeds the maximum available amount on the target device. Pareto-optimal designs along the dimensions of execution time and ALM utilization are highlighted for each benchmark through all three resource plots. We now analyze each benchmark in detail.

Dotproduct (Figure 5 A,B,C) is a memory-bound benchmark. Peak execution time is reached by balancing tile loads and computation. Inner and outer loop parallelization allows us to quickly reach close to the input bandwidth. Runtimes of designs with *MetaPipe*s then slowly decrease as parallelization increases once the dominant stage becomes the dot product reduction tree. In *dotproduct*, designs with *MetaPipe* consume less resources than those with *Sequential* for the same performance. *Sequential*s require larger tile sizes and more parallelism to match *MetaPipe* performance.

Outprod (Figure 5 D,E,F) represents both a BRAM and memory bound benchmark. For $2N$ inputs, the total BRAM requirement is $2N + N^2$ to store the input and output tiles,

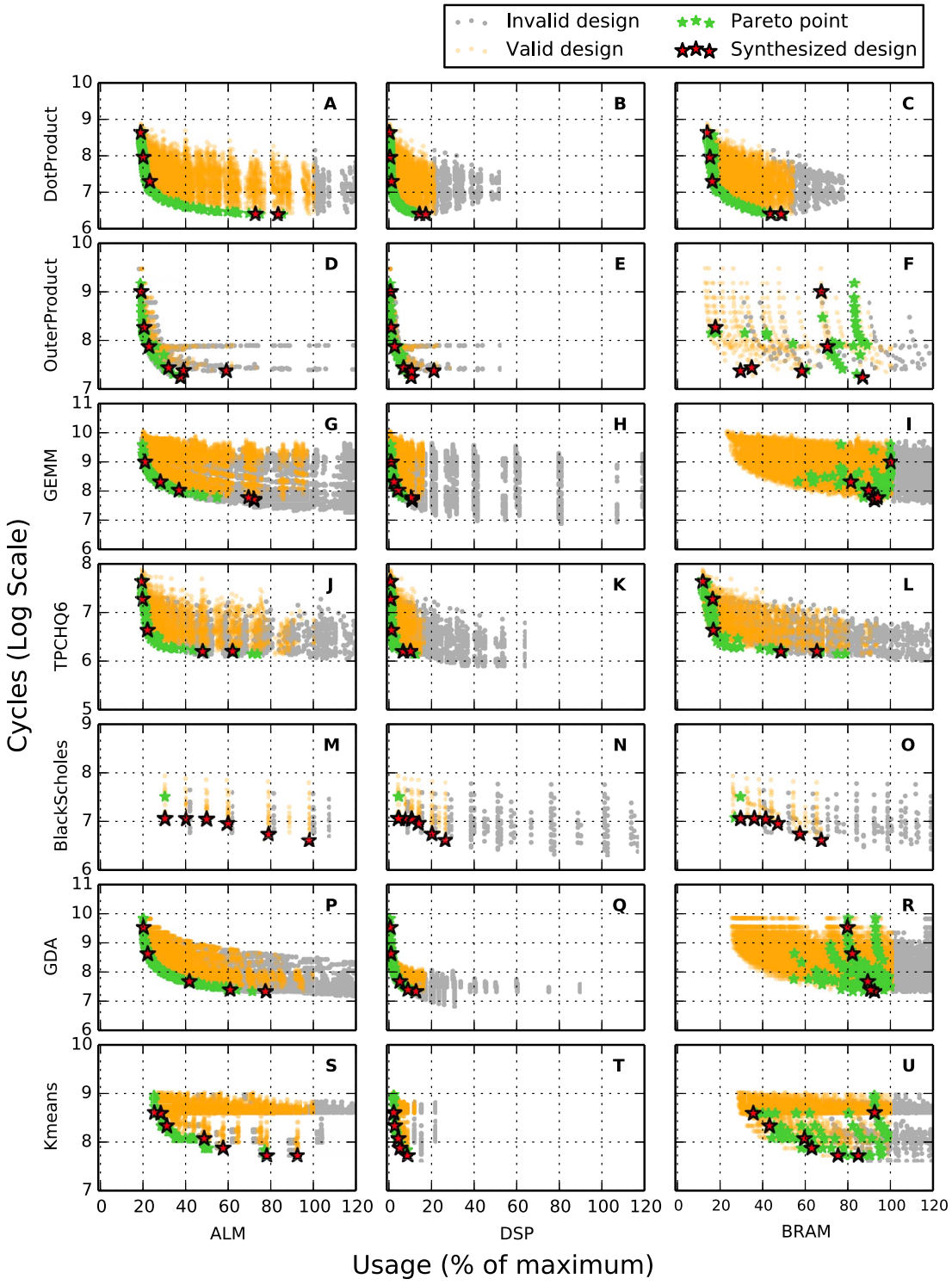

Figure 5. Results of design space exploration. Horizontal axis shows estimated ALM, DSP, and BRAM usages. Vertical axis shows runtime in cycles, given in log scale (base 10).

meaning the BRAM requirement increases quadratically with increases in input tile size. The highest performing designs for outer product do not use *MetaPipes* to overlap loading and storing of tiles. This is because the overhead due to main memory contention from overlapping tile loads and stores turns out to be higher than the cost of executing each stage sequentially.

Gemm (Figure 5 G,H,I) contains a lot of temporal and spatial locality. From Figure 5(I), Pareto-optimal designs for *gemm* occupy almost all BRAM resources on the board. Intuitively, this is because good designs for *gemm* maximize locality by retaining large, two dimensional chunks of data in on-chip memory.

Tpchq6 (Figure 5 J,K,L) exhibits behavior typical of memory intensive applications. Performance reaches a maximum threshold with increased tile size because of overlapping memory access and compute.

Blackscholes (Figure 5 M,N,O) streams through multiple large arrays and performs complex floating point computations on the input data. Points along the same vertical bar in Figure 5(M) share the same inner loop parallelization factor. Increasing parallelization improves performance by increasing utilization of the available off-chip memory bandwidth. Our model suggests that increasing the inner loop parallelization would continue to scale performance until a parallelization of 16, around which point *blackscholes* would be memory bound. Because there are not enough compute resources are available to implement a parallelization factor of 16, *blackscholes* is ALM bound.

Gda (Figure 5 P,Q,R) posseses higher degrees of spatial locality. Because of this, *gda* exhibits compute-bound behavior, where execution time decreases steadily with increased resource utilization, as seen in Figure 5(P). The critical resource is again BRAM. This is because BRAM usage increases with parallelization due to the creation of more banks with fewer words per bank, which can cause under-utilization of the capacity of individual BRAMs.

Kmeans (Figure 5 S,T,U) is bound by the number of ALMs. The critical path in this application is the distance computation done comparing an input point to each centroid. The number of floating point operations to be done to keep up with main memory bandwidth is therefore proportional to $K \times D$, where D is the number of dimensions in one point. The performance of *kmeans* is therefore limited by the number of ALMs on the FPGA, as not enough are available to perform all $K \times D$ operations in parallel. Like GDA, *kmeans* is also limited by BRAMs due to under-utilization of BRAM capacity with increased banking factors.

From our experiments, we observe that capturing parallelism at multiple levels using *MetaPipe*s enables us to generate efficient designs. In addition, effective management of on-chip BRAM resources is critical to good designs as BRAM resources are the limiting factor for performance scaling in most of our benchmarks.

Our approach	Vivado HLS restricted†	Vivado HLS full
0.017s / design	4.75s / design	111.06s / design

†Vivado HLS restricted design space ignores outer loop pipelining

Table IV
AVERAGE ESTIMATION TIME PER DESIGN POINT.

2) Speed of exploration: We compare the speed of our estimation and design space exploration with Vivado HLS [23], a commercial high-level synthesis tool from Xilinx. Our evaluation uses the GDA example in Figure 2 as input to the high-level synthesis tool, and the GDA design in Figure 3 as input to our design space exploration tool. Design parameters for the high-level synthesis tool are the unrolling factors. We also include a pipeline directive toggle for each loop in the design. For DHDL, we vary all design parameters specified in Figure 4. Speed is measured by comparing the average estimation speed per point for 250 design points for each tool. In our experiments, our analysis takes 5 to 29 milliseconds per design depending on the size of the application's intermediate representation.Analysis of GDA also takes 17 milliseconds per design.

Table IV shows a comparison between estimation speeds from our toolchain and Vivado HLS. The "restricted" column refers to the average time spent per design over points whose outer loop ($L1$, in Figure 2) is not pipelined with a pipeline directive. The "full" version refers to all design points where 30 of the 250 points have a pipeline directive to enable outer loop pipelining. We observe the following:

- Our estimation tool is 279× faster than the "restricted" space exploration, and 6533× faster than the "full" space exploration.
- Compared to Vivado HLS, our estimation time is not sensitive to design parameter inputs. Estimation time for Vivado HLS increases dramatically when the outer loop is pipelined in GDA because the tool completely unrolls all inner loops before pipelining the outer loop. This creates a large graph that complicates scheduling. Our approach does not suffer from this limitation because we explicitly capture pipelines in parameterized templates such as *Pipe* and *MetaPipe*, thereby capturing outer loop pipelining more naturally.

D. Comparison with CPU

To evaluate the quality of the generated Pareto-optimal designs, we compare the best FPGA execution times with optimized CPU implementations of all benchmarks. CPU comparison numbers were obtained by running C++ versions of benchmarks in Table II on a 6 core Intel(R) 32nm Xeon(R) CPU E5-2630 processor clocked at 2.30GHz, with a 15MB LLC and a maximum main memory bandwidth of 42.6 GB/s. Each CPU benchmark is run with 6 threads. For *gemm*, we compare to multi-threaded OpenBLAS [37].

The rest of the CPU implementations were generated from OptiML [16], a machine learning DSL which generates high performance, multi-threaded C++ comparable to, or better than, manually optimized code. CPU execution times are obtained by measuring the core computation kernel averaged over 10 runs. Figure 6 shows the the speedups of all our benchmarks normalized to the execution time on the CPU.

Both *dotproduct* and *outerprod* are streaming, memory-intensive benchmarks. For *dotproduct*, we see a speedup of 1.07×, roughly the same performance as the CPU. In *outerprod*, we see a speedup of 2.4×. We associate this speedup with overhead of multithreaded setup and synchronization. However, the CPU *outerprod* implementation can likely be improved further to match the FPGA's performance. Ultimately, we would not expect a significant difference in peformance on either of the benchmarks as the memory bandwidth of the two architectures is roughly the same. In the case of *outerprod*, both architectures should be equally capable of exploting spatial locality as the vector sizes are far smaller than local memory sizes.

We observe a significant slowdown by about 10× for *gemm*. By taking advantage of architecture-specific tiling techniques at multiple memories of the memory hierarchy and by vectorizing floating point operations, the OpenBLAS implementation can achieve a total of about 89 GFLOPs. Our FPGA does not have enough resources to achieve that performance on single precision floating point values. However, larger FPGAs with more compute capacity or, more recently, direct hardware support for floating point operations have been shown to be capable of much higher floating point performance than this.

The *tpchq6* benchmark achieves a speedup of 11.8× in spite of having an access pattern that streams through multiple large arrays. This is because *tpchq6* consists of data-dependent branches which cause frequent stalls in the frontend of the processor's pipeline. On the FPGA, such branches are implemented using simple multiplexors which do not create stalls or bubbles in the dataflow pipeline. Given the appropriate tile sizes, this shows that memory-intensive benchmarks like *tpchq6* that have branches can be accelerated on an FPGA.

Blackscholes achieves a speedup of 16.7×. The core compute kernel of *blackscholes* is amenable to deep pipelining. While the *blackscholes* benchmark is compute bound on the CPU [38], FPGAs can exploit higher levels of instruction-level parallelism than CPUs via deep pipelines. Our *blackscholes* design benefits from this pipeline parallelism.

The *gda* and *kmeans* achieve speedups of 4.5× and 1.15×, respectively. Both benchmarks have nested levels of parallelism which is captured using *MetaPipe*s. By exploting pipeline parallelism and taking advantage of locality within these two applications, our generated designs are able to achieve a modest speedup over the multi-core CPU.

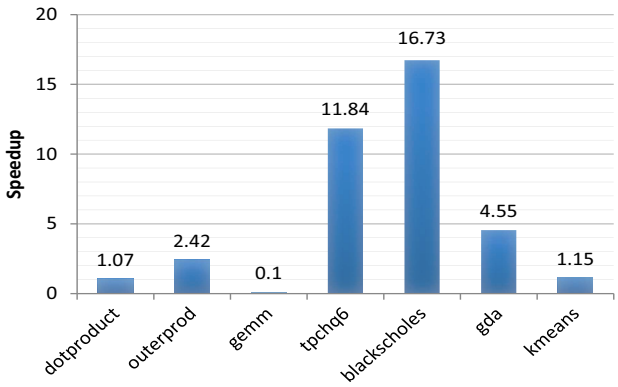

Figure 6. Normalized speedups of most performant FPGA design points over multi-core CPU implementations.

VI. Conclusion

In this paper, we describe a practical framework that can generate efficient FPGA designs automatically from a high-level description based on parallel patterns. We introduce DHDL, a new parameterizable hardware definition language that describes designs using templates such as *MetaPipe* with which we capture a larger design space than previous work. We describe our hybrid area estimation technique and evaluate our approach extensively on various benchmarks from the data analytics, financial analytics and machine learning domains. We show an average area estimation error of 4.8% and average runtime estimation error of 6.1% over all the benchmarks. We perform a detailed study for each benchmark on the space of designs described by tile sizes, parallelism factors, and coarse-grained pipelining and measure their effects on the utilization of different types of FPGA resources. We show that our exploration tool is 279 to 6533 times faster than a commercial high-level synthesis tool. Finally, we show that the Pareto-optimal designs we discover can achieve a speedup of up to 16.7× over optimized multi-core CPU implementations running on a commodity server processor.

Acknowledgments

The authors thank Maxeler Technologies for their assistance with this paper, and the reviewers for their suggestions. This work is supported by DARPA Contract-Air Force FA8750-12-2-0335; Army Contract AHPCRC W911NF-07-2-0027-1; NSF Grants IIS-1247701, CCF-1111943, CCF-1337375, and SHF-1408911; Stanford PPL affiliates program, Pervasive Parallelism Lab: Oracle, AMD, Huawei, Intel, NVIDIA, SAP Labs. Authors acknowledge additional support from Oracle. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

[1] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency & flexibility in specialized computing," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA, 2013, pp. 24–35.

[2] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "Pudiannao: A polyvalent machine learning accelerator," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2015, pp. 369–381.

[3] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The architecture and design of a database processing unit," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2014, pp. 255–268.

[4] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA, 2014, pp. 151–160.

[5] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA, 2010, pp. 37–47.

[6] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," in *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, ser. FPGA, 2006, pp. 21–30.

[7] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA, 2014, pp. 13–24.

[8] J. Ouyang, S. Lin, W. Qi, Y. Wang, B. Yu, and S. Jiang, "Sda: Software-defined accelerator for largescale dnn systems," ser. Hot Chips 26, 2014.

[9] P. K. Gupta, "Xeon+fpga platform for the data center," http://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf, 2015.

[10] "Falcon computing," http://falcon-computing.com/, 2015.

[11] D. Bacon, R. Rabbah, and S. Shukla, "Fpga programming for the masses," *Queue*, vol. 11, no. 2, Feb. 2013.

[12] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 473–491, 2011.

[13] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *TECS*, vol. 13, no. 2, p. 24, 2013.

[14] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: A java-compatible and synthesizable language for heterogeneous architectures," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA, 2010, pp. 89–108.

[15] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 2014, pp. 97–108.

[16] A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Wu, A. R. Atreya, K. Olukotun, T. Rompf, and M. Odersky, "Optiml: an implicitly parallel domain specific language for machine learning," in *ICML*, 2011.

[17] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun, "Composition and reuse with compiled domain-specific languages," in *European Conference on Object Oriented Programming*, 2013.

[18] P. Zhang, M. Huang, B. Xiao, H. Huang, and J. Cong. "CMOST: A System-level FPGA Compilation Framework," *DAC*, 2015.

[19] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, 2013, pp. 519–530.

[20] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," in *TECS'14: ACM Transactions on Embedded Computing Systems*, July 2014.

[21] Maxeler Technologies, "MaxCompiler white paper," 2011.

[22] R. Prabhakar, D. Koeplinger, K. J. Brown, H. Lee, C. De Sa, C. Kozyrakis, and K. Olukotun, "Generating configurable hardware from parallel patterns," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2016, pp. 651–665.

[23] "Vivado high-level synthesis," http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[24] "Vivado design suite 2015.1 user guide: High-level synthesis."

[25] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA, 2013, pp. 29–38.

[26] D. Chen, J. Cong, Y. Fan, and Z. Zhang, "High-level power estimation and low-power design space exploration for fpgas," in *Design Automation Conference, 2007. ASP-DAC '07. Asia and South Pacific*, Jan 2007, pp. 529–534.

[27] L. Deng, K. Sobti, Y. Zhang, and C. Chakrabarti, "Accurate area, time and power models for fpga-based implementations," *J. Signal Process. Syst.*, vol. 63, no. 1, pp. 39–50, Apr. 2011.

[28] S. Bilavarn, G. Gogniat, J.-L. Philippe, and L. Bossuet, "Design space pruning through early estimations of area/delay tradeoffs for fpga implementations," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, no. 10, pp. 1950–1968, 2006.

[29] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee, "Accurate area and delay estimators for fpgas," in *Design, Automation and Test in Europe Conference and Exhibition, Proceedings*, 2002, pp. 862–869.

[30] R. Enzler, T. Jeger, D. Cottet, and G. Tröster, "High-level area and performance estimation of hardware building blocks on fpgas," in *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*. Springer, 2000, pp. 525–534.

[31] P. Bjuréus, M. Millberg, and A. Jantsch, "Fpga resource and timing estimation from matlab execution traces," in *Proceedings of the tenth international symposium on Hardware/software codesign*, 2002.

[32] "Stratix device handbook," https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/stratix-v/stx5_core.pdf.

[33] "Xilinx 7 series fpgas configurable logic block user guide," http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf, 2014.

[34] J. Heaton, "Encog: Library of interchangeable machine learning models for java and c#," *Journal of Machine Learning Research*, vol. 16, pp. 1243–1247, 2015. [Online]. Available: http://jmlr.org/papers/v16/heaton15a.html

[35] F. Scarselli and A. C. Tsoi, "Universal approximation using feedforward neural networks: A survey of some existing methods, and some new results," *Neural Networks*, vol. 11, no. 1, pp. 15 – 37, 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S089360809700097X

[36] J. Cong, B. Liu, R. Prabhakar, and P. Zhang, "A study on the impact of compiler optimizations on high-level synthesis," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, H. Kasahara and K. Kimura, Eds. Springer Berlin Heidelberg, 2013, vol. 7760, pp. 143–157. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37658-0_10

[37] "Openblas," http://www.openblas.net/, 2016.

[38] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.