APEX: A Framework for Automated Processing Element Design Space Exploration using Frequent Subgraph Analysis

Jackson Melchert
Stanford University
Stanford, USA

Kathleen Feng
Stanford University
Stanford, USA

Caleb Donovick
Stanford University
Stanford, USA

Ross Daly
Stanford University
Stanford, USA

Ritvik Sharma
Stanford University
Stanford, USA

Clark Barrett
Stanford University
Stanford, USA

Mark A. Horowitz
Stanford University
Stanford, USA

Pat Hanrahan
Stanford University
Stanford, USA

Priyanka Raina
Stanford University
Stanford, USA

ABSTRACT

The architecture of a coarse-grained reconfigurable array (CGRA) processing element (PE) has a significant effect on the performance and energy-efficiency of an application running on the CGRA. This paper presents APEX, an automated approach for generating specialized PE architectures for an application or an application domain. APEX first analyzes application domain benchmarks using frequent subgraph mining to extract commonly occurring computational subgraphs. APEX then generates specialized PEs by merging subgraphs using a datapath graph merging algorithm. The merged datapath graphs are translated into a PE specification from which we automatically generate the PE hardware description in Verilog along with a compiler that maps applications to the PE. The PE hardware and compiler are inserted into a flexible CGRA generation and compilation toolchain that allows for agile evaluation of CGRAs. We evaluate APEX for two domains, machine learning and image processing. For image processing applications, our automatically generated CGRAs with specialized PEs achieve from 5% to 30% less area and from 22% to 46% less energy compared to a general-purpose CGRA. For machine learning applications, our automatically generated CGRAs consume 16% to 59% less energy and 22% to 39% less area than a general-purpose CGRA. This work paves the way for creation of application domain-driven design-space exploration frameworks that automatically generate efficient programmable accelerators, with a much lower design effort for both hardware and compiler generation.

CCS CONCEPTS

• **Hardware → Hardware accelerators**; **Reconfigurable logic applications**; **Programmable logic elements**;

KEYWORDS

CGRA, reconfigurable accelerators, processing elements, graph analysis, subgraph, design space exploration, domain-specific accelerators, hardware-software co-design

ACM Reference Format:
Jackson Melchert, Kathleen Feng, Caleb Donovick, Ross Daly, Ritvik Sharma, Clark Barrett, Mark A. Horowitz, Pat Hanrahan, and Priyanka Raina. 2023. APEX: A Framework for Automated Processing Element Design Space Exploration using Frequent Subgraph Analysis. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3582016.3582070

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9918-0/23/03…$15.00
https://doi.org/10.1145/3582016.3582070

1 INTRODUCTION

Coarse-grained reconfigurable arrays (CGRAs) have been widely studied in recent years and serve as a midpoint between the flexibility of a field-programmable gate array (FPGA) and the performance and energy-efficiency of an application-specific integrated circuit (ASIC) [26]. A CGRA can achieve better performance and energy-efficiency than an FPGA because its processing elements (PEs) have specialized arithmetic units that operate at a word-level, rather than a bit-level granularity. They are more flexible than an ASIC, due to their reconfigurable interconnect and PEs. However, CGRAs do not constitute just one point in the spectrum between FPGAs and ASICs; rather, they occupy a large design space from specialized, efficient CGRAs to general-purpose CGRAs that can accelerate many applications. CGRAs are useful when designed with an application domain in mind, allowing for appropriately specialized PE and memory architectures while leaving some flexibility to accommodate the evolution of applications in the domain.

In this paper, we present a methodology for application domain-driven design space exploration (DSE) of CGRA PEs. Our methodology analyzes applications to find common computational blocks that can be easily accelerated with specialized PEs. Using this methodology, we explore the space between general CGRAs that can execute many applications and specialized CGRAs that execute a more limited set but achieve high performance and energy-efficiency.

Our framework, called APEX (Automated PE Exploration), encompasses application analysis, PE specification, CGRA hardware generation, and compiler creation in an easy-to-use toolchain that requires little manual effort. Given an application or a set of applications, the framework can produce many PE architectures that explore the design space, along with the rest of the CGRA and the compiler to evaluate the PE designs. Our contributions are:

(1) Develop a framework to analyze applications using subgraph mining and generate candidate PEs specialized for those applications using subgraph merging.
(2) Automatically generate CGRAs with specialized PEs along with a compiler to run applications on those CGRAs utilizing an agile hardware flow.
(3) Provide analysis and insight into how generalization and specialization of PEs impacts energy, performance, and area of a variety of applications running on a CGRA.

To our knowledge, APEX is the first comprehensive framework for design space exploration of CGRA PEs that encompasses application analysis to automatically generate PE designs, a hardware generation toolchain for the entire CGRA with a flexible interconnect, customized PEs, and memory tiles, and finally a full compiler toolchain that automatically schedules, maps, place-and-routes, and pipelines applications onto the resulting customized accelerator.

The rest of this paper is organized as follows: Section 2 introduces the design space axes of CGRA PEs. Section 3 describes our application analysis and PE specialization methodology. Section 4 presents our DSE framework. Finally, Section 5 demonstrates the improvements obtained by specializing PEs using our framework for image processing and machine learning (ML) domains.

2 PROCESSING ELEMENT DESIGN SPACE

CGRAs are typically composed of several processing element (PE) and memory tiles, arranged in a grid-like fashion (Fig. 1). The tiles communicate via an interconnect, which comprises several horizontal and vertical routing tracks, connection boxes (CBs) that bring inputs into the tiles from the routing tracks, and switch boxes (SBs) that take outputs from the tiles and route them onto the routing tracks in different directions.

A survey of existing CGRA PE architectures indicates the presence of three design spaces axes [24] — the number and type of operations within a PE, the intraconnect of a PE, and the number of inputs to and outputs from a PE. A designer's objective when creating a CGRA is to maximize area-efficiency, energy-efficiency, and performance while maintaining enough flexibility to be able to program their accelerator to run new or evolving applications. Each of the three design space axes has a significant impact on these objectives.

2.1 Number and Type of Operations

CGRA PEs contain arithmetic units that execute many operations, enabling a variety of applications to run on the CGRA. At one end of this design space axis are the most simple PEs with just an adder or multiplier. Typical CGRAs contain more general PEs that contain an ALU and a multiplier. One example is the CGRA PE [3] shown in Fig. 1. This PE has an ALU which can execute the operations shown at the bottom of the figure, and a look up table for bit operations.

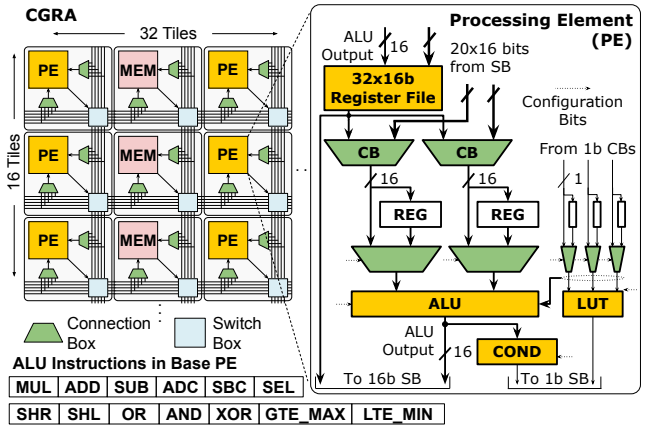

Figure 1: Baseline PE and CGRA architecture. The CGRA is composed of PE and MEM tiles connected with a configurable interconnect. A detailed view of the PE is shown on the right, which includes an ALU and a register file.

It also contains two 16-bit and three 1-bit registers for holding constant values. This type of design lends itself to high utilization and similar efficiency no matter the application running on the CGRA. However, these types of PEs might have significantly lower efficiency and performance than more specialized PEs.

On the other end of this design space axis are more specialized PEs that have multiple ALUs or multipliers. For example, CGRAs targeting image processing or ML are likely to have the ability to do a vector multiply-add operation [22]. This leads to high efficiency and performance when running image processing or ML applications but there might be under-utilization when running applications with fewer multiply-add operations. Some complex PEs [19] contain floating point logic that, while expensive, can enable running a wider range of applications.

2.2 Intraconnect

The number of configurable paths through a PE affects its area, power, and generality. Since CGRA PEs can execute more than one operation, at least one multiplexer is needed to route the result of the desired operation to the PE output. On one end of this axis are PEs that have very few configurable paths. These PEs are area and energy efficient, but are not very flexible. On the other end are PEs that have many multiplexers that enable many different configurable paths through the PE. A common example of this is to have multiplexers at the inputs of each arithmetic unit. Each input to the PE can be routed to either input of the ALU, so non-commutative operations like shifts can be achieved regardless of the order of the operands into the PE. These PEs are very flexible but have high power, performance, and area costs.

2.3 Number of Inputs and Outputs

The number of inputs and outputs (I/O) to each PE directly affects the size and number of connection boxes (CBs) and switch boxes (SBs) in the CGRA. As these components of the interconnect have high area and power costs, minimizing the I/O to the PE is critical

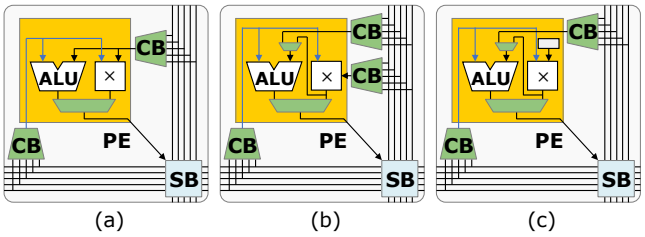

Figure 2: PE I/O: (a) PE with 2 inputs (2 CBs) and 1 output, (b) PE with 3 inputs (3 CBs), (c) Restricting the PE from (b) to receive one input from a constant register to reduce I/O.

for achieving an efficient CGRA. On one end of this design space axis are PEs that have two inputs and one output (Fig. 2a). This enables most arithmetic operations and results in small per-PE interconnect overhead. On the other end are designs that have many inputs to enable more complex operations, for example, a three input operation like a fused multiply add (Fig. 2b).

Another aspect of this design space axis is reducing the number of inputs to the PE using constant registers. In many image processing applications, convolutions are done using a kernel with fixed weights. A PE that has a fused multiply-add to implement this convolution would usually need three inputs, however, one of those inputs, the kernel weight, could be replaced by a constant register (Fig. 2c). This register can then be given a value during the configuration of the CGRA, reducing the overhead of the interconnect.

In summary, each PE design space axis has many potential values that have significant impacts on the energy and area efficiency, performance, and flexibility of the resulting CGRA. A naïve exploration of this space where a designer enumerates many different PEs with every value on each of the axes would lead to many candidate PEs. An intelligent method for exploring this design space and selecting interesting candidate PEs is needed for efficient exploration.

3 APPLICATION DOMAIN DRIVEN PE EXPLORATION

In this section, we introduce our methodology for efficiently identifying interesting candidate PEs, without doing a naïve enumeration of many points in the PE design space. Our methodology relies on generating efficient PE architectures from the application graphs themselves, using frequent subgraph mining and merging.

3.1 Subgraph Mining

To intelligently choose points in the large PE design space, we analyze our applications to determine the most frequent computational patterns. To do this, we leverage a technique called subgraph mining, which takes in an application dataflow graph and produces its frequent subgraphs.

We start with an application written in Halide [20], a domain-specific language (DSL) for image processing and machine learning. We use the Halide compiler from [3] to lower the application to a CoreIR [12] dataflow graph containing compute and memory nodes. Frequent subgraphs of the application dataflow graph represent common (potentially complex) operations in the application.

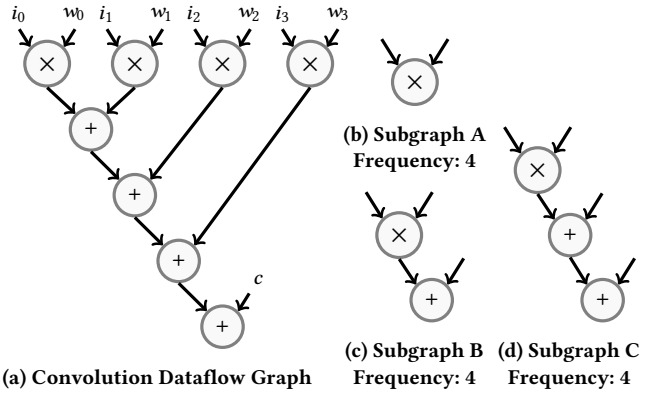

Figure 3: Frequent subgraph mining on a convolution CoreIR graph. 3b, 3c, and 3d are the three most frequent subgraphs, with four occurrences each.

Those frequent subgraphs are the starting points for constructing promising PEs.

Finding, or mining, frequent subgraphs relies on the computation of subgraph isomorphisms. As frequent subgraph mining is very useful in a wide variety of fields, this is a well-researched problem. We use GRAMI [13], a subgraph mining tool for single large graphs.

GRAMI takes the application graph as an input and the minimum number of times a subgraph can appear to be considered frequent. Fig. 3 gives an example of frequent subgraph mining for a simple application. Fig. 3a shows the dataflow graph of a convolution $(((((i_0 \times w_0) + (i_1 \times w_1)) + (i_2 \times w_2)) + (i_3 \times w_3)) + c)$. Some frequent subgraphs of this application are in Fig. 3b, 3c, and 3d.

A subgraph can be directly translated into a PE architecture. Each operation in the subgraph has a hardware interpretation, so constructing PEs from many different subgraphs can be easily automated. However, not all frequent subgraphs are as interesting as they initially seem. Fig. 3d is an example of such a case. While the frequency is four, the occurrences overlap, and therefore, only two of the occurrences can be effectively accelerated. We use maximal independent set analysis to mitigate this issue.

3.2 Maximal Independent Set Analysis

When a subgraph has overlapping occurrences in the application graph, only the non-overlapping occurrences can be accelerated with fully-utilized PEs that implement that subgraph, leading to several partially utilized PEs when mapping the application. Fig. 3d shows one frequent subgraph of a convolution application. This subgraph has many occurrences in the application, although several of these occurrences overlap.

To find when this problem occurs, we employ maximal independent set (MIS) [9] analysis, using the following steps:

(1) Represent each occurrence of the subgraph in the application as a node in a new graph.
(2) Represent overlapping subgraphs as edges between nodes. Overlapping subgraphs are those whose occurrences share any node.
(3) Calculate the maximal independent set (MIS) of this new graph. (An independent set of a graph is a set of vertices in

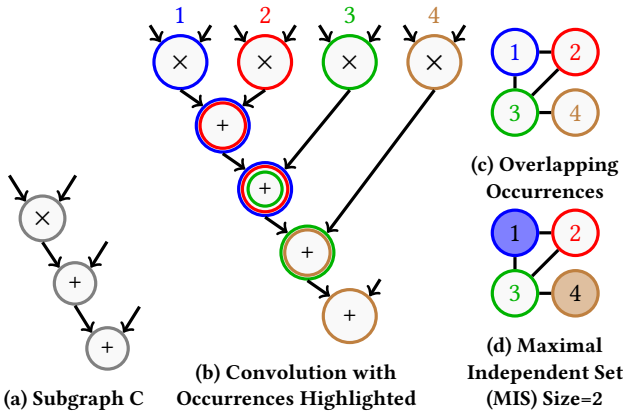

Figure 4: Maximal independent set analysis. Frequent subgraph C from Fig. 3 is shown again in 4a. Each of the four occurrences of subgraph C is given a different color in 4b. Each occurrence is represented by a node in 4c, with overlapping occurrences represented by edges. In 4d the maximal independent set is shown as the filled nodes and the maximal independent set (MIS) size is two.

that graph which do not share a neighbor. A maximal independent set is an independent set which cannot be grown by adding more vertices to it.) The size of this set is the number of times the subgraph exists in the application without overlaps.

Fig. 4 illustrates this on a simple application graph using the subgraph from Fig. 3d. The first occurrence of the subgraph is outlined in blue in Fig. 4b, the second in red, the third in green, and the fourth in brown. Each of these occurrences has a corresponding node in Fig. 4d. The maximal independent set is the set of nodes that are filled with their respective color (brown and blue). In this example, MIS size is two, meaning this subgraph occurs twice in the application graph not including overlapping occurrences. The MIS size of a subgraph indicates how many fully utilized PEs that implement this subgraph can be used to accelerate the application. Using the size of this set, we have a better idea of which application subgraphs might be interesting starting points for PE architectures.

3.3 Subgraph Merging

Implementing a frequent subgraph as an operation in a PE allows for the acceleration of a common computation in an application. Merging many subgraphs from one or more applications allows for the acceleration of multiple distinct parts of one application or even multiple different applications using one PE architecture.

However, merging subgraphs is not a trivial problem. We use a set of algorithms designed for high level synthesis for automated datapath graph merging [18]. The goal of these algorithms is to create a single structure that implements all of the distinct operations in the subgraphs with minimal area overhead. It produces one datapath that can be configured to each of the operations represented by the subgraphs.

As an example, Fig. 5a and Fig. 5b show two subgraphs that we want to merge together. The first step in the subgraph merging

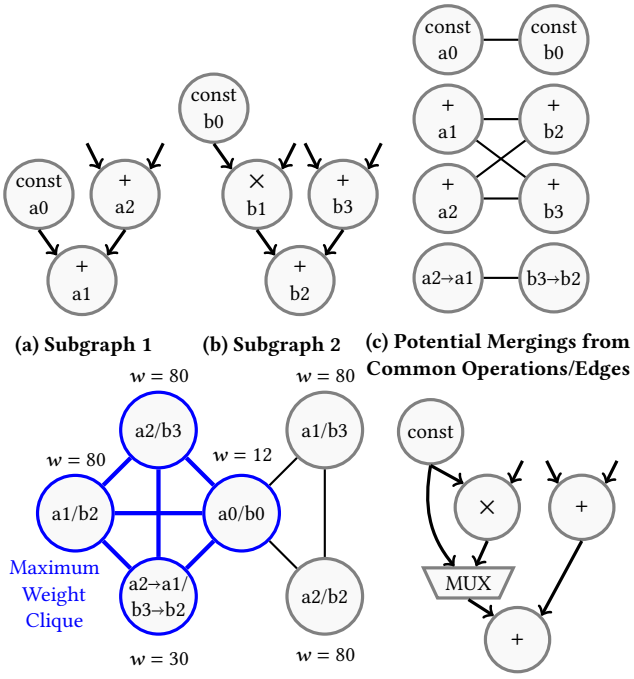

Figure 5: Example of merging subgraph 1 (a) and subgraph 2 (b). (c) represents the enumeration of every potential merging between the nodes of subgraph 1 and subgraph 2. (d) is the representation of the compatible potential mergings with the area benefit of each merging annotated as w. The clique with the highest total w is highlighted in blue. (e) is the reconstructed merged graph with the highest area benefit.

process is to create a set of potential merging opportunities between nodes of the same operation in each subgraph. Fig. 5c shows a bipartite graph with operations/edges of subgraph A shown as nodes on the left and those of subgraph B shown on the right. An edge from one node on the left to one on the right represents a potential merging opportunity. Two nodes can be merged if they are the same operation or can both be implemented on the same hardware block. Two edges can be merged if each of their endpoint nodes can be merged and the ports on the destination node match (this last condition is necessary for non-commutative operations, for example, a left shift: $out = in_0 << in_1$). Nodes and edges which do not have any merging opportunities have been omitted from Fig. 5c for simplicity.

In this example, nodes $a0$ and $b0$ are both constants, so there is a corresponding edge in the bipartite graph. Nodes $a1$, $a2$, $b2$, and $b3$ are all add operations, so there are corresponding edges between these nodes. Finally, the edge $a2 \rightarrow a1$ and the edge $b3 \rightarrow b2$ start at an add operation and end at an add operation, and the port on both $a1$ and $b2$ matches, so those edges can be potentially merged as well.

Next, these potential merging opportunities are transformed into a compatibility graph; each potential merging is represented as a

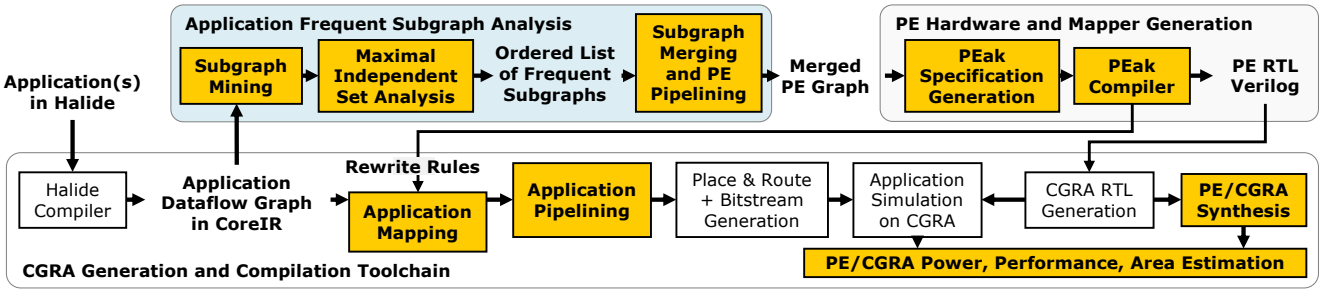

Figure 6: APEX design space exploration framework. Steps created or modified in this work are in yellow.

node, and each compatible merging is represented as an edge. Merging opportunities are compatible if they can both be implemented at the same time. Two mergings are incompatible if they merge one node in subgraph 1, to more than one nodes in subgraph 2, or vice versa. For example, merging $a1/b2$ is incompatible with merging $a2/b2$.

Each node in the compatibility graph is given a weight, w, corresponding to the area reduction associated with applying the given merge. This area reduction is calculated by synthesizing the primitive nodes used in the subgraphs and determining their area. For example, node $a1/b2$ represents merging $a1$ with $b2$. If this merging were applied, the resulting merged subgraph would only contain one adder for both of these nodes. So, the area reduction is the area of one adder.

To find the merging with the lowest area overhead, the maximum weight clique of this compatibility graph is calculated. The maximum weight clique of a graph is the set of fully connected nodes which have the largest sum of weights. In Fig. 5d the maximum weight clique is the set of nodes highlighted in blue. Using the maximum weight clique of the compatibility graph, the lowest cost merging of the two subgraphs can be reconstructed. This resulting merged graph is shown in Fig. 5e. Note that a multiplexer is added to enable multiple paths from nodes $a0$ to $a1$ and $b1$ to $b2$.

While this technique allows for efficiently merging frequent subgraphs from an application/domain, there is still the question of which and how many subgraphs to merge. We use maximal independent set analysis to identify interesting subgraphs mined from the application, which are ranked by MIS size so the most interesting subgraphs are considered first. We can generate more specialized PEs by adjusting the number of subgraphs merged together, which allows a designer to quickly explore different degrees of specialization.

4 APEX DESIGN SPACE EXPLORATION FRAMEWORK

We build the APEX framework (Fig. 6) on top of an open-source agile hardware design flow [3, 16] to create CGRAs with specialized PEs generated from our application analysis flow. Fig. 1 shows the high-level architecture of the CGRAs we generate. It contains an array of PE and memory (MEM) tiles connected through a statically configured interconnect.

The APEX framework consists of the following steps:

(1) **Application Frequent Subgraph Analysis**
   (a) The application analysis flow (Section 3) performs subgraph mining and maximal independent set analysis, generating an ordered list of frequent subgraphs.
   (b) Subgraph merging merges several frequent subgraphs to generate a candidate PE graph, which we automatically convert into a PE specification in PEak [3] DSL.
   (c) An automated pipelining tool then pipelines the PEs.

(2) **PE Hardware and Mapper Generation**
   (a) The PEak compiler takes in the PEak specification of a PE and generates the PE RTL Verilog and the rewrite rules for the application mapper.
   (b) The PE RTL is fed into CGRA RTL Verilog generation to generate the final CGRA hardware.

(3) **Application Mapping and Application Pipelining**
   (a) The application mapper, using the rewrite rules, generates a covering of the application CoreIR graph using PEs, while trying to minimize the number of PEs used and maximize utilization of PE's hardware resources.
   (b) An automated pipelining tool pipelines the mapped application graph using register file pipelining.
   (c) From the pipelined mapped graph, we generate the CGRA configuration bitstream and simulate the CGRA Verilog using Synopsys VCS.

(4) **PE/CGRA Synthesis and Evaluation**
   (a) We synthesize the PE and CGRA Verilog and evaluate performance, energy, and area of the resulting circuits.

Next, we describe in more detail how we use the PEak DSL to generate the hardware description of PEs as well as the rewrite rules for mapping to them. We then describe the optimizations we added to the PEak system to perform automatic pipelining of the PEs and applications to improve performance.

4.1 Using PEak to Generate PE Hardware and Rewrite Rules

We use PEak [3], an open-source python-embedded domain-specific language for specifying processors. A PEak specification defines the processor configuration, declares state, and describes the semantics of each instruction as a function from inputs and current state to outputs and next state. PEak uses multiple interpretations to generate a functional model, a formal model, and RTL Verilog from a single specification. A PEak program is a Python program, so it can be executed as a functional model. A formal model can be generated by symbolically executing the program with SMT (Satisfiability

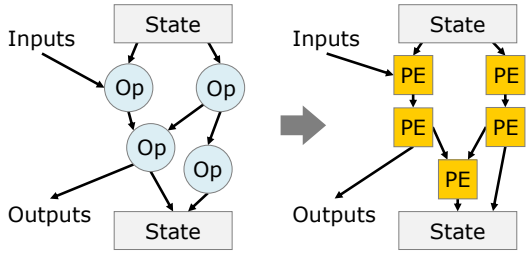

Figure 7: The input application, shown on the left, is composed of IR operations. Instruction selection transforms the dataflow graph of IR operations into a dataflow graph of PEs.

Modulo Theories) variables [5]. Finally, RTL is generated using the hardware description language, Magma [25]. The formal model of a PE makes it possible to automatically map an application to a CGRA with a PE generated using this framework. This mapping process is done in two main steps: rewrite rule synthesis, and instruction selection.

4.1.1 Rewrite Rule Synthesis. Rewrite rules specify how a PE must be configured to perform an operation in an application. For example, a rewrite rule for a two-input add operation would contain values of the PE configuration signals ensuring that for every value of the two inputs, A and B, the output of the PE is $A + B$. Rewrite rules also define which input of the PE is assigned to each operation input and which output of the PE is assigned to the output of the operation.

As described in [11], we can leverage the formal model generated by PEak and the formal model of CoreIR to synthesize these rules. This is achieved by constructing an SMT query that searches for a configuration of the PE that matches the behavior of one or many application operations (i.e., a subgraph). This query is fairly straightforward: does there exist a configuration x for PE P such that for every input y to operation Op, their outputs are equal. More formally:

$$\exists x \in X \, \forall y \in Y : P(x, y) = Op(y)$$

where X is the set of valid configurations of the PE, and Y is the set of possible inputs. Additionally, we must consider all valid permutations of input and output ports. We use the SMT solver Boolector [7] to solve this query. We synthesize rewrite rules for every operation necessary to execute any application that we want to run on the CGRA. In addition, rewrite rules for all complex operations extracted during application analysis and merged into the PE architecture are synthesized.

4.1.2 Instruction Selection. Given this set of rewrite rules, the next stage in application mapping is instruction selection. After the application is compiled to CoreIR, we need to transform the dataflow graph of IR instructions into a dataflow graph of PEs (Fig. 7). We use a version of LLVM's instruction selection algorithm [6] to do this. For each rewrite rule, we determine if the operation matches any subgraph of the application and greedily apply the rule. More complex rewrite rules are considered first; then the algorithm moves on to simpler rules. After attempting to apply all of the rules, we have fully transformed the graph and finished mapping the application.

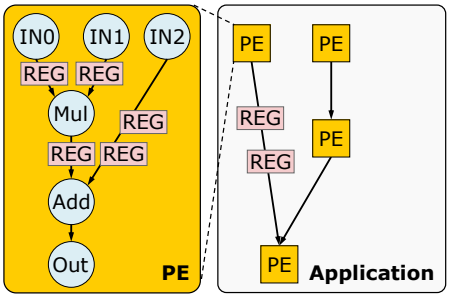

Figure 8: Branch delay matching when mapping to pipelined PEs. A PE with a two cycle latency from inputs to outputs is shown on the left. The mapped application compute kernel after branch delay matching is shown on the right.

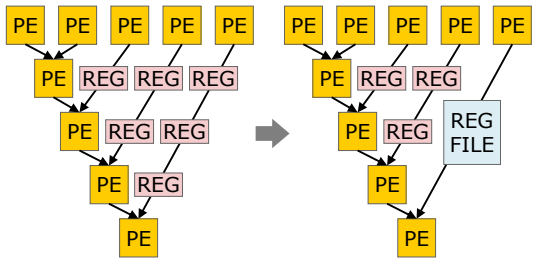

Figure 9: Long chains of pipelining registers are replaced with a register file that functions as a FIFO with a delay of three. Pipelining using register files reduces the number of pipelining registers and improves routability of the application.

4.2 Automated PE Pipelining

Our framework can produce arbitrarily complex PEs with long chains of operations. In order to meet a high target frequency (~1 GHz) during physical design, pipelining may be needed to break long delay paths through the PE. As this is an automated DSE framework, an automated pipelining method is needed to determine how many pipelining stages each PE should have, and where to place the pipelining registers to optimally break the long delay paths.

We use two algorithms to achieve automated pipelining. The first uses an algorithm similar to static timing analysis to model the critical path delay reduction for every additional pipelining stage in order to determine the appropriate number of stages [14]. We use the critical path model to iteratively increase the number of pipelining stages, determining when adding another stage gives a significant benefit. A second algorithm is needed at every iteration to retime the registers into the optimal positions for greatest critical path reduction. Register retiming is a well studied problem. For this framework we implemented the algorithm presented in [8].

4.3 Automated Application Pipelining

Mapping applications to pipelined PEs requires additional consideration during the application mapping process. For example, in Fig. 8 the PE contains pipelining registers, so it takes two cycles to

produce an output. In the mapped application, two registers need to be added in order to preserve functionality.

More generally, a branch delay matching algorithm is needed to ensure that the application is pipelined correctly when mapping to pipelined PEs. The algorithm traverses the application graph from inputs to outputs, keeping track of the number of cycles that a piece of data would take to arrive at the inputs of every PE and memory tile. If at any tile there is a mismatch between the arrival times of its input data, pipeline registers must be added to the shorter path to balance the arrival times.

In some applications, particularly those with long chains of operations, branch delay matching will insert many registers. During the placement and routing stage of the application compiler, these registers need to be placed in the interconnect of the CGRA (our switchboxes have configurable pipelining registers on every track), and if too many registers are added during branch delay matching, placement and routing will fail. In such cases, we employ an alternative approach for pipelining that utilizes the registers files that are present in the PE tiles. We substitute long chains of pipeline registers that may be added during branch delay matching with register files acting as FIFOs, as shown in Fig. 9. The transformation from a chain of registers to a register file is fully automated in our framework. We apply the transformation for register chains greater than length 2. If desired, the designer can adjust the cutoff point (the number of registers in the chain) for when the transformation is applied.

5 RESULTS

We evaluate APEX on the applications listed in Table 1. Our comparison baseline is the CGRA and the PE from [3], shown in Fig. 1. This PE contains an integer arithmetic unit and can perform bit operations using a look-up table (LUT). It is a general PE that can execute most applications and is not specialized for any particular domain. Each memory tile contains two banks of 2KB SRAM, address generators, and controllers. All tiles have a switch box (SB) and multiple connection boxes (CBs). The SB has five incoming and outgoing 16-bit routing tracks in each direction (north, south, east, west), that connect the tile to its neighbors. The CBs connect memory core and PE core inputs to the routing tracks. All CGRAs evaluated in this section are homogeneous, within one CGRA all PE tiles are identical and all memory tiles are identical.

Using APEX, we generate the following PE variations for each application by indicating which subgraphs are merged:

- PE 1: The first PE variation is the baseline PE but with only the operations necessary for the application.
- PE 2: The second variation merges the subgraph with the largest maximal independent set (MIS) with PE 1.
- PEs 3+: Further variations additionally merge other subgraphs into the PE architecture in the order of their MIS size. The last variation for an application is the most specialized PE possible without increasing the area or energy of the application running on the CGRA.

We analyze the effect of PE specialization at two levels: PE core level, which considers the PE's arithmetic and logical units only, and CGRA level, which includes all components of the CGRA: PEs, memories, and interconnect.

Table 1: Applications used for our DSE framework evaluation. IP: Image Processing, ML: Machine Learning.

Application	Domain	Description
Camera Pipeline	IP	Transforms camera data into an RGB image
Harris Corner	IP	Identifies corners within an image
Gaussian Blur	IP	Blurs an image
Unsharp	IP	Sharpens an image
ResNet Layer	ML	Residual neural network layer
MobileNet Layer	ML	Neural network layer for low-power devices

5.1 Specializing a PE for Camera Pipeline

Camera pipeline is a complex image processing application. It denoises, demosaics, color corrects, and color curves an image. It uses all the operations in the baseline PE except for left shift and bitwise logical operations and needs 90 primitive operations to compute one output pixel. For our evaluation, 4 output pixels are computed in parallel to maximally utilize the 32×16 CGRA. Fig. 10 (top left) shows the subgraphs used in the PE variations for camera pipeline and the final architecture used for the most specialized PE (PE 4/PE Spec).

Fig. 11 shows the total area (PE core area × number of PEs used by the application) and the total PE core energy of the different PE variants specialized for camera pipeline. Specializing the PE for camera pipeline results in up to 78% and 68% less area and energy, respectively, than the baseline PE. Table 2 shows the performance per mm^2 of the CGRA with each of the PE designs. From the baseline to PE 4, performance per mm^2 increases by 4×, largely due to the decrease in the total area consumed by the PEs for application execution.

5.2 Generalizing a PE for Image Processing Applications

Designing a PE that benefits all of the image processing applications (PE IP) requires balancing frequent operations from all applications. Fig. 12 shows how the degree of specialization affects PE area and energy using three different PEs generated using APEX (PE IP, PE IP2 and PE IP3). Fig. 10 shows which subgraphs are used in each variation: PE IP2 contains one more subgraph from each application than PE IP does, and PE IP3 specializes more for camera pipeline over the others.

Fig. 12 shows that merging too many subgraphs (PE IP2) can increase area and energy versus PE IP. PE IP3 shows that an unbalanced subgraph merge yields rewards for the application(s) it is more specialized for, but has negative consequences for others. PE IP has a good balance between common operations of all applications, achieving a 22% to 33% decrease in area and a 69% to 82% decrease in energy compared to the baseline PE.

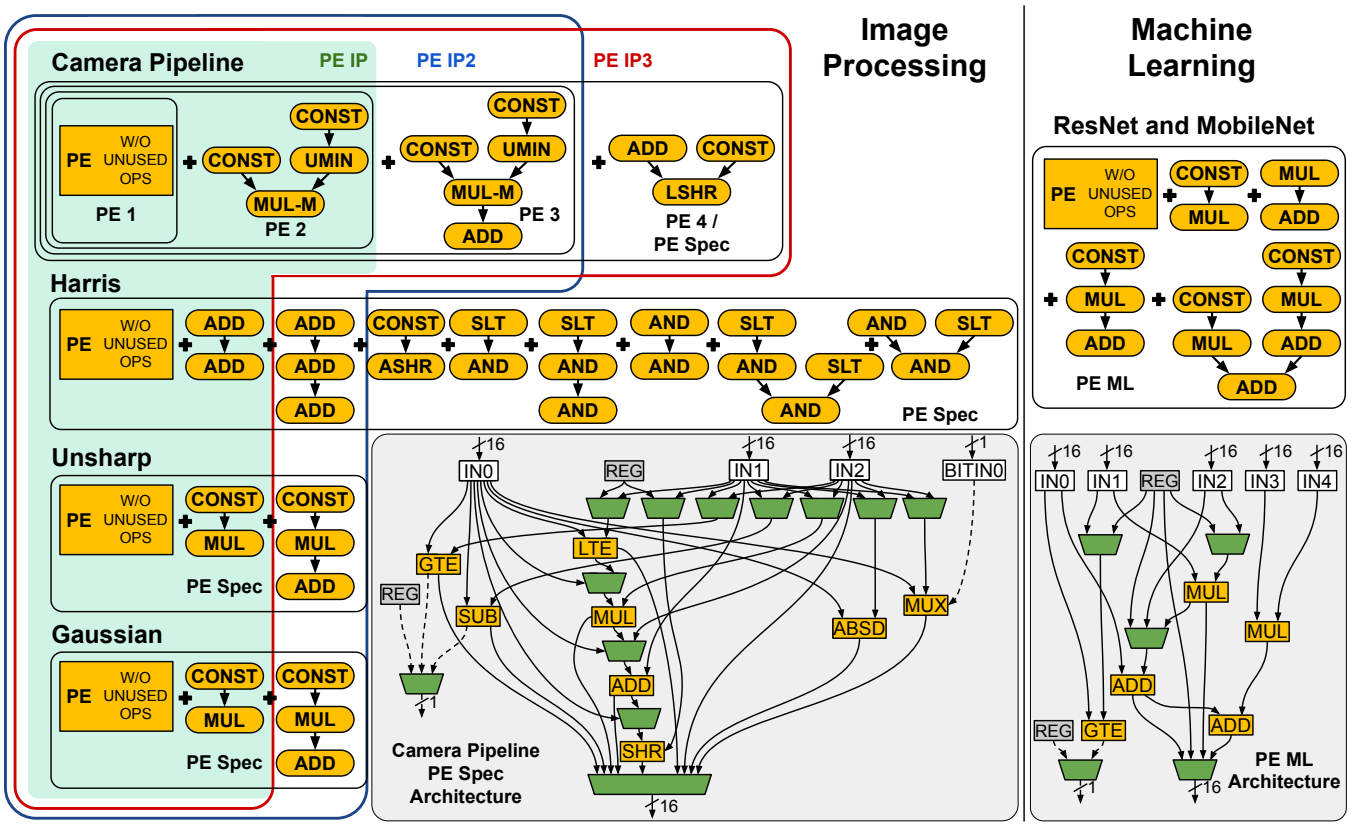

Figure 10: *Left*: Subgraphs merged to form PE variants 1 through 4 for camera pipeline, and the architecture of the most specialized camera pipeline PE (PE 4). The subgraphs used in PE Spec for unsharp, Harris, and Gaussian are also shown. The shading and outlines show which subgraphs are used to create the different PE IP variations evaluated in Fig. 12. *Right*: Subgraphs used in PE ML and architecture of the PE used in ML CGRA to compare with Simba, an ML accelerator.

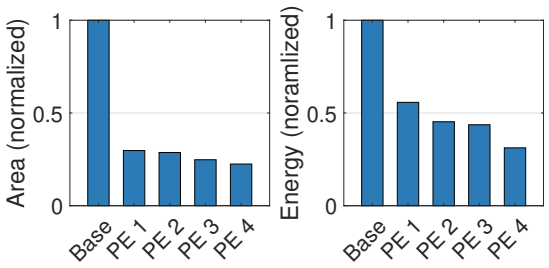

Figure 11: Energy and area results as the CGRA is increasingly specialized for camera pipeline.

Next, we want to demonstrate that PE IP is not only specialized to the four applications analyzed, but rather to the image processing application *domain*. To do this we introduce three new applications from the domain which were not analyzed during the creation of PE IP: (1) Laplacian pyramid, which is a linear invertible image representation consisting of a pyramid of image representations at different resolutions, (2) stereo, which transforms two images from left and right eye perspectives into a depth map, and finally (3) FAST corner detection [21], which identifies corners in an image

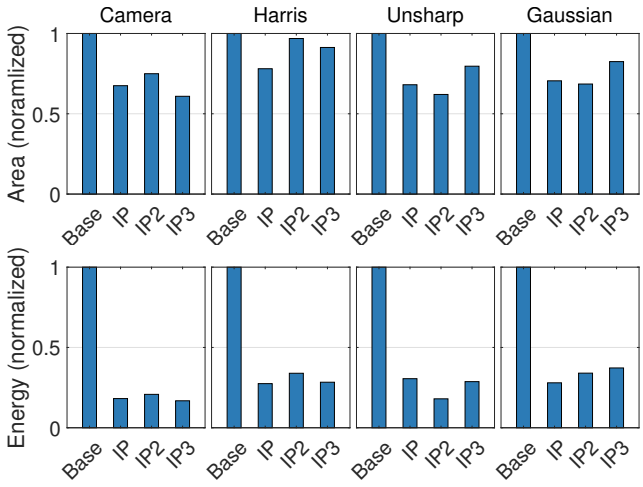

Figure 12: Comparison of three different PE IP variations that contain varying number of frequent subgraphs from each application.

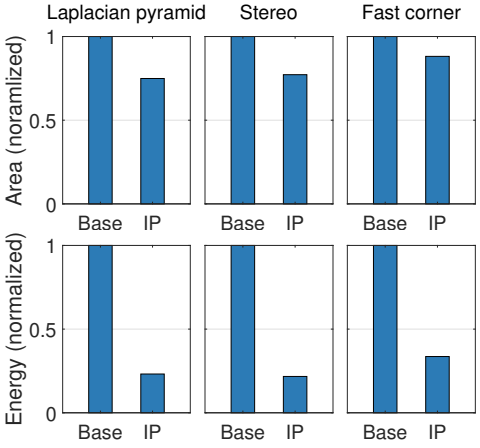

Figure 13: Comparison of the baseline PE and PE IP used when executing applications not seen during application analysis.

using an algorithm that is much simpler and faster than Harris corner detection.

Fig. 13 shows that PE IP executes these new applications much more efficiently than the baseline PE. While the area and energy decreases are not quite as large as the previous experiment, the benefits are still significant, with a 12% to 25% decrease in area and a 66% to 78% decrease in energy. This experiment shows that our approach of extracting frequent subgraphs from several applications and merging them to create PEs results in efficient "domain-specialized" rather than only "application-specialized" architectures.

5.3 CGRA-Level Evaluation of Generated PEs

We replicate the process of analyzing and generating PEs for the remaining image processing applications to create the most efficient specialized PE (PE Spec.) for each application. In addition, as described in the previous section, we merge subgraphs from all image processing applications to create PE IP, a PE specialized for image processing. The subgraphs used in PE IP are shaded green in Fig. 10.

In the following sub-sections, we report three different levels of results: post-mapping, post-place-and-route, and post-pipelining. When using APEX to create a PE design, generating post-mapping results is a quick and easy way to estimate the effect of specialization on the CGRA design. These results are obtained within minutes and represent the data that one would use to determine which PEs are interesting enough to investigate further. The post-place-and-route results are more accurate and more time consuming to produce. These results are generated within hours and represent the data that one would use to determine the effect specialization has on the CGRA as a whole. Finally, post-pipelining results are the most accurate and take into account the runtime of each application.

5.3.1 Post-Mapping Results. Fig. 14 shows the post-mapping evaluation of increasingly specializing the PE. These experiments only show the contributions of the PEs (no memory tiles or interconnect). Optimizing the PE for all image processing applications (PE IP) results in a 22% to 33% decrease in total PE area across all four

Table 2: Performance of camera pipeline implemented using various PE designs on a CGRA with a clock period of 1.1ns. Execution time is the time between the first input pixel going into the pipeline to the last output pixel coming out of the pipeline for a 1920×1080 image.

PE Variant	# PEs	Area/PE (μm^2)	Total Area (μm^2)	Performance (Frames/ms/mm^2)
PE Base	232	988.81	229,403	1.26
PE 1	232	294.18	68,249	4.22
PE 2	196	335.79	65,814	4.22
PE 3	172	330.58	56,859	4.88
PE 4	152	339.09	51,541	5.02

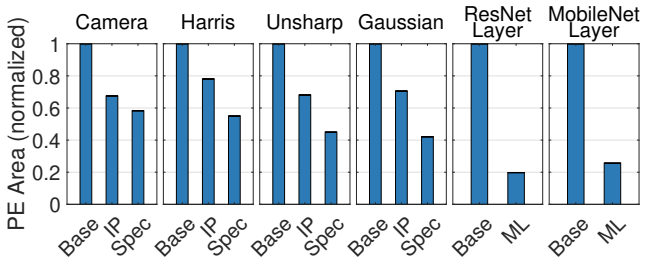

Figure 14: Post-mapping comparison of the baseline PE, PE IP for image processing, PE ML for machine learning, and PE Spec (this PE is specialized for each application individually).

image processing applications compared to the baseline. PE Spec. yields further benefits, with as much as 58% area reduction over the baseline. Optimizing the PE for all machine learning applications (PE ML) results in a 74% to 80% decrease in area across the machine learning applications compared to the baseline.

5.3.2 Post-Place-and-Route Results. PE specialization can also reduces the area and energy spent in the interconnect. Each PE variation has the same switch box (SB) design. The number of connection boxes (CBs) scales with the number of inputs to the PE. The post-place-and-route results are shown in Fig. 15.

Specializing the PE reduces the interconnect area across all applications because fewer PE tiles are needed. For example, with PE Spec. for camera pipeline, SB area and energy reduce by 35% and 26%, respectively, compared to the baseline. CB area and energy also exhibit similar trends – up to a 13% and 45% reduction in area and energy, respectively. In some cases, however, CB area may increase because the specialized PEs may have larger number of inputs. A Harris-specialized CGRA sees a 44% increase in CB area, but each active PE does not necessarily use all 16-bit and 1-bit inputs and experiences a 13% decrease in CB energy. ML applications see a 22% to 39% decrease in area and 16% to 59% decrease in energy compared with the baseline CGRA.

5.3.3 Post-Pipelining Results. As described in Section 4.2, we automatically pipeline PEs that have long critical paths. Additionally, we automatically pipeline the applications running on the CGRA. The maximum frequency of each PE is determined through synthesis, and the maximum frequency of the application is determined using

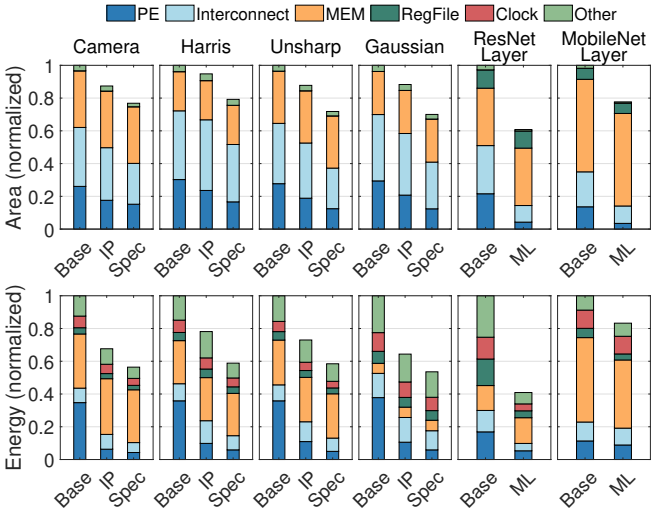

Figure 15: Post-place-and-route comparison of CGRA variants with the baseline PE, PE IP, PE ML, and PE Spec (specialized for each application individually). These results include the energy and area contribution from the interconnect.

Table 3: Post-pipelining resource utilization of CGRAs with each PE variant for each application (corresponds to Fig. 16). Routing-only tiles are CGRA tiles that data has to pass through, but whose PE or memory is not being used.

Application	#PE	#MEM	#RF	#IO	#Reg	#Routing Tiles
Baseline						
Camera	232	39	0	28	187	79
Harris	192	17	0	10	408	39
Unsharp	303	39	180	27	93	86
Gaussian	140	14	0	42	266	120
ResNet Layer	132	24	64	11	71	22
MobileNet Layer	112	52	52	17	304	69
PE IP						
Camera	196	39	0	28	187	89
Harris	170	17	0	10	286	69
Unsharp	234	39	180	27	93	66
Gaussian	112	14	0	42	210	79
PE Spec						
Camera	152	38	0	28	207	82
Harris	136	17	0	10	258	72
Unsharp	192	39	180	27	93	69
Gaussian	98	14	0	42	308	71
PE ML						
ResNet Layer	40	24	64	11	71	24
MobileNet Layer	44	52	52	17	356	63

static timing analysis run on the placed and routed application on the CGRA.

In Fig. 16, we report both pre-pipelining and post-pipelining area, energy and performance results for our suite of applications. In Table 3, we report the utilization of the components of the CGRA when running these applications. Pipelining dramatically decreases the critical path delay in both the PEs and the application as a whole. Applications running on PE-IP achieved a 6.9× to 12.5× increase in performance per mm^2 compared to the un-pipelined PE-IP applications. While running a ResNet layer, PE-ML has a modest increase in performance per mm^2. This is because intra-PE pipelining is done for all four versions of the CGRA running the layer. This is not a fundamental limitation in the compiler, but rather an optimization that we implemented in the scheduler that is turned on by default to enable more efficient execution of ResNet.

Performance is generally not affected as significantly by specialization. Specializing the PEs has the indirect effect of reducing the number of PEs in the application critical path and therefore decreasing the number of clock cycles taken to run the application. However, our applications are heavily unrolled, so this effect is weak, and increases in performance per mm^2 are due to lower area.

5.4 Comparison with Accelerators

Automated PE specialization allows a designer to quickly create energy-efficient and performant CGRAs that approach the energy-efficiency and performance of more dedicated accelerators. To show this, we compare the specialized CGRA designs generated in the previous section to the baseline CGRA, an FPGA, and an ASIC compiled directly from the application specification. For machine learning applications, we also compare with a state of the art neural network accelerator.

5.4.1 Image Processing Applications. Using Clockwork [15], we compile an FPGA and an ASIC implementation directly from a Halide application. The FPGA design is run on a Virtex Ultrascale+ VU9P FPGA, and the ASIC design is compiled to RTL using Catapult HLS and synthesized using Design Compiler in the same technology as the CGRA.

As shown in Fig. 17, compared to an FPGA, CGRA-IP provides significant area and energy reduction, and approaches the efficiency of an ASIC. Using an entirely automated PE DSE flow, we can generate designs that are 38× to 159× more energy-efficient than an FPGA, and 18% to 47% more efficient than a baseline CGRA. Our baseline CGRA and CGRA-IP achieve runtimes comparable to an ASIC.

5.4.2 Machine Learning Applications. As shown in Fig. 18, CGRA-ML achieves 14× lower energy than the FPGA implementation while running a ResNet layer. We also compare CGRA-ML against a state of the art neural network accelerator, Simba [22]. CGRA-ML approaches the area and energy consumption of Simba, while still being configurable. While running a ResNet layer, Simba is 16× more energy efficient than CGRA-ML.

6 RELATED WORK

Several previous approaches have been proposed for the automated specialization of accelerators to applications or application domains.

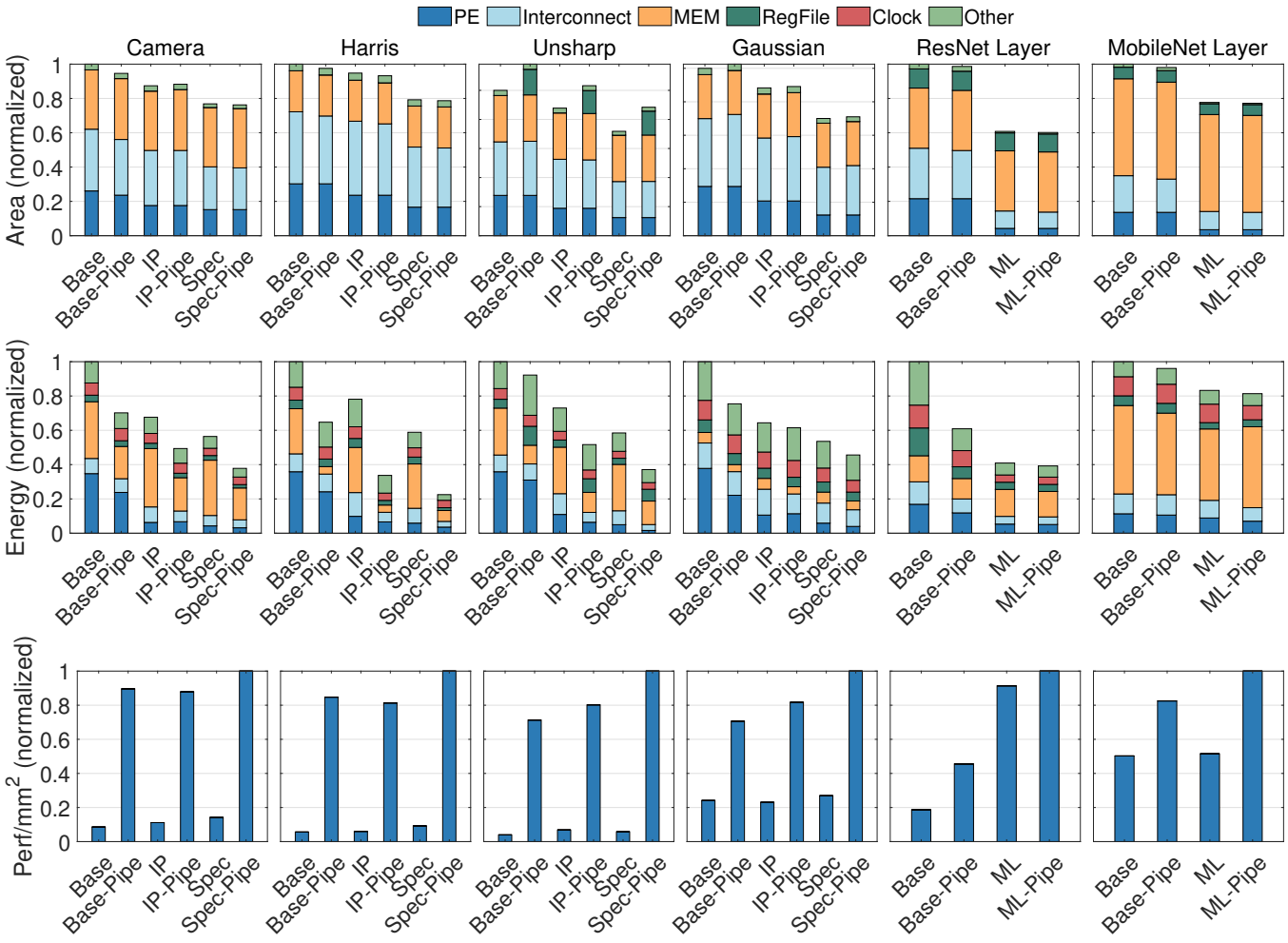

Figure 16: Pre- and post-pipelining area, energy, and performance/mm^2 comparison of the CGRAs with the baseline PE, PE IP, PE ML, and PE Spec (this PE is specialized for each application individually), showing the impact of PE and application pipelining.

[30] is the most similar to our work in that they aim to automatically specialize accelerator PEs to an application domain. One major difference between the two approaches is that the genetic algorithm for producing PE designs used in their work is quite slow compared to our subgraph mining and merging algorithm. While their iterative approach took 9 to 16 hours to generate PE designs, our approach can analyze the applications in the domain, generate a PE design, synthesize a compiler, and generate the RTL for the entire CGRA accelerator in minutes.

AURORA [23] presents an automated design space exploration tool for full CGRAs. This work generates CGRA designs and iteratively improves them with a simulated annealing based approach where the PEs have parameters that are mutated. While they use an application driven simulated annealing approach, our work is a much more directed search of the design space and explores more complex design decisions.

DSAGEN [29] presents an automated design space exploration tool for CGRA-like accelerators. They use an iterative design space exploration technique that removes unneeded resources based on a suite of applications. OVERGEN [17] proposes a framework to generate domain-specific overlays for FPGAs, where an application domain is analyzed to generate the most efficient overlays. In our work we focus on the design space of CGRA PEs in particular, and we consider more complex PE design decisions and generate PEs with more complex architectures.

REVMAP [4] proposes a framework for generating heterogeneous CGRA designs based around a set of optimizations. This work has a different goal from our paper and does not generate complex PE designs with multiple stages of arithmetic operations. To generate specialized PEs for an application, this work starts with an ALU and simplifies it by removing operations, resulting in low compute density, high interconnect overhead, and more difficult placement and routing.

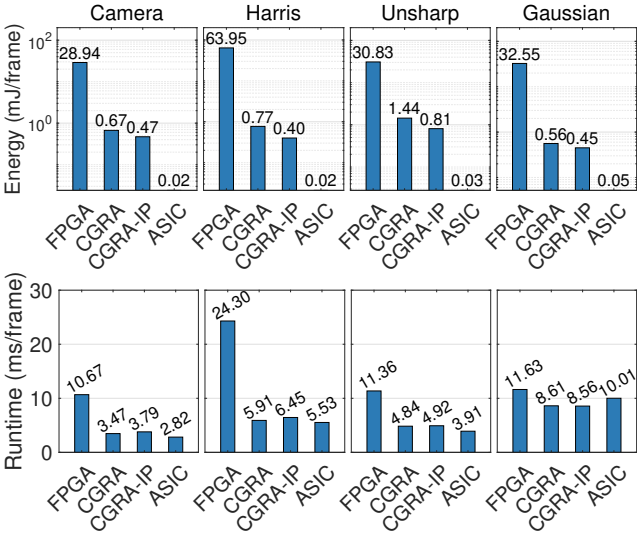

Figure 17: Energy and runtime comparison between an FPGA, the baseline CGRA, a CGRA with PE IP, and an ASIC.

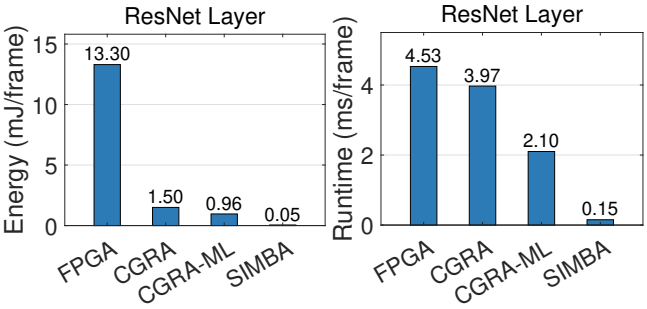

Figure 18: Comparison of machine learning applications running on an FPGA, the baseline CGRA, CGRA-ML, and Simba.

[27] and [28] use graph mining techniques to identify hot spots in applications and merge frequent operations into cores that augment a CPU. [10] uses performance driven analysis to extend the instruction set of an ALU. [2] present algorithms for identifying clusters of dataflow operations to help automate the specialization of processors. These papers do application analysis with similar goals as our work, but do not present design space exploration frameworks for full accelerators.

More generally, exploring the architecture of CGRAs has also been previously studied. Expression-grained reconfigurable array (EGRA) [1] explores hardware tradeoffs in CGRA computing elements using clusters of ALUs. However, the DSE is not guided by application analysis, so many designs need to be evaluated. Plasticine [19] is a parameterizable CGRA with interleaved compute units and memory units in an interconnect, though automated DSE is not included in this work.

In summary, the novelty of this work is the application of several existing graph analysis algorithms to automate CGRA PE design space exploration. We address the considerations of designing a CGRA, including exploring the tradeoffs between PE complexity and overheads introduced by the reconfigurable interconnect, and automatic intra- and inter- PE application pipelining. Another novelty of this work is connecting design space exploration with automatic compiler generation. Our use of SMT based rewrite rule synthesis results in a framework that is more flexible than previous approaches and enables the exploration of more varied and interesting PE designs.

7 CONCLUSION

We have presented a design space exploration framework that allows for automated specialization of CGRA PEs to an application or an application domain, enabling the creation of complex, high-performance PEs that lower overall energy and area costs. We demonstrate that specializing PEs results in 22% to 39% less area and 16% to 59% less energy compared to a general-purpose CGRA. These fast and efficient CGRAs with specialized PEs approach the efficiency of domain-specific accelerators.

8 ACKNOWLEDGEMENTS

This work was supported by the DSSoC DARPA grant, the Stanford AHA Agile Hardware Center and Affiliates Program, Intel's Science and Technology Center (ISTC), and the Stanford SystemX Alliance.

REFERENCES

[1] Giovanni Ansaloni, Paolo Bonzini, and Laura Pozzi. 2008. Design and Architectural Exploration of Expression-Grained Reconfigurable Arrays. In *2008 Symposium on Application Specific Processors*. https://doi.org/10.1109/SASP.2008.4570782

[2] Kubilay Atasu, Laura Pozzi, and Paolo Ienne. 2003. Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints. In *Proceedings of the 40th Annual Design Automation Conference* (Anaheim, CA, USA) *(DAC '03)*. Association for Computing Machinery, New York, NY, USA, 256–261. https://doi.org/10.1145/775832.775897

[3] Rick Bahr, Clark Barrett, Nikhil Bhagdikar, Alex Carsello, Ross Daly, Caleb Donovick, David Durst, Kayvon Fatahalian, Kathleen Feng, Pat Hanrahan, Teguh Hofstee, Mark Horowitz, Dillon Huff, Fredrik Kjolstad, Taeyoung Kong, Qiaoyi Liu, Makai Mann, Jackson Melchert, Ankita Nayak, Aina Niemetz, Gedeon Nyengele, Priyanka Raina, Stephen Richardson, Raj Setaluri, Jeff Setter, Kavya Sreedhar, Maxwell Strange, James Thomas, Christopher Torng, Leonard Truong, Nestan Tsiskaridze, and Keyi Zhang. 2020. Creating an Agile Hardware Design Flow. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. https://doi.org/10.1109/DAC18072.2020.9218553

[4] Thilini Kaushalya Bandara, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. 2022. REVAMP: A Systematic Framework for Heterogeneous CGRA Realization. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3503222.3507772

[5] Clark Barrett and Cesare Tinelli. 2018. Satisfiability Modulo Theories. In *Handbook of Model Checking*, Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). Springer International Publishing. https://doi.org/10.1007/978-3-319-10575-8_11

[6] Eli Bendersky. 2013. A Deeper Look into the LLVM Code Generator, Part 1. https://eli.thegreenplace.net/2013/02/25/a-deeper-look-into-the-llvm-code-generator-part-1

[7] Robert Brummayer, Armin Biere, and Florian Lonsing. 2008. BTOR: Bit-Precise Modelling of Word-Level Problems for Model Checking. In *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning (SMT '08/BPR '08)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1512464.1512472

[8] Pierre-Yves Calland, Anne Mignotte, Olivier Peyran, Yves Robert, and Frédéric Vivien. 1998. Retiming DAGs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (1998). https://doi.org/10.1109/43.736571

[9] Hong Cheng, Xifeng Yan, and Jiawei Han. 2010. *Mining Graph Patterns*. Springer US, Boston, MA. https://doi.org/10.1007/978-1-4419-6045-0_12

[10] Jason Cong, Yiping Fan, Guoling Han, and Zhiru Zhang. 2004. Application-Specific Instruction Generation for Configurable Processor Architectures. In *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays (FPGA '04)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/968280.968307

[11] Ross Daly, Caleb Donovick, Jackson Melchert, Rajsekhar Setaluri, Nestan Tsiskaridze Bullock, Priyanka Raina, Clark Barrett, and Pat Hanrahan. 2022. Synthesizing Instruction Selection Rewrite Rules from RTL using SMT. In *Conference on Formal Methods in Computer-Aided Design (FMCAD)*. 139–150. https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_20

[12] Ross Daly, Leonard Truong, and Pat Hanrahan. 2018. Invoking and Linking Generators from Multiple Hardware Languages using CoreIR. In *Workshop on Open-Source EDA Technology (WOSET)*. https://woset-workshop.github.io/PDFs/2018/a11.pdf

[13] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. 2014. GraMi: Frequent Subgraph and Pattern Mining in a Single Large Graph. *Proc. VLDB Endow.* (2014). https://doi.org/10.14778/2732286.2732289

[14] Robert. B. Hitchcock, Gordon L. Smith, and David D. Cheng. 1982. Timing Analysis of Computer Hardware. *IBM Journal of Research and Development* (1982). https://doi.org/10.1147/rd.261.0100

[15] Dillon Huff, Steve Dai, and Pat Hanrahan. 2021. Clockwork: Resource-Efficient Static Scheduling for Multi-Rate Image Processing Applications on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '21)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3431920.3439457

[16] Kalhan Koul, Jackson Melchert, Kavya Sreedhar, Leonard Truong, Gedeon Nyengele, Keyi Zhang, Qiaoyi Liu, Jeff Setter, Po-Han Chen, Yuchen Mei, Maxwell Strange, Ross Daly, Caleb Donovick, Alex Carsello, Taeyoung Kong, Kathleen Feng, Dillon Huff, Ankita Nayak, Rajsekhar Setaluri, James Thomas, Nikhil Bhagdikar, David Durst, Zachary Myers, Nestan Tsiskaridze, Stephen Richardson, Rick Bahr, Kayvon Fatahalian, Pat Hanrahan, Clark Barrett, Mark Horowitz, Christopher Torng, Fredrik Kjolstad, and Priyanka Raina. 2022. AHA: An Agile Approach to the Design of Coarse-Grained Reconfigurable Accelerators and Compilers. *ACM Transactions on Embedded Computing Systems (TECS)* (July 2022). https://doi.org/10.1145/3534933

[17] Sihao Liu, Jian Weng, Dylan Kupsh, Atefeh Sohrabizadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Rishabh Mani, Lucheng Zhang, Jason Cong, and Tony Nowatzki. 2022. OverGen: Improving FPGA Usability through Domain-specific Overlay Generation. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO56248.2022.00018

[18] Nahri Moreano, Edson Borin, Cid C. de Souza, and Guido Araujo. 2005. Efficient datapath merging for partially reconfigurable architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2005). https://doi.org/10.1109/TCAD.2005.850844

[19] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture for Parallel Patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/3079856.3080256

[20] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *SIGPLAN Not.* (2013). https://doi.org/10.1145/2499370.2462176

[21] Edward Rosten and Tom Drummond. 2006. Machine Learning for High-Speed Corner Detection. In *Computer Vision – ECCV 2006*, Aleš Leonardis, Horst Bischof, and Axel Pinz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/11744023_34

[22] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. 2019. Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture. In *MICRO*. https://doi.org/10.1145/3352460.3358302

[23] Cheng Tan, Chenhao Xie, Ang Li, Kevin J. Barker, and Antonino Tumeo. 2021. AURORA: Automated Refinement of Coarse-Grained Reconfigurable Accelerators. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. https://doi.org/10.23919/DATE51398.2021.9473955

[24] Russell Tessier, Kenneth Pocek, and André DeHon. 2015. Reconfigurable Computing Architectures. *Proc. IEEE* (2015). https://doi.org/10.1109/JPROC.2014.2386883

[25] Lenny Truong and Pat Hanrahan. 2019. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity. In *3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA (LIPIcs)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.SNAPL.2019.7

[26] Artem Vasilyev, Nikhil Bhagdikar, Ardavan Pedram, Stephen Richardson, Shahar Kvatinsky, and Mark Horowitz. 2016. Evaluating Programmable Architectures for Imaging and Vision Applications. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.2016.7783755

[27] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. 2010. Conservation Cores: Reducing the Energy of Mature Computations. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1736020.1736044

[28] Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. 2011. QsCores: Trading Dark Silicon for Scalable Energy Efficiency with Quasi-Specific Cores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2155620.2155640

[29] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. 2020. DSAGEN: Synthesizing Programmable Spatial Accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA45697.2020.00032

[30] Max Willsey, Vincent T. Lee, Alvin Cheung, Rastislav Bodík, and Luis Ceze. 2019. Iterative Search for Reconfigurable Accelerator Blocks With a Compiler in the Loop. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019). https://doi.org/10.1109/TCAD.2018.2878194

Received 2022-10-20; accepted 2023-01-19