Cascade: An Application Pipelining Toolkit for Coarse-Grained Reconfigurable Arrays

Jackson Melchert, Yuchen Mei, Kalhan Koul, *Graduate Student Member, IEEE*, Qiaoyi Liu, Mark Horowitz, *Life Fellow, IEEE*, and Priyanka Raina

***Abstract*—While coarse-grained reconfigurable arrays (CGRAs) have emerged as promising programmable accelerator architectures, they require automatic pipelining of applications during their compilation flow to achieve high performance. Current CGRA compilers either lack pipelining altogether resulting in low application performance, or perform exhaustive pipelining resulting in high power and resource consumption. We address these challenges by proposing Cascade, an end-to-end open-source application compiler for CGRAs that achieves both state-of-the-art performance and fast compilation times. The contributions of this work are: 1) a novel post place-and-route (PnR) application pipelining technique for CGRAs that accounts for interconnect hop delays during pipelining but in a unique way that avoids cyclic scheduling and PnR, 2) a register resource usage optimization technique that leverages the scheduling logic in CGRA memory tiles to minimize the number of register resources used during pipelining, and 3) an automated CGRA timing model generator, an application timing analysis tool, and a large set of existing and novel application pipelining techniques integrated into an end-to-end compilation flow. Cascade achieves 8 - 34× lower critical path delay and 7 - 190× lower energy-delay product (EDP) across a variety of dense image processing and machine learning workloads, and 3 - 5.2× lower critical path delay and 2.5 - 5.2× lower EDP on sparse workloads, compared to a compiler without pipelining. Cascade mitigates the performance and energy-efficiency drawbacks of existing CGRA compilers, and enables further research into CGRAs as flexible, yet competitive accelerator architectures.**

***Index Terms*—Accelerator compilers, application pipelining, coarse-grained reconfigurable arrays (CGRAs), hardware accelerators.**

I. Introduction

COARSE-GRAINED reconfigurable arrays (CGRAs) have been widely studied in recent years as performant and efficient configurable accelerator architectures. A CGRA can achieve better performance and energy-efficiency than an FPGA owing to its coarser-grained computation units and interconnect, while still maintaining much more flexibility than an application-specific integrated circuit (ASIC). However, in order to achieve commercial utility, CGRAs must demonstrate performance and energy-delay product (EDP) that are competitive with ASICs. To do so, CGRAs need to execute applications at high clock frequencies, requiring carefully pipelined applications.

The problem is that existing CGRA compilers fail at this task. They attempt to tackle mapping, scheduling, placement and routing (PnR), and pipelining of an application all within one optimization step [6], [9], [11], [17], [20], [25]. This coupling between the various pieces in the compilation flow makes the search space very large, making the compilers slow, produce poor results, and not scale well to large CGRAs. The basic problem is that, in CGRAs, the programmable wiring has large relative delay that the pipelining must take into account, but since data waves need to be balanced, adding pipeline registers along one path can require adding hardware to many other paths. This additional hardware often changes the PnR of the application on the CGRA, which causes this process to restart.

To address this issue, we took inspiration from FPGA compilers and decoupled mapping, scheduling, placement, routing, and pipelining into largely independent steps. We build upon the work presented in [12], which takes this approach but only does wire-independent pipelining, to create a compiler called Cascade. Our compiler only needs minimal hardware support in the CGRA: configurable pipeline registers on the programmable wires, an interconnect with single-cycle multihop connections, and the ability to adjust the schedules of the memory tiles at a cycle level. Using this hardware, we add post-PnR pipelining, register absorption, and incremental rescheduling to a staged FPGA-style compiler, resulting in state-of-the-art application performance and compilation times.

The contributions of this article are:

1) We adapt FPGA and ASIC-like pipelining and register retiming techniques [2], [13] to register-scarce CGRAs, and propose a technique for absorbing registers into memory tiles without affecting the mapping, placement, and routing.
2) We propose a post-PnR pipelining technique that iteratively identifies the critical path in an application, breaks it by turning on interconnect registers, and performs rescheduling. Post-PnR pipelining accurately accounts for interconnect delays while avoiding cyclic rescheduling and PnR.

Manuscript received 13 November 2023; revised 28 February 2024; accepted 5 April 2024. Date of publication 17 April 2024; date of current version 20 September 2024. This work was supported in part by SRC JUMP 2.0 PRISM Center; in part by NSF CAREER under Award 2238006; in part by the Defense Advanced Research Projects Agency DSSoC; in part by the Stanford Agile Hardware (AHA) Center; in part by the Stanford SystemX Alliance; and in part by the Apple Stanford EE Ph.D. Fellowship in Integrated Systems. This article was recommended by Associate Editor M. D. Santambrogio. *(Corresponding author: Jackson Melchert.)*

The authors are with the Department of Electrical Engineering, Stanford University, Stanford, CA 94305 USA (e-mail: melchert@stanford.edu).

Digital Object Identifier 10.1109/TCAD.2024.3390542

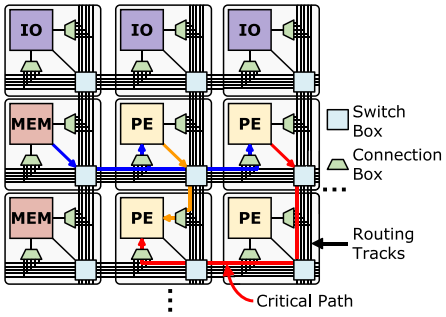


Fig. 1. CGRA architecture, and examples of paths with various delays for an application. The longest path found by STA is the critical path, that determines the maximum clock frequency at which the application can run on the CGRA.

3) An end-to-end open-source CGRA compiler [1],[1] called Cascade, which has (a) an automatic CGRA timing model generator, (b) a static timing analysis (STA) tool that uses the timing model to determine the critical path of an application on a CGRA, and (c) a large set of existing and our proposed pipelining techniques integrated into an end-to-end flow.

Cascade targets a large class of CGRAs like [3], which have big tile arrays, an interconnect that allows for single-cycle multihop connections from any tile to any other tile, and configurable pipelining registers on every hop of the interconnect. Cascade enables 8 - 34× lower critical path delay and 7 - 190× lower EDP on dense workloads, and 3 - 5.2× lower critical path delay and 2.5 - 5.2× lower EDP on sparse workloads, compared to a compiler without pipelining. This article provides a critical set of tools for the creation of high-performance compilation infrastructure for CGRAs.

The rest of this article is organized as follows. After introducing some background, we describe our novel CGRA application pipelining techniques, 1) post-PnR pipelining that avoids cyclic rescheduling and PnR in Section III, and 2) our method for optimizing register resource consumption in Section IV. Finally, 3) we present our end-to-end Cascade compiler in Section V, and evaluate it in Section VI.

II. Background

We first summarize the class of CGRAs we target, the compiler infrastructure we build upon, static scheduling of CGRA applications, and STA.

A. CGRA Architecture

CGRAs are typically composed of several processing element (PE) tiles, memory (MEM) tiles, and input/output (IO) tiles, arranged in a grid, as shown in Fig. 1. The tiles communicate via a configurable interconnect, which is comprised of several horizontal and vertical routing tracks, connection boxes that bring inputs into the tiles from the routing tracks, and switch boxes that take outputs from the tiles and route them onto the routing tracks in different directions. In this article, we primarily target CGRAs with large tile arrays (e.g., 512 tiles), a configurable interconnect that allows for single-cycle multihop connections from any tile to any other tile, and configurable pipelining registers within every switch box. This style of CGRA is similar to an FPGA in that the configuration of the accelerator determines its maximum operating frequency. An FPGA chip can be fabricated with a maximum frequency of 1 GHz, but an FPGA compilation tool may compile an application to the FPGA that has a maximum frequency of 200 MHz. The same is true for this class of CGRAs.

B. CGRA Compiler

We develop our pipelining framework on top of an existing open-source agile hardware design toolchain that encompasses application specification, scheduling, mapping, place and route, and bitstream generation [12], summarized in Fig. 6.

First, the application goes through a compiler which produces a dataflow graph with primitive compute operations like add, multiply, shift, etc. In this representation, the application consists of compute kernels connected by "abstract" memories. Next, the compute kernels get transformed by the compute mapping tool into dataflow graphs of PEs and registers. After compute mapping, the mapped compute kernels and the abstract memories get fed into the scheduling and memory mapping tool that produces a final dataflow graph of PEs, memory tiles, IO tiles, and registers. Next, this fully mapped application goes into the PnR tool which produces a placed and routed result. The final stage of this compiler is bitstream generation, which consumes the placed and routed dataflow graph and generates the configuration bitstream to run the application on the CGRA.

C. Static Scheduling of CGRA Applications

Many image processing and machine learning applications have statically analyzable access patterns. This characteristic enables CGRA compilers to statically schedule such applications (e.g., all memory accesses can be scheduled at compile time). The application scheduling process turns the multidimensional loop statements in these applications into operations (load/store) executed on the memory tiles. The open-source CGRA compiler we build upon [15] generates a cycle-accurate schedule which gives each statement in the application's iteration domain a 1-D timestamp, which represents the hardware's runtime behavior and enables pipelined parallelism. This static schedule overlaps data transfer with computation and extracts data reuse to reduce latency and improve energy efficiency.

D. Static Timing Analysis

STA is a standard technique for determining the critical path, that is the path with the maximum delay, through a circuit [10]. STA can be applied to any directed acyclic graph (DAG), and it involves iterating through every node in the DAG in reverse topological order and calculating the arrival time at each node.

[1]This project includes many techniques spread across several subrepositories. Post-PnR pipelining and the application STA tools are in archipelago, rescheduling is done using clockwork, register absorption and compute pipelining are done using MetaMapper, broadcast signal pipelining and hardening broadcasts are done in garnet, placement function cost optimization is done in cgra_pnr, and low unrolling duplication is done in aha.

The arrival time at node N is

$$\text{arrival}[N] = \text{delay}[N] + \max(\text{arrival}[\text{predecessors}[N]) \quad (1)$$

where delay[N] is the delay through node N in the circuit. The arrival time at a node that breaks the critical path, such as a pipeline register, is 0. Once the arrival time is calculated for each node, the critical path length of the DAG is largest arrival time at any node. The critical path itself can be found by tracing back through the graph, following the nodes that contribute most to the path delay.

III. Post-Place-and-Route Pipelining

Most existing FPGA/ASIC compilers (with decoupled stages) do not change pipelining decisions after the scheduling and mapping stage [4], [5], [23]. Particularly, they do not change pipelining decisions while/after PnR. This means that they do not accurately take into account the effect of wire/interconnect hop delays on pipelining (only coarse wire load estimates are used). This is done in order to keep the scheduling and mapping problem decoupled from PnR, to keep the compilation problem tractable and compile times low — since introducing registers after PnR will change the application schedule, and then one may need to reschedule and run PnR again, and this process may never converge. However, not accurately accounting for interconnect hop delays during pipelining leads to suboptimal clock frequencies. It is partly due to this reason that typical clock frequencies that applications achieve on FPGAs hover around 200–300 MHz, even though the maximum achievable frequency on the hardware may be 3 times higher [8].

Unlike prior work, we perform pipelining that accounts for the interconnect hop delays post-PnR, but in a unique way that avoids cyclic scheduling and PnR, as described below.

A. Post-Place-and-Route Pipelining Algorithm

Post-PnR pipelining iteratively identifies the critical path and inserts pipelining registers to break it. After PnR is complete, we know exactly where each tile will be placed on the array and where the nets will be routed. Using the timing model and application STA tool we designed for CGRAs (detailed later in Sections V-A and V-B), we determine the critical path delay of the application. The CGRA timing model contains delays for both the PE operations and memories, as well as the delays of interconnect hops. Additionally, we can use STA with back-tracing to determine what the critical path is.

The interconnect of our CGRA has configurable pipelining registers within every switchbox of the array. These pipelining registers exist on every 16-bit and 1-bit track going out of the switchbox in each of the four directions. After PnR, we can modify the configuration of the CGRA to turn on individual pipelining registers at places we want to break long paths. Note that *we only turn on registers that are unused on the existing routes in the application PnR graph, and not elsewhere, since we do not want to run PnR again* (this process may not converge if we rerun PnR). Adding registers on existing routes affects the execution of the application, so we must do branch delay matching in order to maintain application functionality.

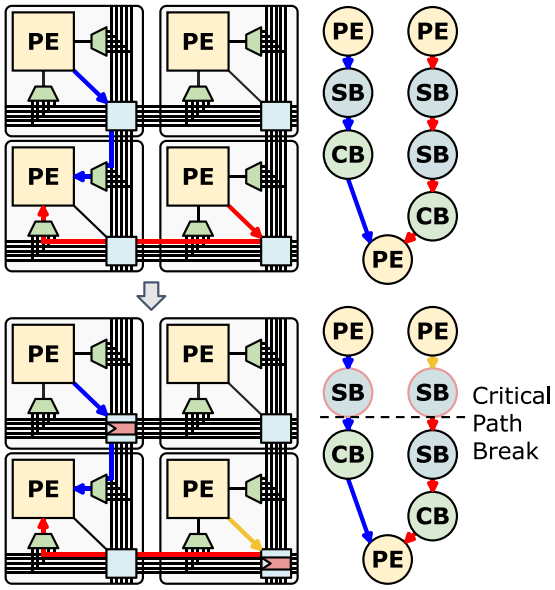


Fig. 2. Post-PnR pipelining takes the place and route result represented as a dataflow graph and performs STA to identify the critical path. This path is then broken by enabling registers in the switch box (SB), and the graph is branch delay matched.

Branch delay matching matches the cycle arrival times of every piece of data arriving at every functional element in the application. We use an algorithm similar to STA for branch delay matching, but instead of using the delay through each hardware element, we instead use the number of cycles each node takes to generate an output. If we find a node that has more than one unique arrival time, we must insert registers to ensure correct application execution. In the next section, we will discuss a technique to reduce the number of registers added during branch delay matching, which saves energy and frees up resources for other purposes.

For example, in Fig. 2, the red path is the longest in the application, so it is the critical path. During post-PnR pipelining, we break this path by turning on the pipelining register in the bottom-right switch box. Next, we run branch delay matching to ensure that the application still functions correctly, inserting pipelining registers to balance any paths if needed. In this example, we need to break the blue path to balance the arrival times of the two pieces of data at the final PE. Then, we analyze the application again and repeat the process until we cannot break any more paths or achieve the desired clock frequency.

B. Updating Application Schedule Post Pipelining

When adding pipeline registers to a statically scheduled CGRA application, the schedule needs to be updated to reflect any changes to the compute latencies. Because we ensure that the topology of the mapped application graph does not change as we perform pipelining, in the first round of application compilation we set all computation latencies to 0.

After an application finishes post-PnR pipelining, we know all of the compute kernel latencies. We send these updated

latencies to the static scheduler to incrementally update the configuration of each memory tile used in the application. The memory tiles send data to and receive data from the compute kernels in the application graph. They have controllers (with address and schedule or 'enable' generators) that read and write to memories based on the static schedule, and we update the delay registers in precisely these controllers. This allows us to do just enough rescheduling to maintain application correctness, while avoiding any change to the mapped application graph, and therefore the PnR result.

Finally, all the memories used by the application also need to be synchronized at the beginning of application execution, and that is done using a global flush signal. As we will explain later, this flush signal also needs to be pipelined. We use the flush arrival time after pipelining to adjust the static schedules so that the counters are all synchronized properly. If the flush signal arrives one cycle late, we need to adjust the starting cycle of the memory address generator to begin one cycle earlier, ensuring that the execution of the application is unchanged.

To the best of our knowledge, ours is the first work that discusses post-PnR pipelining in CGRAs and proposes a technique for incrementally updating the application schedule post pipelining in a way that does not result in cyclic scheduling and PnR. Other CGRA compilers do not have the flexibility to pipeline and reschedule applications after they have been placed and routed.

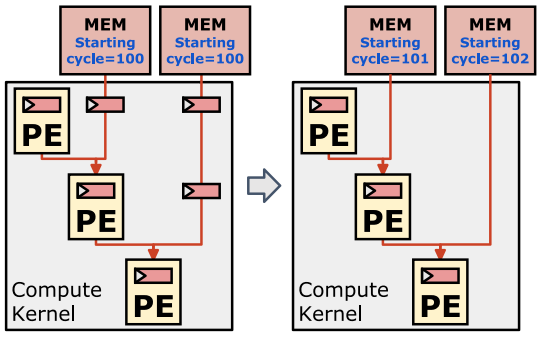


Fig. 3. Absorbing registers that exist inside compute kernels into memory tiles and rescheduling the application to adjust the starting cycles of the memory tile outputs.

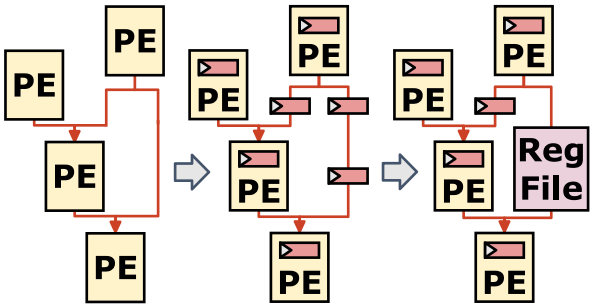


Fig. 4. First, compute pipelining is performed where registers at the inputs of PEs are enabled and all branches are delay matched. Next, chains of registers are absorbed into register files, configured to act as variable length shift registers.

IV. Optimizing Register Resource Usage

The registers added by pipelining and branch delay matching need to be placed on the configurable interconnect. This added resource requirement increases execution energy, and for large applications, may cause placement or routability issues. Therefore, we introduce a technique for absorbing registers into memory tiles and register files that dramatically reduces the register resource usage while maintaining the benefits of pipelining. Note that the registers we want to remove from the configurable interconnect are *only those that are* ***not*** *on the critical path*. These are typically introduced on noncritical paths when we perform branch delay matching after pipelining the critical path.

A register can be "absorbed" into a memory tile or a register file through the mechanism shown in Fig. 3. This process removes registers from the mapped application, so they do not need to be placed and routed on the configurable interconnect. This is possible because the static schedules of memory tiles and register files can be adjusted as described in the previous section. Removing a single register that is connected to the output of a memory tile is as simple as scheduling the memory to start outputting data one cycle later.

To take full advantage of this optimization, we modify the structure of compute kernels to favor the use of chains of operations rather than balanced trees of operations. Any reduction operation can be structured as a balanced or unbalanced tree. Unbalanced trees have chains of operations, which when pipelined, result in chains of registers. As shown in Fig. 3, these chains of registers can easily be absorbed into memory tiles.

In some cases, many pipelining registers might exist in a chain that cannot be absorbed into a memory tile. Many CGRA architectures, including our target architecture, include register files throughout the tile array. In these cases we utilize register files in PE tiles to act as variable length shift registers to eliminate these long chains of registers, thereby freeing up resources. To act as a variable length shift register, the register file needs to be configured to write to and read from the same address with an offset of one or more cycles. In Fig. 4, we show an example of this transformation for a chain of two registers. This transformation is applied to every N chain of registers, where N is a hyperparameter. In our experimental results, we use $N = 10$, as this struck a good balance between routability and energy-efficiency.

Note that this technique relies on the existence of register files and ability to change the configuration of the memory tiles in the array. However, the techniques for absorbing registers into memory tiles and register files are independent, so if a CGRA has a subset of the features of our target architecture, we can apply a subset of the techniques as well.

We evaluate the effect of this technique in Section VI-B.

V. Cascade

Finally, we design an end-to-end open-source CGRA compiler called Cascade, which like [12], has decoupled compiler

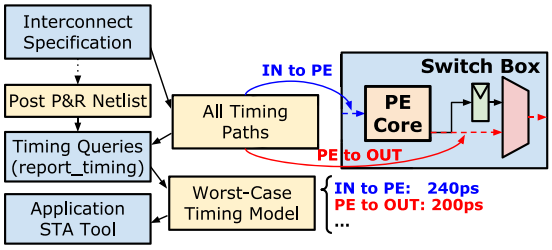


Fig. 5. Given an interconnect specification written in Canal [19], we automatically generate all paths of interest in a PE or memory tile. Using a commercial ASIC STA tool we can find the worst-case delay of these paths and use them in the application STA.

stages, but achieves higher performance. Cascade has 1) an automatic CGRA timing model generator, 2) a STA tool that uses the timing model to determine the critical path of an application on a CGRA, and 3) a large set of existing and our proposed pipelining techniques integrated into an end-to-end flow that works both for dense and sparse applications.

A. Automatic CGRA Timing Model Generator

We propose an automated methodology for generating a timing model of a CGRA, shown in Fig. 5. This model contains information about all significant delays in the CGRA, and is used later for application STA. To generate the model, we first specify our interconnect in Canal [19]. Canal is an interconnect specification language that gives us the ability to describe a wide range of CGRA interconnect topologies with a graph representation. We then add to Canal the ability to enumerate all possible data and clock paths that have significant delays.

Generally, a path within a CGRA (rather than a path at the boundary between the CGRA and the memory system) determines the maximum frequency of the accelerator. There are two components which have large contributions to path delay. First, there is significant delay associated with the functionality within a tile, which we call a core. For example, the PE tile core will include an ALU, and the memory tile core will include SRAMs. Second, on the CGRA, we can configure the interconnect to route any piece of data from one location on the array to another. By default, this path will have no registers along it, and its length will be determined by how many tiles it passes through, or "hops" on the interconnect.

On real hardware, all hops on the interconnect will not have the same delay. A memory tile has a much larger footprint than a PE tile, so traveling from one side of a memory tile to the other takes longer than the same path through a PE tile. We also find that the lengths of the wires going in one direction through a tile are not the same as those going in another direction. Finally, we also have to consider the possibility that the clock does not arrive at the same time at each tile due to clock skew.

Canal generates the start and end points of these paths in the corresponding Verilog representation. Cascade then estimates the delays of these paths using a commercial STA tool (like PrimeTime), running on the tile's post-layout netlist annotated with the parasitic delays of the wires in the circuit.

This methodology enables the timing model to adapt to hardware changes. In an agile hardware development flow where designers iterate on a design, this adaptation is critical.

B. Application Static Timing Analysis Tool for CGRAs

We feed the delays described in the previous section into a STA tool we designed for CGRA applications. The input to this tool is the application dataflow graph representation after the PnR stage of the compiler. As described in Section II-D, we use the STA algorithm to calculate the arrival time of each piece of data at every node in the dataflow graph. As shown in Fig. 1, the maximum of these arrival times determines the resulting critical path and the maximum frequency of the application. We can use the STA tool to make pipelining decisions. For example, for an application we are attempting to pipeline the STA tool will let us know if we decreased the critical path delay or increased it. We evaluate the accuracy of this STA tool in Section VI-A.

C. Pipelining Techniques in Cascade

We integrate a large set of existing and our proposed pipelining and register usage optimization techniques from Sections III and IV into an end-to-end flow, that works both for dense and sparse applications. We describe below the techniques we have not already covered. The complete flow is summarized in Fig. 6.

1) Compute Pipelining: During the compute mapping stage of the application compiler, we translate a DAG of primitive operations into a DAG of PEs. Note that our full application graphs are not DAGs, but can be decomposed into kernels which are DAGs. In our target CGRA, each PE has configurable registers connected to each input. The compute pipelining stage in Cascade turns on every available PE input register and then performs branch delay matching to ensure that the compute kernels maintain their functionality, as shown in Fig. 4. The delay through a PE depends on the operation being performed, but we measured our target CGRA PE to have a maximum delay of 0.8ns. A typical compute kernel for an image processing application has around 10 PEs along one path from the input of the kernel to the output, so enabling PE pipeline registers dramatically reduces the application critical path.

We enable every PE input register because our goal is to reach the maximum possible frequency of our array (the CGRA used for our evaluation has a maximum frequency of 1 GHz). If the target frequency is lower, a subset of PE input registers can be enabled to tradeoff some frequency gains for energy efficiency.

Compute pipelining is an idea that has been used successfully in FPGA and ASIC compilers, and we evaluate it for CGRAs in Section VI. To further improve critical path lengths, we must use techniques focused on minimizing the actual wire delays present in the applications post mapping and PnR.

2) Broadcast Signal Pipelining: In our target CGRA, the delay through a PE tile is a maximum of 0.8 ns, while the delay through one switch box is about 0.14 ns. If every PE is pipelined during compute pipelining, then the path delays

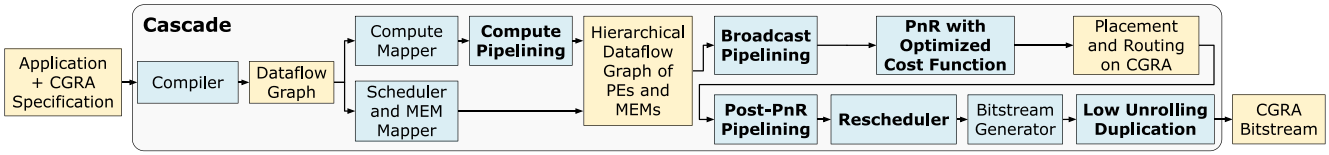


Fig. 6. CGRA application compiler that takes an application and a specification of the CGRA and produces a CGRA bitstream. Intermediate representations are dataflow graphs. Bolded text indicates stages of the compiler that were added or modified in this work for pipelining.

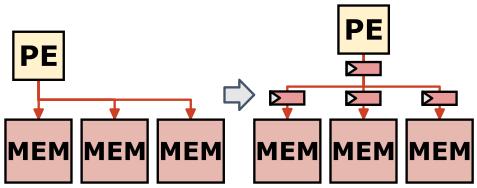


Fig. 7. Broadcast signal pipelining transformation where a broadcast signal with one source and multiple destinations has pipelining registers inserted to transform long combinational paths into many short pipelined paths.

in an application will become dominant after five hops on the interconnect.

Broadcast paths that have one source and many destinations end up being routed inefficiently on the CGRA, and typically have many more than five hops on the interconnect. The primary objective of the place and route algorithm is to minimize wirelength. The minimum wirelength implementation of a broadcast path typically "snakes" from one source to each destination, with the branching points of the path very close to the tiles. If the number of destination nodes is large, then this broadcast path will be long and become the application's critical path.

To solve this problem, we can specifically pipeline broadcast paths and use a tree structure to ensure that the maximum wirelength is minimized, as shown in Fig. 7. There is a tradeoff between number of registers added and critical path length, so the parameters of this transformation pass (tree levels, maximum number of pipeline registers, etc.) can be adjusted.

3) Placement Algorithm Cost Function Optimization: Our compiler places and routes the application onto the CGRA using a simulated annealing-based placement algorithm and an iteration-based routing algorithm [12]. We perform placement in two stages: global placement and detailed placement. Global placement uses an analytical algorithm that leverages the standard conjugate gradient method to minimize the total wirelength of the application. This wirelength is calculated by summing the wirelength of each net, where the wirelength is estimated using the half-perimeter wire length (HPWL). After global placement, we perform detailed placement based on simulated annealing.

The cost function for detailed placement (see (2)) is calculated by summing the HPWL cost for each net. To improve the critical paths of applications that are placed and routed using this technique, we add another hyperparameter α that penalizes longer routes, similar to the criticality exponent introduced in [16]. Typical placement algorithms do not use this parameter, and therefore minimize total wirelength. By introducing α, we can adjust how much long routes are penalized. A higher value will mean that long routes cost much more than shorter routes. The value of α is determined during the execution of the place and route algorithm. Since a single place and route run completes in seconds, we run it multiple times with different values for α. For the experiments in this article, every value of α from 1 to 30 is tried to determine the highest frequency result

$$\text{Cost}_{\text{net}} = (\text{HPWL}_{\text{net}})^{\alpha}. \tag{2}$$

After placement, we route using the same approach as [12].

4) Low Unrolling Duplication: Image processing and machine learning applications are typically unrolled (parallelized) on hardware accelerators to improve performance. We found that compiling applications with no unrolling and performing place and route on a smaller portion of the CGRA often leads to much shorter critical paths. The configuration of the tiles and interconnect is then duplicated across the array, effectively "unrolling" the application by the exact same amount as before. This optimization allows the PnR tool to solve a much smaller problem while maintaining all of the benefits of parallelization.

5) Hardening Frequent Expensive Broadcast Paths: With broadcast signal pipelining, expensive broadcast routes causing long critical paths can be shortened. However, for broadcast paths that have many destinations, the number of registers that need to be placed on the CGRA is very large. For example, the flush signal is a broadcast path with one source and potentially hundreds of destinations, depending on the size of the application. Pipelining this broadcast path using the technique described in Section V-C2 is not feasible for these applications, so if there is flexibility in designing the hardware, we recommend hardening such signals. Instead of routing this signal in the configurable interconnect, we directly connect it to every tile outside of the interconnect. This signal is routed from the top of the array to the bottom, running down each column. We can ensure that this path is not too long by placing pipeline registers at certain rows in the array hardware as shown in Fig. 8.

D. Extensions for Pipelining Sparse Applications

So far we have focused our discussion on statically scheduled applications. However, not all applications can be completely statically scheduled. In particular, sparse applications may have data-dependent memory accesses. When targeting a sparse CGRA application, we need to include several additional considerations.

Sparse applications typically use a ready-valid interface between all stages of an application. If a piece of data is routed

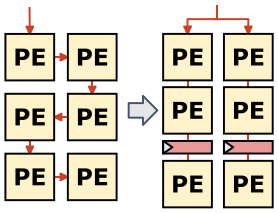


Fig. 8. Hardened flush transformation pass that transforms a flush signal that has to be routed in the configurable interconnect into a hardened signal that does not exist on the interconnect. In this example, the hardened flush signal is pipelined between the second and third rows of the array.

from *Tile A* to *Tile B*, a valid signal will be routed in the exact same way, going through the same switch boxes and connection boxes. A ready signal will be routed in the same way but in the opposite direction. Each of these wires travel the same distance as the original data signal.

If we identify a route in a sparse application as the critical path, we cannot increase application frequency by adding a register to the data signal alone, we need to register the ready and the valid signals as well. However, naïvely adding pipelining registers to all three signals would break the ready-valid interface, which relies on single-cycle communication of when a component is ready to receive another piece of data. Therefore, all the techniques from Section V-C, except placement cost function optimization and low unrolling duplication, require a small change. The change entails inserting FIFOs instead of registers to break the long data, ready, and valid paths together. These FIFOs include logic for handling the ready and valid signals correctly, and allow us to apply our pipelining flow to sparse applications. The insertion of pipelining FIFOs requires that FIFOs exist within the target CGRA interconnect in some form. In Section VI-E, we compare how well this technique works for two different types of interconnect FIFOs.

VI. Results

We first evaluate the prediction accuracy of the application STA model using standard delay format (SDF)-annotated gate-level simulations. Then, we assess the impact of each pipelining technique in Cascade on maximum frequency, runtime, and EDP of several dense and sparse applications. Finally, we compare Cascade against the state-of-the-art CGRA compilers and pipelining approaches. We evaluate these techniques within the application compiler shown in Fig. 6. The CGRA architecture that we use is a 32×16 array with 384 PE tiles and 128 MEM tiles. We perform physical design of the CGRA in GlobalFoundries 12 nm technology. Fig. 9 shows the final layout of the CGRA, along with a global buffer for storing input, output, and intermediate data from the applications.

A. Evaluating the Application STA Model

We evaluate the application STA model to ensure that the critical path derived from the model matches the critical path derived from SDF-annotated gate-level simulation. We use a SDF-annotated gate-level simulation of the CGRA to search for the fastest clock period of applications with different pipelining techniques. The simulation is performed on a post-layout version of the CGRA netlist and includes both gate delays and wire delays. The search granularity is 0.05 ns.

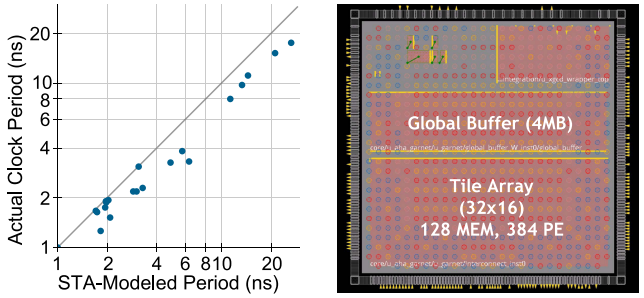


Fig. 9. Left: Evaluation of the STA critical path model. Each dot represents an application running at the frequency indicated by the vertical axis. The dot's horizontal value is the STA-modeled clock period. The gray line represents a perfect match. Right: CGRA chip layout.

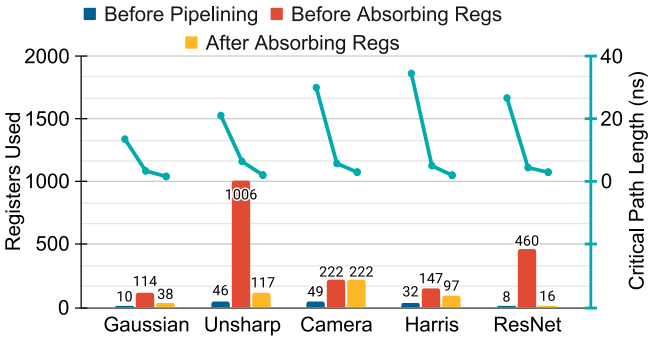


Fig. 10. Register resource utilization before pipelining (blue bar), after pipelining but before absorbing registers (red bar), and after absorbing registers (yellow bar). Critical path delay is also shown, which improves with our optimization.

As shown in Fig. 9, the STA-modeled clock period is generally higher, but well correlated, with that from SDF-annotated gate-level simulation. This means that our STA model is pessimistic and provides a lower bound for the actual maximum frequency (which is what we want). This behavior is expected as we collected the worst-case path lengths when constructing this STA model. At clock frequencies above 500 MHz, which is the range we care the most about, the average error is 13%. That is, the STA model has good accuracy for predicting the critical path length for applications running at high frequencies.

B. Evaluating Register Resource Usage Optimization

Next, we analyze the resource utilization impact of the technique for absorbing interconnect registers from Section IV. We evaluate the impact of this optimization before the impact of the pipelining techniques themselves, as it is necessary to ensure that the applications do not exceed the resources on the reconfigurable interconnect. For these experiments, we report the register resource utilization of five image processing and machine learning applications in Fig. 10. The amount of register resource reduction from this technique varies by application from 0% to 97%. The structure of the compute kernels

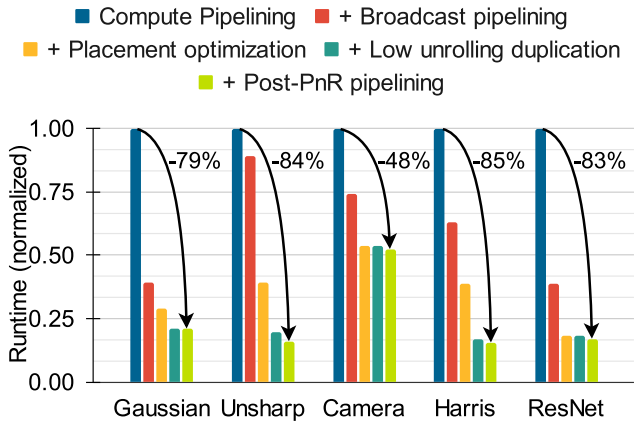


Fig. 11. Incremental effect of each software pipelining technique on the runtime of dense applications.

TABLE I
FREQUENCY, RUNTIME, POWER, AND COMPILER RUNTIME COMPARISON BETWEEN UNPIPELINED AND PIPELINED VERSIONS OF FIVE DENSE APPLICATIONS

Dense Application		Frequency (MHz)	Runtime (ms/frame)	Power (mW)	Compiler Runtime (s)
Unpipelined	Gaussian	103	22.6	156	1072
	Unsharp	66	21.4	139	580
	Camera	47	28.3	318	716
	Harris	30	70.6	85	555
	ResNet	57	31.7	119	543
Pipelined	Gaussian	610	3.66	841	560
	Unsharp	606	1.75	1072	970
	Camera	457	2.96	678	2549
	Harris	571	1.90	614	805
	ResNet	457	3.96	304	1122

within each application determines the number of registers that can be absorbed. Unsharp and ResNet convolution layer benefit the most from this technique as they have a lot of chained multiply-add operations. Without register absorption, these chaining structures require many pipelining registers to balance.

Note that many absorbed registers require no memory resources, as they can be absorbed simply by changing the configuration of the address generators within the memory tiles. Register files that are used to replace chains of pipelining registers cannot be used for anything else, so those memory resources are consumed during this process. However, the memory resource cost of this technique is negligible.

C. Evaluating Cascade on Dense Applications

We analyze the incremental impact of software pipelining techniques from Sections III and V-C on the runtime of five dense applications from image processing and machine learning domains, which are also benchmarks in a previous CGRA work [12]. These software pipelining techniques are implemented as compiler passes. This is shown in Fig. 11 and Table I. The results in Fig. 11 are derived from our STA model, while the results in Table I are verified with SDF-annotated gate-level simulation. For image processing

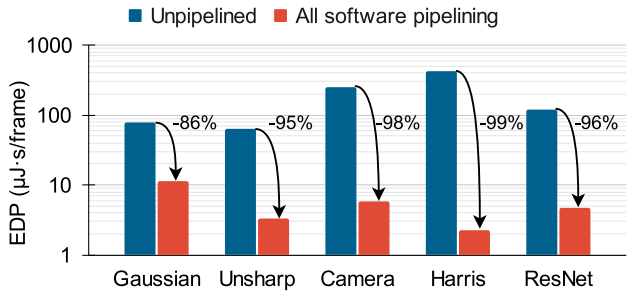


Fig. 12. EDP of unpipelined applications versus applications with all software pipelining. The EDP decreases by 95% on average.

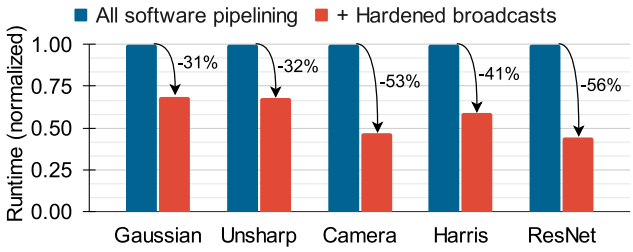


Fig. 13. Hardening broadcast signals that reach many tiles reduces runtime by 31%–56%.

applications, the frame size is 6400 × 4800 for Gaussian, 1536 × 2560 for Unsharp, 2560 × 1920 for Camera, and 1530 × 2554 for Harris. For machine learning, ResNet refers to a single conv5_x layer of ResNet-18. In these experiments, we have applied the flush signal hardening from Section V-C5. Furthermore, Fig. 12 shows the impact of our pipelining flow on EDP. As shown in Table I, the software pipelining techniques achieve an 84%–97% decrease in runtime versus unpipelined implementations. Compute pipelining alone results in a 35%–81% reduction in runtime compared to the un-pipelined applications, while the techniques applied during and after PnR result in an additional 48%–85% reduction in runtime. The pipelining techniques result in an EDP decrease of 86%–99%.

Overall, the compile time generally increased using these pipelining techniques. The exploration needed to find the best placement optimization hyperparameter caused the largest increase in compile time. The low unrolling duplication technique decreased compile time, which in the Gaussian application contributed to a decrease in overall compile time.

Fig. 13 shows the impact of hardening broadcast signals (the flush signal in the case of our CGRA) (Section V-C5). In this experiment, all of the software only pipelining techniques are applied to isolate the impact of the hardware change. As shown in Fig. 13, the runtime is reduced by 31%–56%.

D. Evaluating Cascade on Sparse Applications

We evaluate the effect of our pipelining techniques from Section V-D on four sparse workloads from [7]. These sparse workloads are small benchmarks with a sparsity of 70%. Note that the sparse applications use FIFOs at the input of every compute unit, so compute pipelining is applied by default by the compiler and cannot be turned off. Additionally,

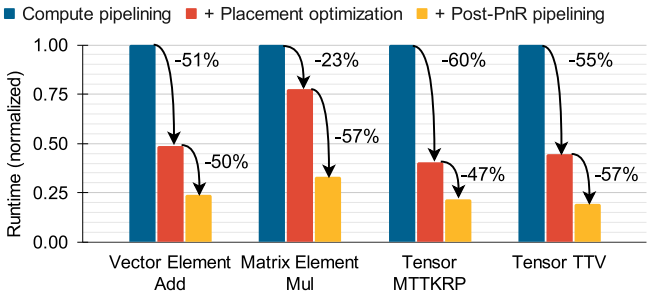


Fig. 14. Incremental application of our software pipelining techniques to sparse applications.

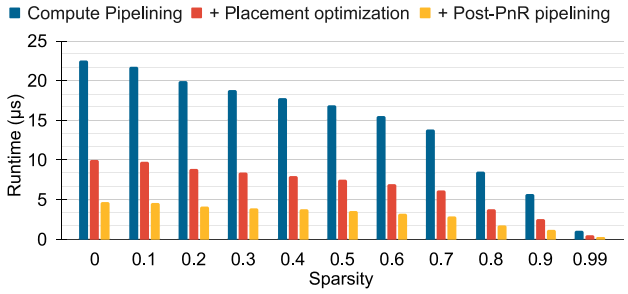


Fig. 15. Runtime impact of the software pipelining techniques on Tensor TTV for different levels of input sparsity.

TABLE II
FREQUENCY, RUNTIME, AND POWER COMPARISON BETWEEN COMPUTE PIPELINED AND FULLY PIPELINED VERSIONS OF FOUR SPARSE APPLICATIONS

	Sparse Application	Frequency (MHz)	Runtime (µs)	Power (mW)
Compute Pipelining	Vector Elementwise Add	305	0.77	187
	Matrix Elementwise Mul	435	0.41	246
	Tensor MTTKRP	300	34.9	194
	Tensor TTV	260	10.0	170
All Software Pipelining	Vector Elementwise Add	870	0.28	410
	Matrix Elementwise Mul	909	0.20	432
	Tensor MTTKRP	625	14.3	333
	Tensor TTV	833	3.87	394

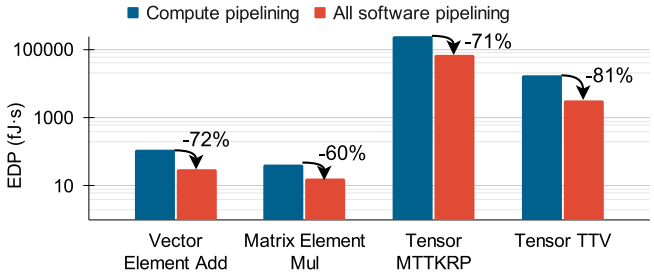


Fig. 16. EDP comparison between compute pipelining only and all software pipelining techniques applied to sparse applications (vertical axis has a log scale).

broadcast pipelining and low unrolling duplication had no effect on the frequency so only placement optimization and post-PnR pipelining are evaluated. Fig. 14 shows the effect of incrementally applying each technique.

Additionally, we show the runtime impact of the software pipelining techniques on the Tensor TTV application for different levels of input sparsity in Fig. 15. As sparsity increases from 0% to 99%, runtime decreases, but the relative runtime impact of the pipelining techniques does not depend on the input data sparsity. The software pipelining techniques have a significant relative impact on the runtime and are necessary to achieve high performance.

Table II shows the final numbers for maximum frequency, runtime, and power consumption for sparse applications, which are verified with SDF-annotated gate-level simulation. As shown in Table II, the runtime of sparse applications decreases by 52%–71% compared to versions with only compute pipelining, and as shown in Fig. 16, the EDP reduces by 60%–81% with our techniques.

E. Comparison Against State of the Art

Finally, we compare the performance and energy efficiency of Cascade versus the state-of-the-art pipelining techniques for CGRA architectures. Cascade targets static CGRA interconnects that allow for any tile to any tile connections with configurable registers. We compare against three alternative architectures: 1) ADRES [18] with its associated compiler DRESC [17], which uses exhaustive pipelining (i.e., it adds a pipelining register after every interconnect hop) and only allows for connections between neighboring tiles and connections to tiles in the same row and column, 2) HyCUBE [9] which allows for single-cycle multihop connections, and 3) Plasticine [21] with its associated compiler Sara [24], which uses exhaustive pipelining and interconnect FIFOs. For a fair comparison, we have implemented these CGRA interconnects within the same CGRA hardware generation platform, meaning they are in the same technology, use the same tile architectures, and are running the same applications.

The architecture and pipelining approach of ADRES and DRESC are not flexible enough for our application benchmarks. ADRES only has neighbor to neighbor and row/column connections and requires that each connection take one cycle. This restrictive interconnect topology is not appropriate for the large benchmarks and arrays that we are targeting. Applications that have simple connectivity, with routes that only have one source and one destination, may be appropriate for this type of interconnect. However, our applications, which have complex connectivity with routes that have many destinations, cannot be placed and routed onto this type of interconnect. The complexity of the tiles themselves also influences which type of interconnect to use. Our PE tiles have six inputs and two outputs, and our memory tiles have six inputs and six outputs. These complex tiles enable efficient computation, but require more flexibility in the interconnect than ADRES provides. Additionally, in an exhaustively pipelined interconnect, the number of cycles a piece of data takes to get from its source to its destination depends on the route that it takes. Matching the branch delays (described in Section V-C1) is much more difficult, and becomes impossible for the complexity of our benchmarks.

Next, we compare Cascade against a HyCUBE-like CGRA and compiler. HyCUBE's pipelining approach is very flexible,

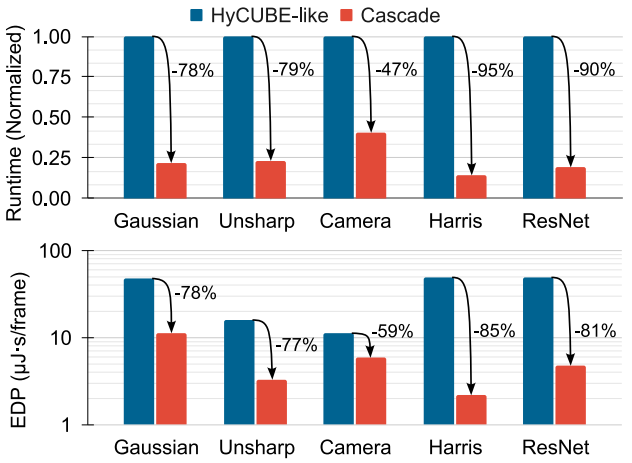


Fig. 17. Runtime and EDP comparison with a HyCUBE-like array and compiler.

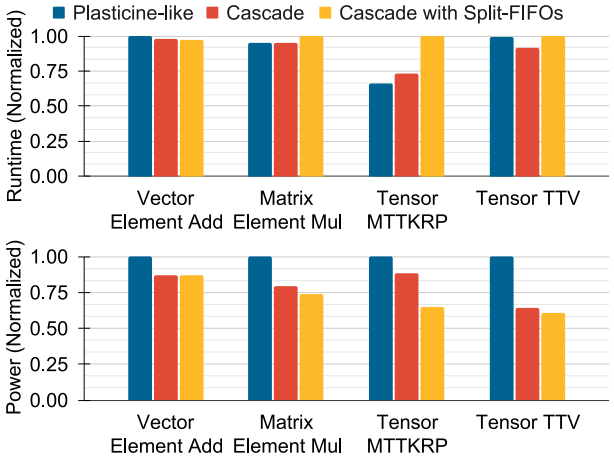


Fig. 18. Runtime and power comparison with a plasticine-like array and compiler.

they allow for any tile to any tile connections, and register data going into every tile. Routes between tiles take 1 cycle. While the approach taken by HyCUBE is quite flexible, as it can place and route applications with a wide range of connection patterns, it was designed with a smaller array in mind. As shown in Fig. 17, when HyCUBE's approach is scaled up to a 32×16 CGRA, the critical paths become long, and compared to HyCUBE, Cascade achieves 47%–95% lower runtime and 59%–85% better EDP. In fact, Cascade would be well suited for compiling to HyCUBE-like CGRAs.

Finally, we compare against a Plasticine-like array and compiler. To isolate the effect of the pipelining approach, we do these experiments using the sparse benchmarks, which require the use of the ready-valid interface present in Plasticine. In this experiment, we present three different pipelining approaches: a Plasticine-like approach that is exhaustively pipelined using FIFOs at every hop in the interconnect, a Cascade approach applied to the exact same interconnect, and a Cascade approach using optimized FIFOs. These optimized FIFOs are distributed, a size two FIFO is constructed using two adjacent switchbox registers. Cascade enables the effective use of this type of optimization, which relies on intelligently analyzing routing delays and enabling registers post-PnR.

The runtime and power comparison is shown in Fig. 18. We can see that the exhaustive pipelining approach taken by Plasticine achieves low runtime, but has higher power consumption than Cascade. The Plasticine-like array is within 10% of the runtime of Cascade approach, these differences can be attributed to some randomness in the PnR results. The power consumption of Cascade is 21%–36% less than the Plasticine approach, demonstrating that Cascade can achieve similar level of performance at a lower power cost. The FIFOs needed at the input of every tile and within every switchbox have a high power cost. The Cascade approach will selectively enable only the FIFOs that are on the critical paths, thus saving power. Finally, the optimized split-FIFO interconnect, pipelined using Cascade, shows further power benefits at a runtime cost. The runtime cost is usually small, with the exception of Tensor MTTKRP, but the power savings are larger than just using Cascade by itself, with a 13%–39% reduction in power over the Plasticine approach.

VII. Related Work

A. FPGA and ASIC Pipelining Tools

Application pipelining when targeting FPGAs and ASICs is a well studied problem; however, many of these techniques cannot be directly applied to CGRA applications or do not achieve the same level of improvement for CGRAs. Register retiming [2], [13] is a common pipelining technique where pipelining register stages are added to a design and retimed into optimal positions.

Post-placement retiming and pipelining has been applied in the past to both ASICs and FPGAs. Tien et al. [22] introduced post-placement retiming and register insertion for ASICs. The critical paths are identified post-placement and iteratively pipelined to enable higher maximum frequency. Similarly, Intel's Stratix 10 [14] architecture includes pipelining registers in the interconnect and does post-PnR performance tuning including retiming and pipelining.

This class of techniques can be applied to CGRA applications, although the limited register resources and long delays through PE tiles mean that the algorithms used for ASICs and FPGAs do not directly apply to CGRA applications.

B. Types of CGRA Interconnects

CGRA interconnects fall into two main categories: static and dynamic. Statically configured interconnects have routes that are reserved during compile time and do not change during the execution of the application. In contrast, a dynamically configured interconnect, or network-on-chip, can change/time-multiplex the connections on the array during application execution. In this article, we only target static interconnects. Note that a CGRA with a static interconnect is not necessarily fully statically scheduled — while the routes in the application may not change during application execution, the data flowing on those routes may not be statically scheduled [21].

There are several different types of static CGRA interconnects, and for this discussion, we break this design

TABLE III
CATEGORIZATION OF CGRAS AND THEIR COMPILERS BASED ON CONNECTION FLEXIBILITY AND PIPELINING FLEXIBILITY OF THE INTERCONNECT. ALSO SHOWN ARE THE CGRA SIZES USED WHEN EVALUATING THE COMPILERS AND THE REPORTED APPLICATION FREQUENCIES

CGRA Compiler	Connection Flexibility	Pipelining Flexibility	CGRA Size (# Tiles)	Application Frequency (MHz)
EPImap [6]	Neighbors	Exhaustive Pipelining	16	Not reported
Nowatzki [20]	Neighbors	Exhaustive Pipelining	16	Not reported
Zhao [25]	Neighbors	Exhaustive Pipelining	64	Not reported
DRESC [17]	Neighbors and row/column	Exhaustive Pipelining	64	100
Sara [24]	Any tile to any tile	Exhaustive Pipelining with FIFOs	420	1000
HyCUBE [9]	Any tile to any tile	Configurable Registers	16	704
Cascade (This work)	Any tile to any tile	Configurable Registers	512	450 - 1000

space along two main axes: connection flexibility and pipelining flexibility. In terms of connection flexibility there are three main categories of interconnect topologies: neighbor to neighbor connections (low flexibility), row and column connections (medium flexibility), and any tile to any tile connections (high flexibility). In terms of pipelining flexibility, there are three main categories: 1) no pipelining registers, 2) exhaustive pipelining registers, and 3) configurable pipelining registers. Table III classifies several existing CGRAs and their compilers along these axes.

C. Comparison With Prior CGRA Compilers

A comparison between the different classes of CGRA compilers is provided in Fig. 19 and Table III. Sara [24], the compiler for the Plasticine architecture [21], assumes an exhaustively pipelined interconnect. To accommodate this exhaustive pipelining, Plasticine has FIFOs that handle network delays. This type of architecture and exhaustive pipelining results in high power consumption; [9] reports that exhaustive pipelining has a 28% higher power cost than the alternative without exhaustive pipelining. Sara targets CGRA interconnects that have any tile to any tile connections so the compiler can compile large applications to large arrays.

EPImap [6], DRESC [17], and the compilers introduced in [20] and [25] target CGRAs that have neighbor to neighbor connections and row and column connections, and are exhaustively pipelined. Additionally, these CGRA architectures allow for per-cycle reconfiguration of the tiles in the array. These compilers must ensure that the paths in the array are balanced in length so that the data arrives at each tile at the correct cycle. These types of interconnect architectures are less flexible than those that allow for any tile to any tile connections.

HyCUBE [9] is a CGRA architecture that supports single-cycle multihop connections between tiles. It supports any-tile to any-tile connections and uses configurable registers to allow for single-cycle multihop connections. While this type of interconnect is more flexible, the critical path of the application running on the accelerator depends on the configuration of the interconnect. Their work acknowledges that this architecture may have longer critical paths than other types of interconnects, so they limit the number of hops a piece of data can make on the interconnect in one cycle. The HyCUBE array has only four rows and four columns, so the longest possible critical path is very short.

Cascade targets CGRAs with high connection flexibility (any tile can send data to any other tile through the interconnect) and has high pipelining flexibility through its ability to selectively turn on registers only where they are required (i.e., the most flexible option in Table III). Furthermore, we target CGRA architectures that are much larger than HyCUBE (e.g., containing 512 tiles versus 16 tiles), so limiting the number of hops on the interconnect is not a solution that we can employ.

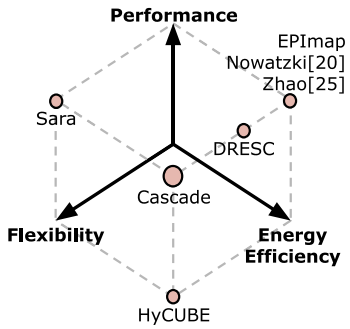


Fig. 19. Performance, flexibility, and energy efficiency comparison between the various CGRA compilers.

VIII. CONCLUSION

Cascade is an open-source end-to-end CGRA compiler that achieves high performance and fast compilation through the use of a decoupled FPGA-style compilation flow and new pipelining techniques. It includes a novel post PnR application pipelining technique for CGRAs that accounts for interconnect hop delays during pipelining but in a unique way that avoids cyclic scheduling and PnR. To work with the limited number of registers present on CGRAs (compared to FPGAs or ASICs), Cascade performs register resource optimization that leverages the scheduling logic in CGRA memory tiles to minimize the number of register resources used during pipelining. We put these techniques together with an automated CGRA timing model generator, an application timing analysis tool, and a large set of pipelining techniques to create an end-to-end compiler. Cascade achieves 8–34× lower critical path delay and 7–190× lower EDP across a variety of dense image processing and machine learning workloads, and 3–5.2× lower critical path delay and 2.5–5.2× lower EDP on sparse workloads, compared to a compiler without pipelining. While Cascade is a standalone compiler, the pipelining techniques can be integrated into other CGRA compilers as well, enabling the creation of high performance compilation infrastructure for CGRAs and encouraging research into CGRAs as promising acceleration platforms.

REFERENCES

[1] "Cascade compiler." Accessed: Jan. 30, 2024. [Online]. Available: https://github.com/StanfordAHA/aha

[2] P.-Y. Calland, A. Mignotte, O. Peyran, Y. Robert, and F. Vivien, "Retiming DAGs [direct acyclic graph]," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 17, no. 12, pp. 1319–1325, Dec. 1998. [Online]. Available: https://doi.org/10.1109/43.736571

[3] A. Carsello et al., "Amber: A 367 GOPS, 538 GOPS/W 16nm SoC with a coarse-grained reconfigurable array for flexible acceleration of dense linear algebra," in *Proc. IEEE Symp. VLSI Technol. Circuits*, 2022, pp. 70–71. [Online]. Available: https://doi.org/10.1109/VLSITechnologyandCir46769.2022.9830509

[4] (Siemens Autom. Co., Munich, Germany). *High-Level Synthesis and Verification: Catapult*. Accessed: Aug. 8, 2023. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/

[5] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[6] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "EPIMap: Using epimorphism to map applications on CGRAs," in *Proc. 49th Annu. Design Autom. Conf.*, 2012, pp. 1284–1291. [Online]. Available: https://doi.org/10.1145/2228360.2228600

[7] O. Hsu et al., "The sparse abstract machine," in *Proc. 28th ACM Int. Conf. Archit. Support Program. Languages Oper. Syst.*, 2023, pp. 710–726. [Online]. Available: https://doi.org/10.1145/3582016.3582051

[8] D. Huff, S. Dai, and P. Hanrahan, "Clockwork: Resource-efficient static scheduling for multi-rate image processing applications on FPGAs," in *Proc. IEEE 29th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2021, pp. 186–194.

[9] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "HyCUBE: A CGRA with reconfigurable single-cycle multi-hop interconnect," in *Proc. 54th Annu. Design Autom. Conf.*, 2017, pp. 1–6. [Online]. Available: https://doi.org/10.1145/3061639.3062262

[10] T. I. Kirkpatrick and N. R. Clark, "Pert as an aid to logic design," *IBM J. Res. Dev.*, vol. 10, no. 2, pp. 135–141, Mar. 1966. [Online]. Available: https://doi.org/10.1147/rd.102.0135

[11] X. Kong et al., "MapZero: Mapping for coarse-grained reconfigurable architectures with reinforcement learning and monte-carlo tree search," in *Proc. 50th Annu. Int. Symp. Comput. Archit.*, 2023, pp. 1–14. [Online]. Available: https://doi.org/10.1145/3579371.3589081

[12] K. Koul et al., "AHA: An agile approach to the design of coarse-grained reconfigurable accelerators and compilers," *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 2, pp. 1–34, Jan. 2023. [Online]. Available: https://doi.org/10.1145/3534933

[13] C. E. Leiserson, F. M. Rose, and J. B. Saxe, "Optimizing synchronous circuitry by retiming (preliminary version)," in *Proc. 3rd Caltech Conf. Very Large Scale Integr.*, 1983, pp. 87–116. [Online]. Available: https://doi.org/10.1007/978-3-642-95432-0_7

[14] D. Lewis et al., "The Stratix 10 highly pipelined FPGA architecture," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2016, pp. 159–168. [Online]. Available: https://doi.org/10.1145/2847263.2847267

[15] Q. Liu et al., "Unified buffer: Compiling image processing and machine learning applications to push-memory accelerators," *ACM Trans. Archit. Code Optim.*, vol. 20, no. 2, pp. 1–26, Mar. 2023. [Online]. Available: https://doi.org/10.1145/3572908

[16] A. Marquardt, V. Betz, and J. Rose, "Timing-driven placement for FPGAs," in *Proc. ACM/SIGDA 8th Int. Symp. Field Program. Gate Arrays*, 2000, pp. 203–213. [Online]. Available: https://doi.org/10.1145/329166.329208

[17] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "DRESC: A retargetable compiler for coarse-grained reconfigurable architectures," in *Proc. IEEE Int. Conf. Field-Program. Technol. (FPT)*, 2002, pp. 166–173. [Online]. Available: https://doi.org/10.1109/FPT.2002.1188678

[18] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Proc. 13th Int. Conf. Field Program. Logic Appl. (FPL)*, 2003, pp. 61–70. [Online]. Available: https://doi.org/10.1007/978-3-540-45234-8_7

[19] J. Melchert, K. Zhang, Y. Mei, M. Horowitz, C. Torng, and P. Raina, "Canal: A flexible interconnect generator for coarse-grained reconfigurable arrays," *IEEE Comput. Archit. Lett.*, vol. 22, no. 1, pp. 45–48, Jan.–Jun. 2023. [Online]. Available: https://doi.org/10.1109/LCA.2023.3268126

[20] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili, "A general constraint-centric scheduling framework for spatial architectures," *SIGPLAN Not.*, vol. 48, no. 6, pp. 495–506, Jun. 2013. [Online]. Available: https://doi.org/10.1145/2499370.2462163

[21] R. Prabhakar et al., "Plasticine: A reconfigurable architecture for parallel patterns," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 389–402. [Online]. Available: https://doi.org/10.1145/3079856.3080256

[22] T. C. Tien, H. P. Su, and Y. W. Tsay, "Integrating logic retiming and register placement," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design*, 1998, pp. 136–139. [Online]. Available: https://doi.org/10.1145/288548.288591

[23] *Design Suite—VivadoHLS: Vivado*. Accessed: Aug. 8, 2023. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

[24] Y. Zhang, N. Zhang, T. Zhao, M. Vilim, M. Shahbaz, and K. Olukotun, "SARA: Scaling a reconfigurable dataflow accelerator," in *Proc. 48th Annu. Int. Symp. Comput. Archit.*, 2021, pp. 1041–1054. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00085

[25] Z. Zhao et al., "Towards higher performance and robust compilation for CGRA modulo scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 9, pp. 2201–2219, Sep. 2020. [Online]. Available: https://doi.org/10.1109/TPDS.2020.2989149

Jackson Melchert received the B.S. degree in electrical and computer engineering and computer science from the University of Wisconsin–Madison, Madison, WI, USA, in 2019. He is currently pursuing the Ph.D. degree in electrical engineering with Stanford University, Stanford, CA, USA, supervised by Prof. Priyanka Raina.

He is broadly interested in optimizing configurable hardware to approach the performance and efficiency of application-specific accelerators.

Yuchen Mei received the B.S. degree in electronic information science and technology from Nanjing University, Nanjing, China, in 2021, and the M.S. degree in electrical engineering from Stanford University, Stanford, CA, USA, in 2023, where he is currently pursuing the Ph.D. degree in electrical engineering advised by Prof. Priyanka Raina.

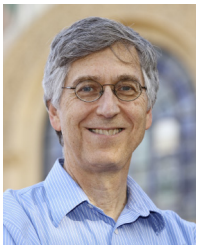
His research focuses on application mapping and optimization for domain-specific accelerators, with a particular interest in auto-scheduling for compute-intensive applications, such as image processing and deep learning.

Kalhan Koul (Graduate Student Member, IEEE) received the B.S. degree (Hons.) in electrical engineering and the B.A. degree in Plan II Honors from the College of Liberal Arts, University of Texas, Austin, TX, USA, in 2018. He is currently pursuing the Ph.D. degree in electrical engineering supervised by Prof. Priyanka Raina.

He has interned at Apple Inc., Cupertino, CA, USA; Micron Inc., San Jose, CA; and Silicon Labs, Austin, TX, USA. His current research focuses on domain specific hardware architectures and design methodology. Specifically, he is working on automatically mapping applications, ranging from machine learning to image processing, onto CGRAs and reconfigurable logic devices.

Qiaoyi Liu received the B.S. degree in electronic engineering from Tsinghua University, Beijing, China, in 2017, and the M.S. degree in electrical engineering from Stanford University, Stanford, CA, USA, in 2020, and the Ph.D. degree in electrical engineering from Stanford University advised by Prof. Mark Horowitz.

His research focuses on computer architecture and compilation for domain-specific accelerators, with a particular interest in optimizing schedules and memory mappings for compute-intensive tensor applications, such as image processing and deep learning.



Mark Horowitz (Life Fellow, IEEE) received the B.S. and M.S. degrees in electrical engineering from the Massachusetts Institute of Technology, Cambridge, MA, USA, in 1978, and the Ph.D. degree from Stanford University, Stanford, CA, USA, in 1984.

He is the Yahoo! Founders Professor with Stanford University and the Chair of the Electrical Engineering Department. He has worked on many processor designs, from early RISC chips to distributed shared memory multiprocessors, and in 1990 he took leave from Stanford to help start Rambus Inc, Sunnyvale, CA, USA, a company designing high-bandwidth memory interface technology. His work with Rambus and Stanford drove high-speed link designs for many decades. In the 2000s, he started a collaboration with Marc Levoy in computational photography which led to light-field photography and microscopy. His current research includes updating both analog and digital design methods, agile hardware design, and applying engineering to biology. He remains interested in learning new things and building interdisciplinary teams. He is a Fellow of ACM and a Member of the National Academy of Engineering and the American Academy of Arts and Science.

Priyanka Raina received the B.Tech. degree in electrical engineering from the Indian Institute of Technology at Delhi, Hauz Khas, India, in 2011, and the M.S. and Ph.D. degrees in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, MA, USA, in 2013 and 2018, respectively.

She was a Visiting Research Scientist with NVIDIA Corporation, Santa Clara, CA, USA, in 2018. She is currently an Assistant Professor of Electrical Engineering with Stanford University, Stanford, CA, where she works on domain-specific hardware architectures and agile hardware–software codesign methodology.

Dr. Raina was a co-recipient of the Best Demo Paper Award at VLSI 2022, the Best Student Paper Award at VLSI 2021, the IEEE Journal of Solid-State Circuits Best Paper Award in 2020, the Best Paper Award at MICRO 2019, and the Best Young Scientist Paper Award at ESSCIRC 2016. She has won the Sloan Research Fellowship in 2024, the National Science Foundation CAREER Award in 2023, the Intel Rising Star Faculty Award in 2021, and the Hellman Faculty Scholar Award in 2019. She was the Program Chair of the IEEE Hot Chips in 2020. She serves as an Associate Editor for the IEEE JOURNAL OF SOLID-STATE CIRCUITS and IEEE SOLID-STATE CIRCUITS LETTERS. She is a 2018 Terman Faculty Fellow.