Mocarabe: High-Performance Time-Multiplexed Overlays for FPGAs

Frederick Tombs
fgjtombs@uwaterloo.ca
University of Waterloo
Ontario, Canada

Alireza Mellat
amellat@uwaterloo.ca
University of Waterloo
Ontario, Canada

Nachiket Kapre
nachiket@uwaterloo.ca
University of Waterloo
Ontario, Canada

***Abstract*—Coarse-grained reconfigurable array (CGRA) overlays can improve dataflow kernel throughput by an order of magnitude over Vivado HLS. This is possible with a combination of carefully floorplanned high-frequency (645–768 MHz) design and a scalable, communication-aware compiler. Our 2D torus CGRA architecture supports versatile processing element (PE) functionality and a configurable number of communication channels to match application demands. Compared to recent FPGA overlays like CGRA-ME's ADRES and HyCUBE implementations, our design operates at 1.8–3.4× faster clock frequency, while scaling to an orders-of-magnitude larger array size of 19×69 on Xilinx Alveo U280. Our communication-aware compiler targets HLS objectives such as initiation interval (II) and minimizes communication cost using an integer linear programming (ILP) formulation. Unlike SDC schedulers in FPGA HLS tools, we treat data movement as a first-class citizen by encoding the space and time resources communication network of the overlay in the ILP formulation. Given the same constraints on operational resources as Vivado HLS, we can retain our target II and achieve up to 9.2× higher frequency. Our ILP scheduler outperforms a PathFinder space-time router implementation in quality of result by up to nearly 2×.**

I. Introduction

With the slowdown in Moore's Law limiting improvements to CPU performance scaling [1], specialized accelerators can increase computational throughput and efficiency of computations across various application domains. Field-Programmable Gate Arrays (FPGAs) are being adopted by cloud service providers such as Microsoft, IBM, and Intel [2] [3] [4], but low-level register-transfer level (RTL) development for these platforms is difficult and expensive. One solution is the use of High-Level Synthesis (HLS) to express the desired algorithmic behavior in high-level C/C++ rather than low-level RTL, but this approach can suffer from inefficiencies and overheads resulting in low frequency at scale (up to 9× away from peak, demonstrated in Section VI). We still need to complete FPGA place and route times that can take hours or days of run-time for large cloud-scale designs. Instead, we can use structured coarse-grained FPGA overlays with careful floorplanning that trade-off FPGA flexibility in exchange for performance guarantees and potential for faster application mapping times [5].

Coarse-grained reconfigurable arrays (CGRAs) are a class of programmable logic devices consisting of several coarse logic blocks such as adders and multipliers connected together

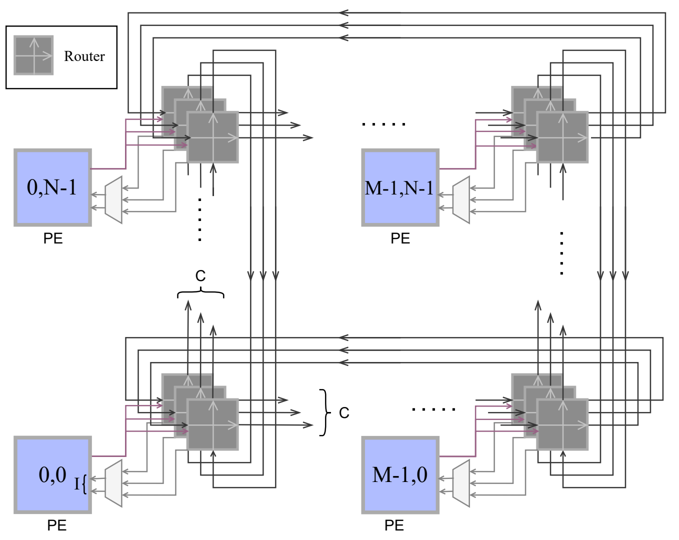


Fig. 1. An $M \times N$ Mocarabe array with I-input PEs and a C channel NoC.

with reconfigurable interconnect to provide high performance for specific classes of applications. Each logic block can have multiple inputs and outputs, a configurable ALU, and provide support for storage and data movement. In contrast to FPGAs that contain finer-grained logic blocks (e.g. look-up-tables) supported by bit-level interconnect, CGRAs are less flexible and provide ALU-like building blocks with bus-oriented interconnect. One can naively overlay existing CGRAs on top of modern FPGAs [3] [4], but these overlays do not fully leverage the benefits of operating on an FPGA. They have small fixed array sizes (e.g. 4×4 [6] [7][8]), and operate at low frequencies (226–382 MHz [6][8]).

In this work, we introduce Mocarabe, a flexible FPGA overlay CGRA that supports variable array sizes (up to 19×69) and operates at high frequencies (645–768MHz) even for large array sizes on the Xilinx Alveo U280 chip. Mocarabe offers rich interconnect flexibility in the form of multiple channels and logic block I/Os that are essential for supporting communication-rich dataflow kernels. Our compiler is an interconnect-aware ILP scheduling formulation which routes data movement in space and time while minimizing communication costs.

The key contributions of this work are:

- Design of a CGRA architecture with pipelined inter-

connect for high-frequency operation with support for multiple communication channels.
- A CGRA compiler with a communication-aware ILP scheduler formulation backbone that encodes interconnect resources in the ILP.
- Comparison of the ILP scheduler with a PathFinder-based space-time router.
- A carefully floorplanned implementation of a 19×69 Mocarabe configuration with up to 3 channels on the Xilinx Alveo U280 for 645–693 MHz operation frequency.

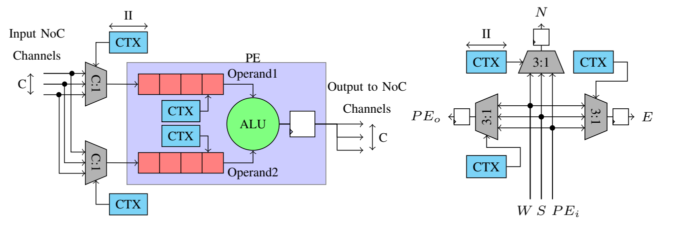
II. Background

In this section, we will cover notable and relevant CGRAs. CGRAs are surveyed in-depth in [9] [10] [11] [12]. We summarize results in Table I.

TABLE I
Review of Existing CGRAs

CGRA	Frequency (MHz)	FPGA/Technology	size
FPGA			
ADRES	382	Ultrascale+	4×4
ADRES	260	Stratix 10	4×4
HyCUBE	307	Ultrascale+	4×4
HyCUBE	226	Stratix 10	4×4
ASIC			
HyCUBE	704	28 nm	4×4
ULP-SRP	100	40nm	3×3
Cascade	510	40nm	4×4

- ADRES [13] is composed of multiple PEs arranged in a 2D array, and couples the reconfigurable array with a Very Long Instruction Word (VLIW) processor by integrating them into a single architecture. Each PE is connected to a Register File (RF) and its neighboring PEs, and has an ALU that operates on the data it receives from the RF or the neighboring PEs. The PEs at the top row are more capable and form the VLIW view.
- Ultra low-power Samsung Reconfigurable Processor (ULP-SRP) [6] is a variation of ADRES that has runtime-configurable low and high performance modes. ULP-SRP is composed of a 3×3 reconfigurable array and a VLIW processor. In high performance mode, the entire 3×3 array is active, while a 2×2 subset of the array is used for low performance mode. Every application must be mapped to both modes. ULP-SRP is implemented in 40nm ASIC and runs at 100MHz.
- HyCUBE [14] is a CGRA with reconfigurable single-cycle multi-hop interconnect. The single-cycle multi-hop interconnect enables distant PEs to exchange data with minimal timing overheads. HyCUBE is implemented as a 4×4 array on 28nm ASIC and operates at 704 MHz.
- Cascade [7] aims at high-throughput data streaming by decoupling memory accesses from computations. By moving the address generation outside the CGRA, Cascade reduces address generation overhead and makes the array focus on computations. Cascade is implemented as a 4×4 array in 40nm CMOS and reaches a maximum operating frequency of 510 MHz.



(a) PE internals (b) Router internals

Fig. 2. Building blocks of Mocarabe CGRA (CTX=Context Memory).

- CGRA Modelling and Exploration (CGRA-ME) [15] is an open-source framework that allows describing arbitrary CGRA architecture and enables mapping, placement, and scheduling C benchmarks to the arbitrary CGRA. It generates Verilog code for the resulting design for simulation and synthesis. CGRA-ME also allows for area and performance modeling of CGRAs [16] by synthesizing commonly occurring primitives in isolation and adding component wise results together. ADRES and HyCUBE are implemented as FPGA overlays using CGRA-ME on Intel Stratix 10 (S10) and Xilinx Ultrascale+ (US+) in [8]. After optimizations, the ADRES and HyCUBE overlays operate at up to 226–382 MHz on S10 and US+.

HLS schedulers, such as Vivado's scheduling SDC formulation [17], have to contend with many types of constraints: data dependency constraints, timing constraints such as latency and frequency constraints, and resource constraints. A well-designed CGRA architecture and compiler can alleviate the need to account for all those issues in the scheduler while still effectively coordinating data movement. While SDC scheduling is a powerful tool, it provides no frequency guarantees and must balance the aforementioned constraints, which hurts its resource sharing performance as shown in Section VI.

III. CGRA Architecture

The Mocarabe architecture consists of a 2D array of building blocks connected by a directional torus network-on-chip (NoC) as shown in Figure 1. Each block contains both a PE to execute operations on incoming data and a set of NoC routers to control data movement.

- A PE can be configured as either an operator (multiply or add) or a data input/output. PEs store incoming operands in shift registers and select the relevant stored operands as inputs to their ALU at each cycle, as shown in Figure 2a. Operand selection at each cycle is extracted from the compiler output.

- A key feature of our architecture is the variable number of parallel physical communication channels [18]. Every router accepts inputs from the local PE and the south and west neighbors on the same channel and sends outputs north, east, and to the local PE. A single-channel router is shown in Figure 2b.

TABLE II
PE AND ROUTER RESOURCE USAGE FOR II=4

	LUT (logic)	LUT (memory)	FF	DSP blocks
Adder PE	35	64	197	0
Multiplier PE	3	48	169	3
Router	99	0	290	0

Table II shows the resource consumption of each PE and router on average. The entire architecture is designed for statically-scheduled, time-multiplexed operation. With an initiation interval or (context count) II, every routing and functional resource will repeat the same task, accept inputs, and drive outputs in a repeating phase of II cycles. II is thus also the number of operations mapped to a resource which can enable larger applications to be mapped to fewer blocks at the cost of more LUTs to drive multiplexer select lines (context memories are labeled "CTX" in Figures 2a and 2b). II is the number of cycles in the modulo schedule found by the compiler. Operation execution and data movement are statically scheduled and encoded as multiplexer select line memories, as outlined in section IV.

Unlike other CGRAs [13][6][14], which have fixed array sizes, an application can be mapped over a subset of all available PEs and unrolled (repeated) by tiling over the full array.

If the number of communication channels is greater than one, PE inputs are fanned in from each channel to both shift registers. Figure 1 shows an $M \times N$ array with C communication channels. Each PE has two inputs and fan-in muxes are used to connect PEs to the NoC.

IV. COMPILER

The Mocarabe compiler framework extracts the data-flow graph (DFG) from a C kernel (with gcc and GIMPLE) and generates an architecture configuration and a schedule to coordinate data movement. The compiler flow, shown in Figure 3, consists of four phases: ① operator allocation, ② DFG partitioning, ③ placement, and ④ scheduling; the objective is to find a feasible schedule with a minimum number of channels. We use simple linear search for channels, and most of our benchmarks need two or three channels when scheduled with ILP, except for one outlier, `deriche`, which needs four.

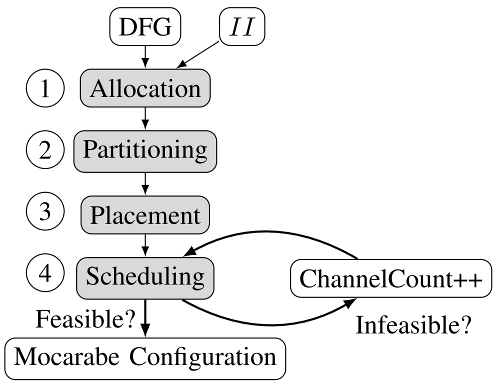


Fig. 3. Mocarabe compiler flow.

Compared to SDC scheduling [17], data dependencies are handled differently in our scheduler. We schedule each DFG edge in isolation, but use rotating registers to ensure correct alignment of data dependency. Frequency constraints ('cycle time') found in SDC schedulers are not found in the compiler, as the overlay is guaranteed to run close to the FPGA fabric f_{max} (645–768 MHz).

Unlike the CGRA-ME ILP scheduler [19] that unifies partitioning, placement, and scheduling into a monolithic ILP formulation, we split these tasks into separate disjoint phases to ensure feasible computational runtime for large problem sizes. The CGRA-ME scheduler routinely times out after 24 hours for a benchmark with an operation count under 30. Our compiler can tackle 80 operations in less than 30 minutes, with most benchmarks taking less than a minute.

A. Formal Description

The DFG is encoded in a hypergraph format [20] to retain multi-fanout attributes. A DFG is comprised of a set of nodes, Ops, which represent operations, inputs, and outputs, while edges, $Vals$, represent the data dependencies between Ops. The kernel in Figure 4 is mapped to the architecture as a motivating example, with $II = 2$ (dual-context).

B. Operator Allocation and Partitioning ① ②

Given a context count II, the compiler allocates $M \times N = K$ PEs DFG Ops are partitioned among the K PEs, each PE holding at most II Ops to allow for time-multiplexed operator sharing. We use an ILP formulation for hypergraph partitioning [21] that is solved using the Gurobi solver [22].

Variables: The formulation has two binary variables.

- $ValInPartition_{j,k} = 1$ indicates that DFG edge j is entirely in partition k.
- $OpInPartition_{op,k} = 1$ indicates that DFG node op is in partition k.

Constraints:
Every operator must be in exactly one partition.

$$\sum_{k}^{K} OpInPartition_{op,k} = 1, \forall op \in Ops \tag{1}$$

No more than II operators can be mapped to one partition.

$$\sum_{op}^{Ops} OpInPartition_{op,k} \leq II, \forall k \in K \tag{2}$$

Graph dependency is encoded with the following constraint. $Op(j)$ denotes all DFG nodes incident on edge j.

$$OpInPartition_{op,k} >= ValInPartition_{j,k}, \quad \forall j \in Vals, \forall op \in Op(j), \forall k \in K \tag{3}$$

There is a fixed number of partitions for each operator type. $K(op)$ denotes the partitions reserved for op's operator type (e.g. '*').

$$\sum_{k}^{K(op)} OpInPartition_{op,k} = 1, \forall op \in Ops \tag{4}$$

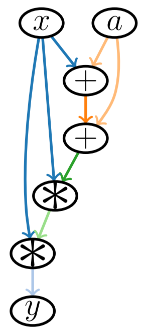


Fig. 4. Example Dataflow for $y = (2a + x) \times x^2$.

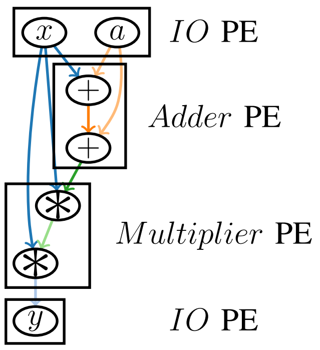


Fig. 5. Partitioned DFG into CGRA PEs.

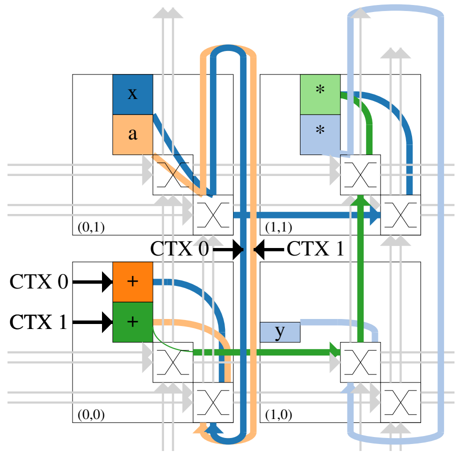


Fig. 6. Placement and schedule if CGRA

Objective Function:
The objective function is set to minimize the sum of cut nets ($Vals$), which is encoded as the maximization of uncut nets.

$$Maximize \sum_{j}^{Vals} \sum_{k}^{K} ValInPartition_{j,k} \tag{5}$$

A partition may only group operations of the same type into groups no larger than II. The motivating example has four operator types (input, add, multiply, output) and the resulting partitioning is shown in Figure 5. Here, the sum is two uncut nets, one between both adds and another between both multiplies. Note that in general, operations need not be neighbours to be in the same partition.

C. Placement ③

Every DFG operator is now mapped to one of K partitions, which must then be placed in a specific (x, y) location. We use simulated annealing [23], an approach which has been used in other CGRA compilers [15]. Any PE location can be fixed to be any type of operator. The placer's objective is to minimize the minimal torus distance a net must travel, as in (6). The placer and the scheduler are decoupled, but the aim is to provide the next stage with a placement that will enable it to find a feasible solution using the fewest parallel channels. Moves from one placement state to another are unrestricted. The cost of a certain placement is the sum, across every source-destination pairs, of the squared distance each net would have to travel through the torus interconnect. The result is a mapping from every DFG operator to its PE, which is used to create a netlist to be scheduled in the next step.

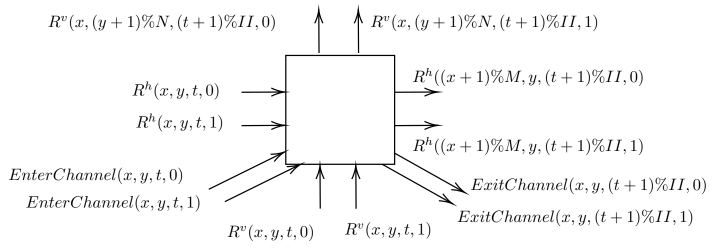
$$Minimize \sum_{j}^{Vals} (MinTorusDistance(source(j), dest) \\ \forall dest \in dests(j)). \tag{6}$$

D. Scheduler ④

SDC schedulers encode the execution cycle of an operation as an integer. Our scheduling problem, formulated as an ILP and solved with the Gurobi solver [22], encodes resource occupancy in space and time as boolean unknowns. The resulting ILP is larger, but we make it feasible by limiting schedule length (the time dimension) to II and realigning dependencies after scheduling. The input is a netlist of sources and destinations on the array with the same number of Ops and $Vals$ as the input DFG, but with decoupled dataflow dependencies (which will be realigned after scheduling). The resulting modulo schedule has up to II cycles. We now show how to set up this formulation.

1) ILP Variables: We define six sets of binary variables, grouped into pairs. The connectivity between some of these is illustrated in Figure 7.

- $R^h_{(x,y,t,c),j}$ and $R^v_{(x,y,t,c),j}$: routing resource at (x, y) on channel c is used by value j in cycle t. Horizontal and vertical routing resources are denoted by ^{h} and ^{v}, respectively. $R^{h,v}$ indicates that a constraint applies to both types of resources independently.
- $EnterRouting_{(x,y,t),j}$: At cycle t, value j leaves PE (x, y). $EnterChannel_{(x,y,t,c),j}$ specifies which channel to use.
- $ExitRouting_{(x,y,t),j}$: At cycle t, value j enters PE (x, y). $ExitChannel_{(x,y,t,c),j}$ specifies which channel to use.



$R^v(x,(y+1)\%N,(t+1)\%II,0)$
$R^v(x,(y+1)\%N,(t+1)\%II,1)$
$R^h((x+1)\%M,y,(t+1)\%II,0)$
$R^h(x,y,t,0)$
$R^h(x,y,t,1)$
$R^h((x+1)\%M,y,(t+1)\%II,1)$
$EnterChannel(x,y,t,0)$
$EnterChannel(x,y,t,1)$
$ExitChannel(x,y,(t+1)\%II,0)$
$ExitChannel(x,y,(t+1)\%II,1)$
$R^v(x,y,t,0)$
$R^v(x,y,t,1)$

Fig. 7. ILP Variables for Scheduling.

For the sake of brevity, we denote the tuple (x, y, t) as "i". For example, $R^h_{(i,c),j}$ represents value j's use of the horizontal routing resource at (x, y, t) on channel c. A denotes $M \times N \times II$, the cube over the 2D array and the schedule length II.

2) ILP Constraints and Objective Function: We present a number of constraints for the ILP formulation. The first six sets are somewhat trivial, while the last is the core of how the formulation encodes data movement.

Source/Destination Mapping: A value must leave its source exactly once, but can do so any time. For all values j in $Vals$, with source PE $src_{x,y}(j)$,

$$\sum_{t=0}^{T} EnterChannel_{(src_{x,y}(j),t,c),j} = 1. \tag{7}$$

A value j must enter all of its destinations exactly once. For every j in $Vals$ with destination $dest_{x,y}(j)$,and for all $t \in II$,

$$\sum_{t=0}^{T} exitRouting_{(dest_{x,y}(j),t),j} = 1. \tag{8}$$

A value cannot enter or exit any PE that is not its source or one of its destinations. For every j in $Vals$, and every location which is not one of j's destinations, $notDest_{x,y}(j)$, and for all $t \in II$,

$$\sum_{t=0}^{T} exitRouting_{(notDest_{x,y}(j),t),j} = 0. \tag{9}$$

Routing Resource Exclusivity: Each routing resource may be used by at most one value in each cycle.

$$\sum_{j}^{Vals} R^{h,v}_{(i,c),j} \leq 1, \forall i \in A, \forall c \in C. \tag{10}$$

PE Output: A PE can emit at most one value per cycle, and cannot enter multiple channels simultaneously.

$$\begin{gathered}\sum_{j}^{Vals} EnterChannel_{(i,c),j} \leq 1, \forall i \in A, \forall c \in C \\ \sum_{j}^{Vals} EnterRouting_{(i),j} \leq 1, \forall i \in A.\end{gathered} \tag{11}$$

Single-Channel Entry: When a value enters the NoC, it must choose a single channel. For all $i \in A, c \in C, j \in Vals$,

$$\sum_{c}^{C} EnterChannel_{(i,c),j} \geq EnterRouting_{(i),j} \tag{12}$$

$$EnterChannel_{(i,c),j} \leq EnterRouting_{(i),j}. \tag{13}$$

PE Input: A PE can absorb at most 2 values from the NoC each cycle.

$$\sum_{j}^{Vals} ExitChannel_{(i,c),j} \leq 2, \forall i \in A, \forall c \in C. \tag{14}$$

Single-Channel Exit: Similarly, when a value exits a PE, it must choose a single channel. For all $i \in A, c \in C, j \in Vals$

$$\sum_{c}^{C} ExitChannel_{(i,c),j} \geq ExitRouting_{(i),j} \tag{15}$$

$$ExitChannel_{(i,c),j} \leq ExitRouting_{(i),j}. \tag{16}$$

Value Propagation: These core constraints illustrate the interconnect's torus connectivity and modulo scheduling.

This constraint ensures that information cannot be created from nothing. This is encoded for horizontal routing (17), vertical routing (18), and for leaving the NoC (19). For all $(x, y, t) \in A, c \in C, j \in Vals$,

$$\begin{gathered}R^h_{(x,y,t,c),j} + R^v_{(x,y,t,c),j} + EnterChannel_{(x,y,t,c),j} \geq \\ R^h_{((x+1)\%M,y,c,(t+1)\%II),j},\end{gathered} \tag{17}$$

$$\begin{gathered}R^h_{(x,y,t,c),j} + R^v_{(x,y,t,c),j} + EnterChannel_{(x,y,t,c),j} \geq \\ R^v_{(x,(y+1)\%N,c,(t+1)\%II),j},\end{gathered} \tag{18}$$

$$\begin{gathered}R^h_{(x,y,t,c),j} + R^v_{(x,y,t,c),j} + EnterChannel_{(x,y,t,c),j} \geq \\ ExitChannel_{(x,y,c,(t+1)\%II),j}.\end{gathered} \tag{19}$$

A value must fan-out to at least one routing resource, i.e. information cannot be destroyed. For all $\forall i \in A, \forall c \in C, \forall j \in Vals$

$$\begin{gathered}R^h_{((x+1)\%Nx,y,c,(t+1)\%II),j} + R^v_{(x,(y+1)\%Ny,c,(t+1)\%II),j} + \\ ExitChannel_{(x,y,c,(t+1)\%II),j} \\ \geq R^h_{(i,c),j} + R^v_{(i,c),j} + EnterChannel_{(x,y,t,c),j}.\end{gathered} \tag{20}$$

Objective Function: Our objective function minimizes the sum of all routing resources used, across every PE and every location.

$$\sum_{i}^{A} \sum_{c}^{C} \sum_{j}^{Vals} R^h_{(i,c),j} + R^v_{(i,c),j} \tag{21}$$

Scheduling concludes the generation of a repeatable schedule for a fixed size array, which can then be replicated over a chip. Figure 6 illustrates a schedule of the motivating example. Inputs leave the IO PE at $(0, 1)$ and both use the same router and the same channel in different contexts (cycles). Both x and a propagate to the adder PE at $(0, 0)$ and, in x's case, to the multiplier PE at $(1, 1)$, and so on until the final multiply is propagated to the IO PE at $(1, 1)$. This dual-context run ($II = 2$, represented by two close parallel lines) was mapped with $C = 2$ channels which is required because of PE self-communication (e.g. the result of one add is the input to another, and there is one adder).

E. Configuration

After an appropriate schedule is computed, we generate a hardware configuration with the final channel count, II, and PE operator arrangement. The static schedule is processed to generate context memories for all routers, PE input multiplexers, and operand rotating register addresses, which are then fed into the synthesis tool. Here, the array can be unrolled to the greatest extent allowed by the chip-specific implementation (e.g. a 3x19 array can be unrolled 23 times to utilize the entire overlay).

V. CGRA Implementation and floorplanning

We implement the Mocarabe overlay using parametric Verilog for PEs and switches. We use Xilinx Vivado 2020.1 to synthesize, place, and route the design on a Xilinx Alveo U280 card for analysis. We design hand-crafted placement scripts to effectively map the design and make use of FPGA resources while keeping the operation frequency high. Each logical block containing PE and switches is assigned to a physical block (Pblock) on the chip. We define an arbitrary estimate for each Pblock's size as the number of logic slices it contains, with each slice containing Look-Up Tables (LUTs) and flip flops. For instance, a 10×10 Pblock can span the chip from slice X0Y0 to slice X9Y9, creating a rectangular area over the device that contains 100 slices.

As the first step, we use folded layout for the physical placement of logic blocks and routers to reduce the torus critical path delay.

During our floorplanning experiments we notice that the unused portions at each Pblock add to routing delay. We place and route a 10×10 array for varying Pblock sizes up to 100 slices shown in Figure 8. Compact Pblocks generally deliver higher frequency with the 8×10 PE achieving the highest frequency of 980MHz. To scale up the CGRA and

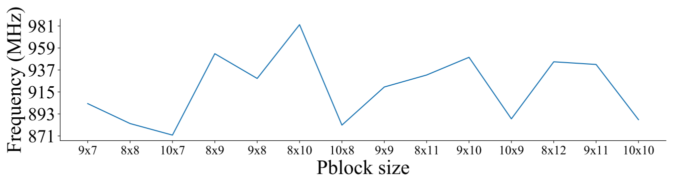


Fig. 8. Frequency as a function of block size for a 10×10 Mocarabe array.

span the FPGA, we set the array size to 20×60, with the same configuration of having multipliers on the first column and the rest of the blocks being adders. Spanning the entire FPGA required communication between different Super Logic Regions (SLRs) which lead to large routing delays between SLRs. We fixed Pblock size to 10×10 in order to avoid having single Pblocks spread among multiple SLRs. Furthermore, we added extra pipeline registers to each router's output to the nearby routers and registered the incoming data. We forced Vivado to map the router registers that had incoming or outgoing SLR crossing nets to SLR crossing Laguna registers, which are connected together by Super Long Lines (SLLs), inside the router's corresponding Pblock. We use Vivado's "phys_opt_design" compiler option to include various physical optimizations (e.g. an SLR-crossing optimization), once after placement and once after routing to enable the 20×60 array to run at 650 MHz+.

Although the 20×60 array operates at 650 MHz, the operation frequency is not stable between different operator configurations. For instance, having all blocks performing multiplication results in a noticeable frequency drop to 400 MHz. The drop in frequency is because some PEs use DSPs placed far away outside their Pblocks. To overcome this issue, we modify the placement scripts to force each PE to only use the DSPs located inside its Pblock and added paddings to array placement on the chip to make sure each Pblock contained at least the amount of DSP blocks needed by each PE.

TABLE III
CGRA sizes and frequencies

CGRA	Frequency (MHz)	Array size
Mocarabe (C=1)	711–768	19×69
Mocarabe (C=2)	691–750	19×69
Mocarabe (C=3)	645–693	19×69
Mocarabe (C=1)	813–921	4×4
ADRES (Ultrascale+)	382	4×4
ADRES (Stratix10)	260	4×4
HyCUBE (Ultrascale+)	307	4×4
HyCUBE (Stratix 10)	226	4×4

The 10×10 size for each Pblock is a suitable option here as well to make sure all Pblocks receive the resources they need. By using the updated placement scripts, we are able to scale the array size to 19×69, spanning the entire chip and operate at 640 MHz for an array composed only from multiplier PEs. To further increase the operation frequency we pipelined the router outputs to PEs, which improved operation frequency, taking the all-multiplier 19×69 array frequency up from 640 – 740MHz. Since some applications at a given II require higher channel counts, we support 1–3 channel, two-input PE configurations.

VI. Experimental results

In the following section, we showcase our benchmark results, compare our overlay to similar recent CGRA overlays, evaluate our CGRA against Vivado HLS, and compare a PathFinder implementation with our ILP implementation. Experiments were run with GNU parallel [24].

A. Benchmarks

Table IV shows an overview of every benchmark's size and minimum channel count obtained by our ILP scheduler. All benchmarks can run on an implemented channel count of 3 or fewer, except `deriche` with $II = 4$ which requires 4.

TABLE IV
Mocarable compiler II and Channel count results

Benchmark	DFG Size			Channel Count				
	I/Os	Adds	Multiplies	II=1	II=2	II=3	II=4	II=5
adder_chain	5	3	0	2	2	2	2	2
approx1	6	4	3	2	2	2	3	3
bellido	4	5	3	2	2	2	2	3
caprasse	6	3	6	2	2	2	2	2
dct	23	32	22	3	3	2	2	2
deriche	49	35	45	3	3	3	4	3
fig3	4	2	1	2	2	2	2	2
fir32	67	32	33	2	3	2	2	2
gaussian	19	8	9	2	2	2	2	2
iir8	31	7	14	2	2	2	2	2
level1_linear	5	2	2	2	2	2	2	2
level1_saturation	7	5	2	2	2	2	3	3
poly10	12	9	9	2	2	2	2	2
poly20	21	19	9	3	2	2	2	3
poly3	5	2	3	2	2	2	2	3
poly4	6	3	4	2	2	2	2	3
poly6	8	5	6	2	2	2	2	2
poly8	10	7	7	2	2	2	2	2
poly_quadratic	5	2	2	2	2	2	2	2
rgb	15	3	9	2	2	2	2	3
sobel	11	11	4	2	2	2	2	3

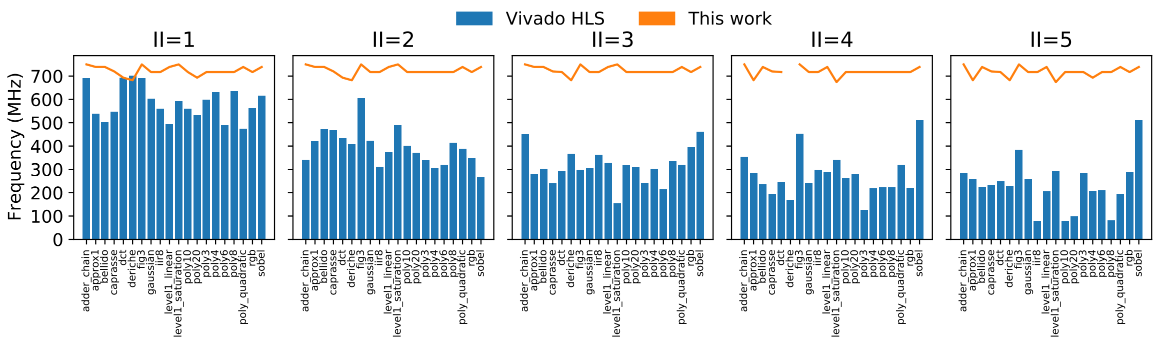


Fig. 9. Vivado HLS f_{max} vs. this work (`deriche` at $II = 4$ needs C=4 that is not supported).

B. CGRA Overlay

We implemented Mocarabe on a Xilinx Alveo U280 using the floorplanning described in Section V to carefully optimize the design. Mocarabe is available in four different configurations:

- A 19×69 single-channel ($C = 1$), single-input PE
- A 19×69 two-channel ($C = 2$), two-input PE
- A 19×69 three-channel ($C = 3$), two-input PE
- A 4×4 single-channel ($C = 1$), single-input PE (to compare with other CGRAs)

To demonstrate high operation frequencies across a spectrum of operator configurations (multipliers and adders), each CGRA configuration was tested with 10–100% of PEs as multipliers (in steps of 10%), randomly distributed across the array, with the remaining PEs configured as adders. Figure 10 shows the operation frequencies of different configurations. The single-channel $C = 1$, 16×69 array ran at 748 MHz on average, the two- channel $C = 2$, 16×69 array ran at 717 MHz on average, the three-channel $C = 3$, 16×69 array ran at 675 MHz on average, and the 4×4 array ran at 864 MHz on average. There are small frequency variations between different PE configurations, and Mocarabe maintains high operation frequencies regardless of the PE configurations because of Pblock sizing and adequate pipelining.

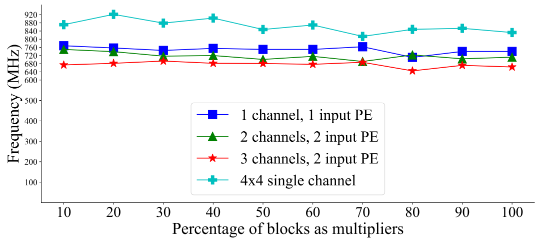


Fig. 10. Frequency for different channel-count and PE configurations.

We compare Mocarabe to recent CGRA-ME [8] overlay implementations of ADRES [13] and HyCUBE [14]. In [8], the authors implement 4×4 ADRES and HyCUBE CGRAs as overlays on the Xilinx Ultrascale+ XCVU3P and the Intel Stratix 10 GX850. The 32-bit ADRES overlay operates at up to 382 MHz on average on the Ultrascale+ and up to 260MHz on average on the Stratix 10. The 32-bit HyCUBE overlay operates at up to 307 MHz on average on the Ultrascale+ and up to 226 MHz on average on the Stratix 10. Mocarabe maintains higher operation frequencies while having orders-of-magnitude larger array sizes. In Table III, we show operating frequencies for different CGRAs. For Mocarabe we provide frequency ranges, as frequency varies in different operator configurations. In Table V, we show the clock frequency gains for different Mocarabe configurations when compared to ADRES and HyCUBE overlays.

TABLE V
MOCARABE FREQUENCY GAINS OVER OTHER CGRAS

CGRA	C=1	C=2	C=3	C=1 (4×4)
ADRES (U+)	$1.8 - 2\times$	$1.8 - 2\times$	$1.7 - 1.8\times$	$2.1 - 2.4\times$
ADRES (S10)	$2.7 - 3\times$	$2.7 - 2.9\times$	$2.5 - 2.7\times$	$3.1 - 3.5\times$
HyCUBE (U+)	$2.3 - 2.5\times$	$2.3 - 2.4\times$	$2.1 - 2.3\times$	$2.6 - 3\times$
HyCUBE (S10)	$3.1 - 3.4\times$	$3.1 - 3.3\times$	$2.9 - 3.1\times$	$3.6 - 4\times$

C. Vivado HLS

To compare with Vivado HLS, we implement each benchmark with a given initiation interval and unroll the resulting array to take advantage of every available PE (19×69). We give Vivado HLS the same benchmark, II, unroll factor and frequency target. Though Vivado provides the option to constrain the number of adders and multipliers, we outperform Vivado without providing these constraints, as shown in Figure 9. Vivado HLS can, in some cases, retain a high f_{max} for benchmarks with no resource sharing ($II = 1$), but this quickly drops off by factors of up to $2\times$ for $II = 2$, $4.5\times$ for $II = 3$, and $5.5\times$ for $II = 4$ as more operations are assigned to one functional unit. At $II = 5$, the lowest f_{max} achieved by Vivado HLS is with the `poly10` benchmark at $II = 5$ at 78.06 MHz, $9\times$ away from the peak at 750 MHz. These results do not apply any resource constraints to Vivado HLS.

For all benchmarks, Mocarabe has met the target II. Vivado HLS can also meet target II when not given any resource constraints. However, when we count the number of functional units we use and force Vivado HLS to use the same, Vivado's resource sharing simply fails beyond $II = 1$, as shown in Figure 11. II can increase by up to 2 cycles for $II = 4$, and up to 4 for $II = 5$. Vivado HLS timed out (did not finish in 24 hours) for one benchmark, `fir32`. Table VI compares Vivado HLS resource usage to Mocarabe resource usage. Mocarabe uses a similar number of DSP blocks in most cases while using more LUTs, which is in part due to the communication network. Vivado HLS failed to implement `dct`, `poly10`, and `poly20` for targeting $II = 5$ with the same unroll

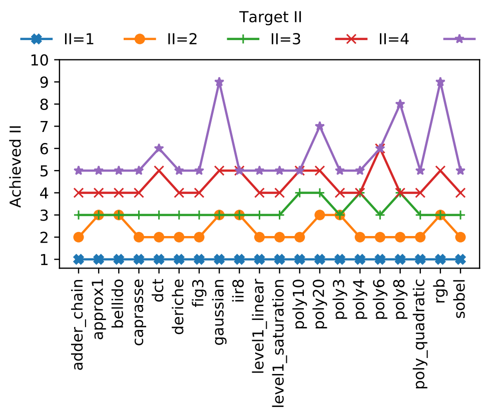


Fig. 11. Achieved II vs. Target II for various dataflow benchmarks in Vivado HLS. Mocarabe always meets target II.

as us. Furthermore, `deriche` could not be implemented on our current design for targetting $II = 4$ as it needs more channels. Implementation failures are indicated by "0" in the table. However, as each multiplier PE uses 3 DSP blocks in our design, we are providing comparison data for `deriche` $II = 4$ DSP usage. Table VII compares Mocarabe latency in cycles to Vivado HLS latency. Mocarabe latency can be up to 5.6× (2.1× - 2.4× mean) larger than Vivado HLS depending on the number of communication channels. This is because the PE takes 6-8 cycles and the router takes 2-3 cycles for high-frequency pipelined operation.

D. PathFinder Space-Time Scheduler

To explore possible tradeoffs between quality-of-result (channel count) and runtime, we implement a space-time scheduler with a PathFinder approach, inspired by [25] which can be swapped in to replace the ILP scheduler in our compiler. The algorithm routes over a Modulo Routing Resource Graph (MRRG [15]), which is a graph representation of every PE and routing resource over time, with multiple stages of rip-up and re-route. As shown in Figure 12, the algorithm almost never outperforms the ILP formulation by mapping to a lower channel count configuration, and channel count can be as high as 2× higher with the PathFinder scheduler. In all cases the ILP formulation was faster with the exception of `deriche` for $II = 2$.

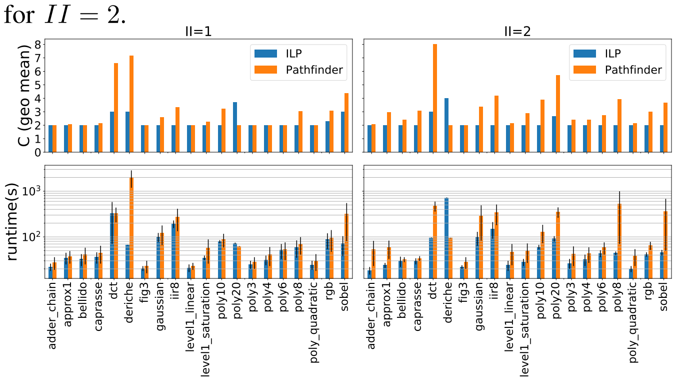


Fig. 12. ILP vs PathFinder Scheduler, Channel Count and Runtime.

TABLE VI
MOCARABE VS VIVADO HLS RESOURCE USAGE
(MOCARABE/VIVADO HLS)

Benchmark	II=1		II=2		II=3		II=4		II=5	
	DSP	LUT	DSP	LUT	DSP	LUT	DSP	LUT	DSP	LUT
adder_chain	–	22.6	–	18.8	–	18.6	–	5.3	–	4.9
approx1	1.0	23.4	1.0	9.4	1.0	7.5	1.0	10.6	1.0	7.2
bellido	1.0	46.3	0.9	16.1	1.0	7.9	1.0	9.2	1.0	9.5
caprasse	1.5	21.2	1.0	6.7	1.0	5.1	1.5	9.3	1.5	7.4
dct	1.0	47.9	1.0	13.8	1.0	6.8	1.0	4.0	0	0
deriche	1.4	58.5	1.0	19.0	1.0	9.2	1.0	0	1.0	6.5
fig3	1.0	20.4	1.0	16.1	1.0	12.0	1.0	6.0	1.0	6.6
gaussian	1.0	31.9	1.0	10.5	1.0	7.8	1.0	4.1	1.0	3.2
iir8	0.9	12.7	1.0	10.8	1.0	7.3	1.0	3.9	1.0	3.5
level1_lin	1.0	25.7	1.0	8.0	1.0	9.7	1.0	6.7	1.0	6.4
level1_sat	1.0	21.3	1.0	9.3	1.0	5.1	1.5	10.5	1.5	9.4
poly10	0.9	19.4	1.0	5.6	1.0	4.3	1.0	2.9	0	0
poly20	0.9	24.3	1.0	5.1	1.0	4.7	1.0	3.0	0	0
poly3	1.0	16.7	1.0	6.9	1.0	8.2	1.0	7.1	1.0	8.2
poly4	1.1	33.8	1.0	4.3	1.0	6.9	1.1	12.3	1.1	7.8
poly6	1.0	25.0	1.0	5.4	1.0	3.8	1.0	4.0	1.0	4.0
poly8	0.9	20	1.0	4.8	1.0	4.0	1.0	2.6	1.0	2.8
poly_quad	1.0	85.4	1.0	13.3	1.0	37.4	1.0	8.0	1.0	8.2
rgb	1.0	28.2	1.0	11.3	1.0	10.6	1.0	4.7	1.0	7.0
sobel	4.0	32.6	2.0	14.9	1.0	14.1	1.0	12.5	1.0	18.8

TABLE VII
MOCARABE VS VIVADO HLS LATENCY IN CYCLES
(MOCARABE/VIVADO HLS RATIO IS SHOWN IN RED)

Benchmark	II=1		II=2		II=3		II=4		II=5	
	Moc	HLS	Moc	HLS	Moc	HLS	Moc	HLS	Moc	HLS
adder_chain	52	10 (5.2)	56	10 (5.6)	58	13 (4.5)	56	10 (5.6)	58	11 (5.3)
approx1	80	30 (2.7)	66	34 (1.9)	74	28 (2.6)	74	31 (2.4)	84	29 (2.9)
bellido	64	23 (2.8)	74	25 (3.0)	82	22 (3.7)	78	20 (3.9)	80	22 (3.6)
caprasse	66	30 (2.2)	88	27 (3.3)	94	28 (3.4)	92	31 (3.0)	96	29 (3.3)
dct	78	98 (0.8)	98	88 (1.1)	100	82 (1.2)	92	89 (1.0)	88	87 (1.0)
deriche	328	212 (1.5)	216	186 (1.2)	222	155 (1.4)	0	172 (0.0)	234	172 (1.4)
fig3	54	17 (3.2)	52	17 (3.1)	60	17 (3.5)	60	17 (3.5)	62	18 (3.4)
gaussian	120	45 (2.7)	130	58 (2.2)	126	43 (2.9)	144	45 (3.2)	152	56 (2.7)
iir8	158	119 (1.3)	170	166 (1.0)	180	136 (1.3)	190	140 (1.4)	210	131 (1.6)
level1_lin	54	25 (2.2)	60	24 (2.5)	62	24 (2.6)	66	24 (2.8)	66	25 (2.6)
level1_sat	66	31 (2.1)	62	28 (2.2)	72	30 (2.4)	70	32 (2.2)	68	30 (2.3)
poly10	202	104 (1.9)	210	106 (2.0)	218	108 (2.0)	222	101 (2.2)	230	98 (2.3)
poly20	432	206 (2.1)	410	238 (1.7)	430	233 (1.8)	444	214 (2.1)	460	210 (2.2)
poly3	76	32 (2.4)	72	34 (2.1)	78	31 (2.5)	74	33 (2.2)	80	32 (2.5)
poly4	100	44 (2.3)	94	40 (2.4)	96	48 (2.0)	98	41 (2.4)	106	40 (2.6)
poly6	130	64 (2.0)	130	62 (2.1)	142	59 (2.4)	148	68 (2.2)	142	68 (2.1)
poly8	166	83 (2.0)	158	82 (1.9)	172	84 (2.0)	174	80 (2.2)	192	96 (2.0)
poly_quad	60	25 (2.4)	56	25 (2.2)	58	24 (2.4)	68	24 (2.8)	66	25 (2.6)
rgb	44	33 (1.3)	50	47 (1.1)	46	36 (1.3)	36	34 (1.1)	46	54 (0.9)
sobel	102	32 (3.2)	98	32 (3.1)	110	30 (3.7)	102	29 (3.5)	104	33 (3.2)
geomean		2.2		2.1		2.3		2.4		2.3

VII. FUTURE WORK

To extend this work, accelerating FPGA placement by leveraging tools such as RapidWright [26] and exploiting the regular structure of out architecture (as in [27]) may eliminate the very painful task of finding an optimal floorplan for each target device and speed up the end-to-end flow. Breaking up the DFG into separate but adjacent arrays, when possible, would allow for even larger applications to be easily mapped. Coalescing different operations (e.g. sequential multiply and add) into one PE could reduce communication between PEs.

VIII. CONCLUSION

We introduce Mocarabe, a new CGRA overlay architecture for Xilinx FPGAs and its associated communication-aware compiler. We present a scalable and flexible FPGA overlay implementation engineered with generous pipelining and careful floorplanning to achieve 650MHz+ operation. Our implementation outperforms other CGRA overlay implementations on FPGAs with 1.8–3× higher frequency. At scale, our overlay outperforms Vivado HLS with up to 9.2× higher frequency. We can share functional resources very effectively while HLS struggles to do so at higher II. Mocarabe code can be found at: https://git.uwaterloo.ca/watcag-public/mocarabe.

References

[1] T. N. Theis and H. S. P. Wong, "The end of moore's law: A new beginning for information technology," *Computing in Science and Engg.*, vol. 19, no. 2, p. 41–50, Mar. 2017. [Online]. Available: https://doi.org/10.1109/MCSE.2017.29

[2] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang, "Enabling fpgas in the cloud," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, ser. CF '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2597917.2597929

[3] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang, "Sda: Software-defined accelerator for large-scale dnn systems," in *2014 IEEE Hot Chips 26 Symposium (HCS)*, 2014, pp. 1–23.

[4] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," *IEEE Micro*, vol. 35, no. 3, pp. 10–22, 2015.

[5] H. K.-H. So and C. Liu, *FPGA Overlays*. Cham: Springer International Publishing, 2016, pp. 285–305. [Online]. Available: https://doi.org/10.1007/978-3-319-26408-0_16

[6] C. Kim, M. Chung, Y. Cho, M. Konijnenburg, S. Ryu, and J. Kim, "Ulp-srp: Ultra low-power samsung reconfigurable processor for biomedical applications," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 3, Sep. 2014. [Online]. Available: https://doi.org/10.1145/2629610

[7] D. Wijerathne, Z. Li, M. Karunarathne, A. Pathania, and T. Mitra, "Cascade: High throughput data streaming via decoupled access-execute cgra," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3358177

[8] I. Taras and J. H. Anderson, "Impact of fpga architecture on area and performance of cgra overlays," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 87–95.

[9] A. Chattopadhyay, "Ingredients of adaptability: A survey of reconfigurable processors," *VLSI Design*, vol. 2013, 07 2013.

[10] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," *ACM Comput. Surv.*, vol. 52, no. 6, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3357375

[11] A. Podobas, K. Sano, and S. Matsuoka, "A survey on coarse-grained reconfigurable architectures from a performance perspective," *IEEE Access*, vol. 8, pp. 146 719–146 743, 2020.

[12] V. Tehre and R. Kshirsagar, "Survey on coarse grained reconfigurable architectures," *International Journal of Computer Applications*, vol. 48, pp. 1–7, 06 2012.

[13] B. Mei, S. Vernalde, D. Verkest, H. Man, and R. Lauwereins, "Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix," 09 2003, pp. 61–70.

[14] M. Karunaratne, A. Mohite, T. Mitra, and L.-S. Peh, "Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect," 06 2017, pp. 1–6.

[15] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "Cgra-me: A unified framework for cgra modelling and exploration," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Seattle, WA, 2017, pp. 184–189.

[16] K. Niu and J. H. Anderson, "Compact area and performance modelling for cgra architecture evaluation," in *2018 International Conference on Field-Programmable Technology (FPT)*, 2018, pp. 126–133.

[17] J. Cong and Zhiru Zhang, "An efficient and versatile scheduling algorithm based on sdc formulation," in *2006 43rd ACM/IEEE Design Automation Conference*, 2006, pp. 433–438.

[18] B. Van Essen, A. Wood, A. Carroll, S. Friedman, R. Panda, B. Ylvisaker, C. Ebeling, and S. Hauck, "Static versus scheduled interconnect in coarse-grained reconfigurable arrays," in *2009 International Conference on Field Programmable Logic and Applications*, 2009, pp. 268–275.

[19] S. A. Chin and J. H. Anderson, "An architecture-agnostic integer linear programming approach to cgra mapping," in *DAC '18*. San Francisco, CA: ACM, 2018.

[20] A. Ritz, B. Avent, A. Pratapa, and T. Murali. (2018) halp: Hypergraph algorithms package. [Online]. Available: https://github.com/Murali-group/halp

[21] D. Kucar, S. Areibi, and A. Vannelli, "Hypergraph partitioning techniques," *DYNAMICS OF CONTINUOUS DISCRETE AND IMPULSIVE SYSTEMS SERIES A 11*, 2004.

[22] L. Gurobi Optimization, "Gurobi optimizer reference manual," 2020. [Online]. Available: http://www.gurobi.com

[23] M. Perry. (2019) simanneal: Python module for simulated annealing. [Online]. Available: https://github.com/perrygeo/simanneal

[24] O. Tange, "Gnu parallel - the command-line power tool," *;login: The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, Feb 2011. [Online]. Available: http://www.gnu.org/s/parallel

[25] S. B. Patil, T. Liu, and R. Tessier, "A bandwidth-optimized routing algorithm for hybrid fpga networks-on-chip," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 25–28.

[26] C. Lavin and A. Kaviani, "Rapidwright: Enabling custom crafted implementations for fpgas," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 133–140.

[27] N. Zhang, X. Chen, and N. Kapre, "Rapidlayout: Fast hard block placement of fpga-optimized systolic arrays using evolutionary algorithms," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, 2020, pp. 145–152.