2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)

Impact of FPGA Architecture on Area and Performance of CGRA Overlays

Ian Taras and Jason H. Anderson
Dept. of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada
Email: {tarasian, janders}@ece.utoronto.ca

***Abstract*—Coarse-grained reconfigurable arrays (CGRAs) are programmable logic devices with ALU-style processing elements and datapath interconnect. CGRAs can be realized as custom ASICs or implemented on FPGAs as *overlays*. A key element of CGRAs is that they are typically software programmable with rapid compile times – an advantage arising from their coarse-grained characteristics, simplifying CAD mapping tasks. We implement two previously published CGRAs as overlays on two commercial FPGAs (Intel and Xilinx), and consider the impact of the underlying FPGA architecture on the CGRA area and performance. We present optimizations for the overlays to take advantage of the FPGA architectural features and show a peak performance improvement of 1.93×, as well as maximum area savings of 31.1% and 48.5% for Intel and Xilinx, respectively, relative to a naive first-cut implementation. We also present a novel technique for a configurable multiplexer implementation, which embeds the select signals into SRAM configuration, saving 35.7% in area. The research is conducted using the open-source CGRA-ME (modeling and exploration) framework [1].**

***Index Terms*—FPGA, Hardware, Optimization, FPGA Overlay, CGRA.**

I. Introduction

Coarse-grained reconfigurable arrays (CGRAs) are programmable hardware architectures with logic blocks containing ALU-like functionality, and datapath interconnect. In contrast, field-programmable gate arrays (FPGAs) contain a mix of fine and coarse-grained logic blocks (e.g., look-up-tables and hardened DSP blocks), and independently routed logic signals. Due to their coarseness, CGRAs are inherently less flexible than FPGAs, but offer superior performance and power for applications whose compute/communication needs are well aligned with CGRA capabilities. A compelling use case for CGRAs is efficient implementation of hardware accelerators for loop-heavy compute kernels (e.g. machine learning or bioinformatics applications). As clock frequency scaling in standard processors plateaus, CGRA-based accelerators are a path towards higher computational throughput and energy efficiency. Since major cloud vendors have deployed FPGAs in their data centers, we envision software-programmable CGRAs, implemented as FPGA overlays, becoming a key solution to acceleration.

An *overlay* is a programmable architecture implemented on top of another programmable architecture. Benefits include improved designer productivity, as overlays are typically at a higher level of abstraction than the underlying architecture, making them easier to program. In particular, FPGA-overlay CGRAs are beneficial because they raise the level of abstraction beyond that of FPGA design, where the accelerator can be described using C-code, instead of traditional HDL (much like high-level synthesis, but without the lengthy compile times). This is an enticing avenue for software designers producing digital hardware. Configuring a high-performance, area-efficient CGRA on top of an FPGA requires careful consideration of the architectural features of the underlying FPGA. In this paper, we address the question: how best to implement a CGRA overlay on an FPGA?

We consider CGRAs modeled using the open-source framework, CGRA-ME (modeling and exploration) [1]. We realize CGRAs on two high-end commercial FPGAs: Intel Stratix 10 and Xilinx Ultrascale+, considering two datapath widths: 16-bit and 32-bit. The overlays are analyzed to understand the major bottlenecks, and optimizations are applied to achieve FPGA-overlay CGRAs that yield high performance. By applying our optimizations, we can achieve an area savings of 35.7% on the overhead of multiplexing logic, and a peak area savings of 31.1% and 48.5% on Intel and Xilinx FPGAs, respectively.

The contributions of this paper are: 1) Model two CGRA architectures, ADRES and HyCUBE, using the open-source CGRA-ME modeling framework [1]–[3]; 2) Implement CGRA architectures as overlays on high-end FPGAs with optimizations, providing analysis of area and performance overheads; 3) Apply optimizations to CGRA overlay multiplexers, functional units, and RAMs that take advantage of the underlying FPGA architectural features, realized directly from CGRA-ME; and, 4) An experimental study illustrating the impact of our optimizations, datapath width, and floorplanning on CGRA overlay performance and area for both commercial devices.

II. Related Work

A number of prior works have considered implementing FPGA-overlay CGRAs with an emphasis on optimizing area and performance [4]–[7]. QUKU is a CGRA implemented as an FPGA overlay, configured as a 4×4 matrix of processing elements (PEs) consisting of ALUs and interconnect [4], [8]. The architecture was implemented on a Xilinx Virtex-4 FPGA and was compared to a Microblaze soft processor implementation. Although an ∼8× performance improvement is seen, QUKU was highly inflexible, as kernel scheduling and mapping for each PE had to be realized manually. Additionally, the authors did not account for potential architecture optimizations, as in the present work.

2576-2621/19/$31.00 ©2019 IEEE
DOI 10.1109/FCCM.2019.00022

Other works have focused on leveraging DSP blocks as optimization techniques, to enhance area and performance of FPGA-overlay CGRAs [5]–[7]. The DySER overlay was modeled after the DySER architecture, a mesh-network consisting of functional units (FUs) and switches, where the switches have 5 inputs and 8 outputs (requiring a 5:1 multiplexer), and the FUs realize mathematical operations (OPs) and handshaking/synchronization [5], [9]. The overlay was implemented on a Xilinx Zynq FPGA. Configured with a 16-bit datapath, the original implementation achieved a maximum operating frequency of 150MHz. The DSP-based DySER with fixed FU arrays achieved a maximum operating frequency of 375MHz. If the FU array is integrated into a DySER overlay, however, the performance decreases to 175MHz, shifting the bottleneck of the design to the interconnect and multiplexing overhead. This highlights the impact of interconnect and multiplexing logic on the overall performance of the design, which we specifically target in this work.

Jain et al. [7] present a CGRA overlay that emphasized DSP utilization. It consists of a 2D grid of FUs with nearest-neighbor interconnect, configured at 16-bits wide, with variable channel width (CW) to increase routing flexibility. The work includes a DSP-aware mapping tool, with the ability to merge nodes in the dataflow graph (DFG) that can be executed in the same cycle, reducing DSP utilization. When implemented on a Xilinx Virtex-6 FPGA, a frequency of 315MHz for a 15×15 array was achieved. The DeCO overlay [6] uses similar techniques to [7], with the improvement of shaping the architecture as a cone to reduce the routing overhead between PEs. When deployed on higher-end FPGAs, such as the Virtex-7, the cone shaped architecture can achieve a frequency of 645MHz. The work in [6], [7] focuses solely on DSP-related optimizations, whereas we take a broader approach to consider multiplexers, memories and floorplanning.

Poulos et al. [10] proposed a technique for leveraging FPGA reconfigurability to dynamically configure multiplexer (MUX) select signals by modifying the bitstream. By configuring the MUXes with the proposed technique, an average area savings of 35% is seen. Zuma also presents a method for implementing MUXes as LUTs, in LUTRAMs by configuring the LUTRAM as a pass-through [11]. A benefit to this is the ability to configure wide MUXes for datapath-oriented designs. Although a reduction in LUTs/MUX is seen, the process to realize the MUX configuration is tedious and requires a complicated toolflow. Our method directly configures LUTs as MUXes from within CGRA-ME, offering a turnkey solution to MUX configuration.

III. Background

A. CGRA-ME

CGRA-ME is an open-source framework for modeling and exploration of CGRAs [1]; it is overviewed in Fig. 1. CGRA-ME allows designers to describe a CGRA using an XML-based language or a C++ API. The specified CGRA is then transformed into a modulo routing resource graph (MRRG) [12] – a graph-based device representation. Benchmarks to map onto the CGRA are written in C and compute kernels from these are transformed into DFGs, using the LLVM compiler [13]. Given a MRRG device model and a DFG for an application, an integer linear programming-based (ILP) mapper determines the mapping feasibility of the benchmark onto the CGRA [14].

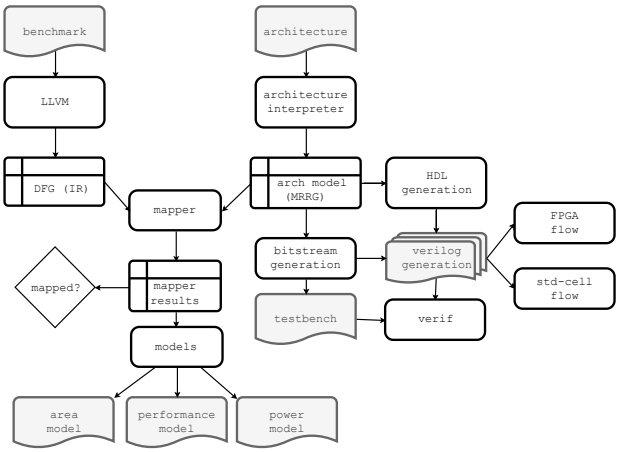

Fig. 1. Overview of CGRA-ME framework.

Along with feasibility of mapping benchmarks, CGRA-ME produces area and performance estimates for applications implemented on the modeled CGRA. Furthermore, Verilog HDL can be generated, allowing physical realization of the CGRA as a standard-cell ASIC or as an FPGA overlay. The latter FPGA flow is a main focus of this paper. We model two previously-published CGRA architectures, ADRES and HyCUBE, using CGRA-ME and discuss them in the following subsections.

B. ADRES

ADRES is a CGRA comprising a 2D array of processing elements (PEs), interconnect and memory [2], shown in Fig. 2. In Fig. 2(a), the top row of PEs connects to a multi-ported memory unit with 8 read ports and 4 write ports. The connectivity between PEs is N,S,E,W. Toroidal wrap-around connections are also present on the edges of the array (only some are shown for simplicity). The PE, depicted in Fig. 2(b) contains a functional unit (FU), a local register file (RF), an output register, and MUXes: bp, a b, and o. Inputs to MUXes bp, a, and b are from neighboring PEs, constant values, and potentially from memory. The purpose of the bp MUX is to allow the functional unit to be *bypassed* – in cases where the PE is used as a route-through. The FU can perform ALU-like OPs: add, subtract, multiply, logical shift left/right, arithmetic shift right, and logical OPs, including XOR, AND, and OR. The local register file in each PE has 2 outputs. A key takeaway regarding the interconnect is that PEs are *always* registered at the output; there is no strict combinational path through a PE. An brief investigation into modeling ADRES as an overlay using CGRA-ME appeared in [15], however, architecture optimizations were not explored.

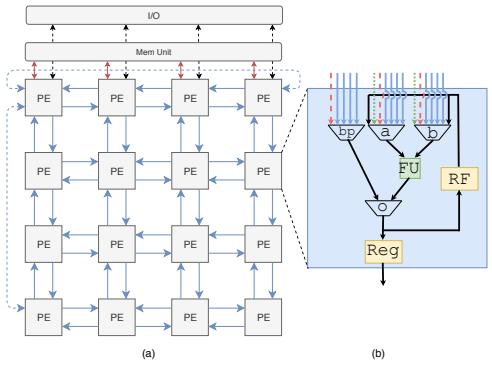

Fig. 2. (a) 4×4 array of ADRES PEs, (b) Enlarged view of PE, where the solid inputs are from neighboring PEs, long dashed input is optional from the memory unit, and the short dashed input is a constant input. Only toroid connection for one PE is shown.

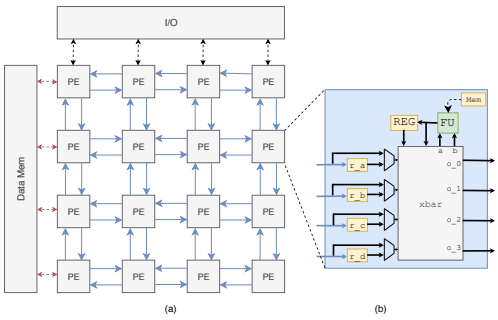

Fig. 3. (a) 4×4 array of HyCUBE PEs, (b) Enlarged view of PE, where inputs are from neighboring PEs.

C. HyCUBE

HyCUBE is a CGRA that supports single-cycle, multi-hop (combinational) communication between PEs [3], shown in Fig. 3. This has the benefit of allowing for a more flexible scheduler, specifically with regard to which OPs are intended to execute in different clock cycles. The PE consists of a crossbar switch, an FU, and configuration memory. At the input of the PE, the data has the ability to be registered or to bypass the register, and proceed into the crossbar switch, whereby it can be forwarded to a neighboring PE or to the FU for computation. We investigate how the single-cycle, multi-hop network compares to a more traditional CGRA (i.e., ADRES). The FU is identical to that in ADRES, described above.

IV. FPGA Architecture Optimizations

We now describe FPGA architecture-specific optimizations explored to improve the RTL produced by CGRA-ME. Optimizations deemed impactful: 1) Embedding the MUX select in SRAM configuration, 2) DSP utilization, 3) Multi-ported memory replication, and 4) Floorplanning. Optimizations have been integrated into CGRA-ME, along with other improvements for an enhanced FPGA-overlay flow.

Before outlining our optimizations, we first describe CGRA-ME's default behavior for generating Verilog. 1) MUXes produced by CGRA-ME are specified using a traditional Verilog case-statement. The back-end RTL synthesis tools then infers a MUX implementation using a network of LUTs. 2) FU functionality for CGRA-ME-generated Verilog consists of the standard operators: $+$, $-$, $*$, $\gg$, and so on. Again, the RTL synthesis is free to infer a hardware implementation for each OP. 3) For multi-ported memories, Verilog by CGRA-ME is very generic. On each clock cycle, data is read from all read ports from each corresponding read address. Each write port has a write-enable (WE) signal, and these are checked in specific order, implying port-level priority when writing to a specific address. No data-hazard checking is performed.

A. Optimizing Multiplexer Implementation

Both of the modeled CGRAs contain an abundance of MUXes. ADRES has 4 MUXes per PE, of varying sizes. The bypass MUX is 5:1 or 6:1, MUXes *a* and *b* are either 7:1 or 8:1, and the output MUX is 2:1. The first row of PEs contain 8:1 and 6:1 MUXes to communicate with memory, so a 4×4 ADRES CGRA will have eight 8:1, four 6:1 and four 2:1 MUXes along the first row. The remaining rows will have twenty-four 7:1, twelve 5:1 and twelve 2:1 MUXes combined, totalling 64 MUXes, either with 16- or 32-bit datapaths. HyCUBE also contains an abundance of MUXes, four 2:1 MUXes at the input of the crossbar switch, six 6:1 and two 2:1 within the crossbar switch itself, totalling 42 MUXes *per PE*. A 4×4 HyCUBE will therefore have 672 MUXes, again either with 16- or 32-bit datapaths. Because FPGAs are inefficient at implementing wide MUXes, and because CGRAs employ bus-based routing, we have identified them as a significant area/performance bottleneck and propose an optimization to improve their area and speed.

Fig. 4 illustrates the optimization concept. Fig. 4(a) shows a 6:1 MUX, having 3 select inputs (dashed) and 6 data inputs and Fig. 4(b) shows a traditional implementation that uses two 6-input LUTs. The three select inputs are exposed as inputs to the two 6-input LUTs. These select inputs would normally be attached to configuration cells, and the specific 0/1 values in the configuration cells would be set according to the CGRA mapping results. For example, if the CGRA mapping results indicate that input `i2` is to be passed to the MUX output, then the configuration cells would be set to `010` (2 in binary). Fig. 4(c) shows the optimized version of the LUT, which relies on our ability to directly modify the SRAM configuration cells

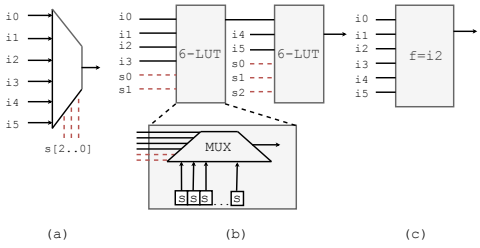

Fig. 4. (a) Traditional 6:1 MUX, (b) 6:1 MUX realized as a LUT, (c) 6:1 MUX with proposed technique. Signals that are dashed are embedded within SRAM configuration in (c).

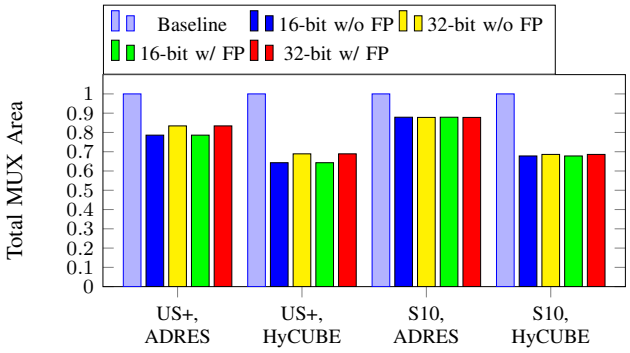

Fig. 5. Total MUX area normalized to unoptimized ADRES and HyCUBE variants for Stratix 10 and Ultrascale+ devices, with and without floorplanning.

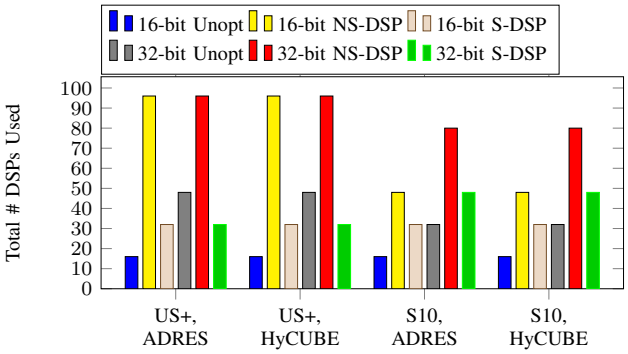

Fig. 6. Total Number of DSPs used for 16-bit and 32-bit ADRES and HyCUBE CGRAs. Three variants include the baseline (Unopt), DSPs used *without* DSP sharing (NS-DSP), and DSP usage *with* DSP sharing (S-DSP).

within a LUT. Based on the mapping results, we simply set the LUT's logic function `f` to equal `i2` (i.e. `f=i2`). A single 6-LUT is now required instead of two. This approach can also be extended to multi-context CGRAs by configuring the context number as an input into the LUTs used to realize the MUX. In so doing, the LUTs can select different inputs based on the context number.

We investigate this optimization by instantiating LUT primitives provided by Intel and Xilinx, and configuring the LUT mask accordingly. However, it can also be done by direct bitstream manipulation in a fully place-and-routed design, as long as the specific bits within the bitstream which contain the LUT truth-table values are known. This process is significantly faster than re-running the compilation flow.

Fig. 5 highlights the area savings of the multiplexer optimization for both 16-bit wide and 32-bit wide MUXes on both Intel and Xilinx FPGAs. We show a maximum area saving of 35.7% for HyCUBE with a 16-bit datapath on US+. In the worst case, the area saving due to our MUX optimization is 12.1% on ADRES, configured with a datapath of 32-bits on the S10. We can see that in general, HyCUBE is more amenable to MUX optimizations, due to the abundance of LUTs configured as MUXes, which is about 15% and 20% more than ADRES for US+ and S10, respectively.

B. Optimizing Functional Unit Implementation

DSPs are hardened logic blocks and are capable of efficient implementations of mathematical operations, such as add, subtract and multiply. They operate much faster than the equivalent soft-logic implementation, with a theoretical limit near ∼900MHz [16], [17]. We are interested in maximizing the use of the embedded DSP blocks in both S10 and US+ in order to optimize FU performance and reduce area of the utilized soft-logic. We are also interested in analyzing the effects of DSP sharing amongst the supported OPs on area and performance.

The US+ DSP contains the ability to perform many of the FU OPs supported by our overlays, including add, subtract, multiply, AND, OR, and XOR; these DSPs are unable to perform logical-shift left, logical-shift right or arithmetic-shift right [18]. We have configured the Xilinx DSPs to perform 6 out of 9 of our FU OPs. The S10 DSP has a more limited functionality when compared to its counterpart, having the ability to perform add, subtract and multiply [19]. For the S10, we configure 3 out of 9 FU OPs to use DSPs. When FU OPs are realized using DSPs, we consider two scenarios: unshared and shared. In the former case, we use a separate DSP for each OP; in the latter case, multiple OPs are realized on the same DSP, to the extent possible. In both FPGAs, the OPs not supported by the DSPs are performed using soft-logic. The baseline overlays perform all OPs in soft-logic, except for the multiply OP.

For the US+ DSP, we instantiate the DSP48E2 primitive in the CGRA Verilog. This primitive has four direct input ports, A, B, C, D and an output port P. It also contains runtime-configurable control inputs, OPMODE, to control the data passed to the ALU and ALUMODE to control the ALU OP, which is useful for multi-context CGRAs. Along with the 48-bit C-port, we can concatenate the 30-bit A-port and 18-bit B-port, A:B to form another 48-bit port for wide OPs. Appropriate configuration of OPMODE and the ALUMODE settings is also required for internal data movement. The multiply OP supports a 27×18-bit OP, and the add/subtract OPs and logic unit operations can support 48-bit OPs.

The S10 DSP contains four 18-bit and two 19-bit wide datapath inputs. Internally, the DSP contains a pre-adder, multiplier, add/subtract unit and an accumulator. It supports a single 27×27 multiply (fracturable to accommodate two 18×19 multiplies) that cannot be bypassed, and a 38-bit add/subtract after the multipliers. To support add/subtract OPs, we multiply the input signal by 1 to maintain the original input, before adding/subtracting the two values produced by the multipliers. For a 32-bit datapath, two DSPs are required per add/subtract OP, with carry functionality between the units, since each DSP can only add two values of up to 18-bits.

Fig. 6 shows a comparison of DSPs used before and after optimization. For the US+ designs, we see a 6× and a 2× DSP increase when we opt for no DSP sharing for 16-bit and 32-bit designs. We also see a 2× increase and a 1.5× decrease in DSP usage when we use DSP sharing. The reason we do not see a larger increase in DSPs used when no DSP

sharing is performed and a decrease when DSP sharing is performed for our 32-bit design is because we realize our multiply with a single DSP, while the baseline uses 3 DSPs. For the non-shared DSP this equates to 6 DSPs per FU for all DSP-supported OPs, whereas the baseline uses 3 DSPs per FU. Additionally, when we share DSPs, we perform a multiply in its own DSP for reduced configuration complexity, resulting in 2 DSPs per FU for all DSP-supported OPs, and again, 3 DSPs in the baseline. The naive 16-bit design uses a single DSP per FU in both shared-DSP and non-shared DSP variants, resulting in 6× increase when DSPs are not shared (1 DSP per DSP-supported OP), and a 2× increase in DSPs when we support DSP sharing.

The optimized S10 overlays use 3× and 2.5× more DSPs for 16-bit and 32-bit datapaths without DSP sharing. With DSP sharing, however see a 2× and 1.5× increase in DSPs used for 16- and 32-bit datapaths. In both cases, the increase is larger than the baseline, and in some cases the US+ variants, even though less OPs are supported, for two reasons. The first reason is that we need 2 DSPs per add or subtract OP, whereas US+ only needs one for these OPs. Secondly, as with the US+, the multiply OP occupies its own DSP.

Comparing the DSP utilization of these FPGAs, we can see that US+ uses 2× and 1.2× more DSPs than S10, for the 16-bit and 32-bit datapaths, respectively in the non-shared DSP variant. When DSP-sharing is allowed, we see the same DSP usage at 16-bit (32 DSPs) and a 1.5x decrease at 32-bit due to US+ only requiring one DSP per add or subtract OP. These ratios reflect the US+ DSPs flexibility and capabilities to implement a wide array of OPs, using a minimal amount of hardened logic.

Overall, the main reason that *more* DSPs are used in the optimized variant when compared to the unoptimized is that more logic is subsumed into the DSP blocks in the new designs. This provides higher speed and a soft-logic area reduction, as shown in the experimental study below.

C. Optimizing Multi-Ported Memories

Our ADRES variant contains a memory with 8 read and 4 write ports. Commercial FPGA vendors only provide dual-port RAM, leading to poor performance for systems requiring many read or writes, since there is contention for the limited number of RAM ports. To realize more ports, potential techniques include banking, replication and multi-pumping. In this

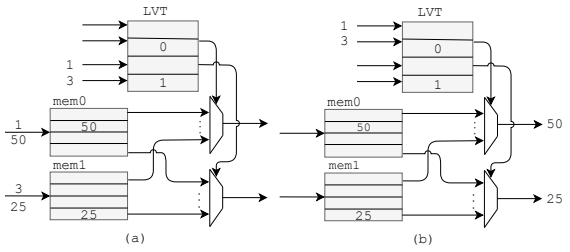

Fig. 7. (a) LVT multi-ported memory during write, (b) LVT multi-ported memory during read.

TABLE I
MULTIPORTED-MEMORY LOGIC USAGE FOR 16-BIT AND 32-BIT ADRES AND ON STRATIX 10 AND ULTRASCALE+.

	US+				S10			
	UNOPT		OPT		UNOPT		OPT	
	16b	32b	16b	32b	16b	32b	16b	32b
LUT	2340	4602	556	683	2024	3816	520	648
FF	640	1280	592	1104	640	1280	1104	2128
LUTRAM	0	0	512	768	0	0	320	640
ALM	–	–	–	–	1940	3712	923	1557

work, we use the live-value-table (LVT) approach to realize multi-ported memory [20]. The LVT approach is essentially replication with bookkeeping.

To illustrate the LVT multi-ported memory concept, consider the example in Fig. 7, for a 2-read/2-write-port RAM. Observe there are two memory banks. Part (a) shows a write of two values: one to each bank. A value of 50 is stored at address 1 in bank 0; 25 is stored at address 3 in bank 1. In a small scratchpad memory, called the LVT (top of figure), a 0 is stored at address 1, and a 1 is stored at address 3, representing the *bank index* of the *most recent* write to the corresponding address. Part (b) of the figure illustrates read behavior: the LVT values are used as the select inputs of multiplexers to retrieve the correct written value for an address. The original work implemented the LVT bookkeeping with registers, whereas we have modified it to implement the LVT in LUTRAM. We refer the readers to [20] for further details on the implementation of this multi-ported memory.

Table I highlights the number of LUTs, FFs, and LUTRAMs used in LVT implementation for our ADRES variant, using a 32-word memory. Without optimizations, we observe a comparable amount of logic usage between S10 and US+, where the S10 uses slightly less LUT logic. After our optimzations have been implemented, we still see a comparable usage in LUT logic and LUTRAM, however the S10 now uses ∼2× more FFs than before. The US+ has the ability to configure multiple LUTs for memory up to 512 bits (called SLICEM) and combine multiple SLICEMs for even larger memories, with the ability for data/address sharing. The S10 memory-LAB (MLAB) can be configured to store up to 640 bits. For S10 designs, MLABs usage is converted into the equivalent LUTRAMs, where 1 MLAB=10 LUTRAMs.

We elaborate on the optimized memory usage for US+ as follows: For a 32-bit datapath, we require 24 LUTRAMs per read-port (8) per memory bank (4), which consumes ($24 \times 8 \times 4$) 768 LUTRAMs. There are also 1024 FFs ($32 \times 8 \times 4$) for the memory bank logic. For our 16-bit variant, we have 16 LUTRAMs per read-port (8) per memory bank (4), using ($16 \times 8 \times 4$) 512 LUTRAMs. At this bitwidth, we also have ($16 \times 8 \times 4$) 512 FFs. LUT logic for both 16-bit and 32-bit datapaths comes from the MUXes at the read ports and from the LVT output. Additionally, for both bitwidths, the reported number of FFs is higher than calculated, used to implement the LVT.

An MLAB contains 640 bits, implying that at depth 32,

we have a maximum word width of 20-bits. In order to accommodate a 32-bit datapath, we need 2 MLABs, or 20 ALMs, to realize a dual-port SRAM. The 32-bit datapath then requires 20 ALMs per read port (8) per write-bank (4), so we consume ($20 \times 8 \times 4$) 640 LUTRAMs. The 16-bit datapath is similar, except we use 10 ALMs for this bitwidth, and therefore only need ($10 \times 8 \times 4$) 320 LUTRAMs. We see the increase in FFs because for every 10 ALMs, we consume 32 FFs, which gives us ($64 \times 4 \times 8$) 2048 FFs in the 32-bit case and ($32 \times 4 \times 8$) 1024 FFs in the 16-bit case, with the remaining FFs being used in the LVT. The increase in LUTs is also due to the LVT and the MUX logic at the output of the multi-port RAM.

D. Floorplanning

We applied floorplanning to ensure that the physical layout matches the logical layout of tiles, and also to improve packing, performance and routability. This is a manual process and can be tedious to arrive at the desired configuration. Techniques outlined in [21] served as a guide for our floorplanning methodology. We partitioned the physical regions of the floorplan into rectangles (calculated based on number of logic cells), occupying 85-90% of the logic consumed by a PE and spaced them in such a way that each PE was not overlapping. As we will see in the next section, floorplanning improved the F_{max} of the implemented overlays.

TABLE II
BENCHMARK STATISTICS, INCLUDING MAPPING RUNTIME FOR ADRES AND HYCUBE.

Benchmarks	OPs	Mults	I/O	ADRES (s)	HyCUBE (s)
conv2	16	5	3	31.21	1599.07
conv3	24	7	4	34.68	–
mac	11	3	3	100.23	1092.97
multS1	31	8	5	816.81	6796.99
nomem1	6	1	1	47.31	651.94
sum	7	1	2	44.20	42.88
simple	12	3	3	76.08	963.36
simple2	12	3	3	75.81	957.89

V. EXPERIMENTAL SETUP AND RESULTS

We study the following CGRA overlay implementations:

1) ADRES and HyCUBE.
2) Intel Stratix 10 and Xilinx Ultrascale+ devices.
3) Unoptimized and architecturally-optimized CGRAs.
4) 16- and 32-bit-wide datapaths.
5) Non-floorplanned and floorplanned designs.

With the above parameters, we have a total of 32 different CGRA overlay configurations. The CGRAs are 4×4 arrays and single context. We set a target F_{max} of 1GHz, and reported F_{max} results are post-place/route, generated using the static-timing analysis tools that are included in Intel Quartus and Xilinx Vivado. We synthesize our designs on the Stratix 10 GX850 and the Ultrascale+ XCVU3P, and both Intel and Xilinx tool versions used are 18.1.

A. Benchmarks

Information about the benchmarks is given in Table II. The column 'OPs' is the number of OPs for each benchmark (consists of load, store, FU OP, etc.), the 'Mults' column is the number of multiplications performed in a benchmark and 'I/O' is the number of inputs and outputs for the benchmark. We highlight 'Mults' because this OP usually resides on the critical path. We also provide the mapping runtime for both ADRES and HyCUBE. We note that HyCUBE was unable to map benchmark 'conv3' because the required number of ports to the memory unit is greater than is offered by this architecture.

Benchmarks were first mapped using CGRA-ME in order to determine mapping feasibility. Using the mapping results (PE configuration, routing paths, etc.), we use an in-house tool to configure the overlay LUT masks and to generate the constraints for STA. In order to assess the critical paths of each benchmark, we use the mapping results to direct STA to focus solely on the *used* part of the overlay. This is done through 'set_false_path' and 'set_disable_timing' SDC constraints provided to the STA engine. By forcing the STA engine to ignore all unused paths in a CGRA overlay, we are able to find the critical path in the portion of the overlay used by a particular application mapping.

Fig. 8 showcases average F_{max} across all benchmarks for all overlay implementation scenarios, elaborated upon in the next section.

B. Xilinx Ultrascale+

Table III reports resources consumed by ADRES, for the unoptimized and optimized scenarios, configured at 16-bit and 32-bit. Table IV also shows similar results for the HyCUBE CGRA architecture. Analysis of Table III reveals that optimizations allow us to achieve an average of 41.9% and 41.8% savings in LUT usage, as well as savings of 28.4% and 31.8% in LUTs/PE, for 16-bit and 32-bit, respectively for our non-

TABLE III
RESOURCE CONSUMPTION FOR 16-BIT (TOP) AND 32-BIT (BOTTOM) DATAPATH ADRES CGRA USING ULTRASCALE+. FP DENOTES FLOORPLANNING, NS-DSP DENOTES NO DSP SHARING, AND S-DSP DENOTES DSP SHARING.

16b	Unopt w/o FP	Unopt w/ FP	Opt NS-DSP w/o FP	Opt NS-DSP w/ FP	Opt S-DSP w/o FP	Opt S-DSP w/ FP
LUT	8770	8659	5075	5041	4792	4770
FF	2488	2488	2520	2520	2456	2456
DSP	16	16	96	96	32	32
LUTRAM	32	32	544	544	544	544
AVG LUT/PE	369	362	262	261	230	228

32b	Unopt w/o FP	Unopt w/ FP	Opt NS-DSP w/o FP	Opt NS-DSP w/ FP	Opt S-DSP w/o FP	Opt S-DSP w/ FP
LUT	18565	18416	10600	10555	9513	9509
FF	4600	4600	4731	4731	4696	4696
DSP	48	48	96	96	32	32
LUTRAM	64	64	832	832	832	832
AVG LUT/PE	809	801	552	546	484	480

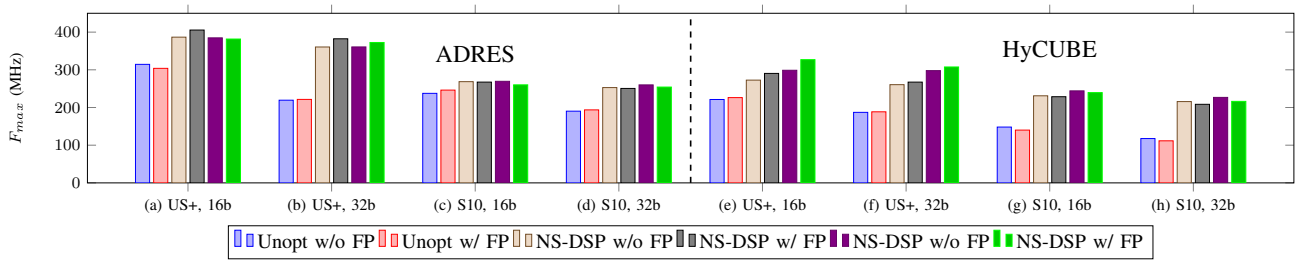

Fig. 8. Average operating frequency for ADRES and HyCUBE CGRAs at 16-bit and 32-bit, for unoptimized designs (Unopt), non-shared DSP designs (NS-DSP), and shared-DSP designs (S-DSP).

shared DSP designs, despite seeing an increase in LUTRAMS and FFs. Additionally, we see an average of 45.1% and 48.5% savings in LUTs and an average savings of 37.4% and 40.1% LUTs/PE for 16-bit and 32-bit designs, with DSP sharing. This increase in area savings highlights the importance of DSPs and the effect of minimizing the amount of soft logic, where applicable. We see an increase in LUTRAMs and FFs due to the need for memory as LUTRAMS and registers for our LVT implementation.

Performance of ADRES can be seen in Fig. 8(a) and Fig. 8(b). With our optimizations, we are able to achieve an average of 1.28× and 1.68× $Fmax$ improvement over the baseline, at 16-bit and 32-bit, respectively for our non-shared DSP variant. When we perform DSP sharing, we see an average of 1.24× and 1.66× $Fmax$ improvement over the baseline, for 16-bit and 32-bit designs. Although soft logic is saved with a shared-DSP variant, ADRES does not see a larger improvement in performance due to the path between registers remaining relatively consistent between the two designs. We also see a much larger speedup for the 32-bit design, largely due to the impact that the naive usage of DSPs have, accounting for ∼50% of the logic delay between the two implementations, where MUX logic and interconnect delay accounts for the remainder. The 16-bit design does not display the same magnitude of speedup compared to the 32-bit design because the naive implementation for DSP multiply compared to the optimized variant is the same, meaning that the performance improvement is solely due to the MUX optimizations.

Table IV outlines HyCUBE resource utilization for both unoptimized and optimized designs. For the non-shared DSP case, HyCUBE uses 15.4% and 19.6% more LUTs than ADRES at 16-bit and 32-bit designs, and 6.84% and 17.2% more LUTs than ADRES in the shared-DSP case, at 16-bit and 32-bit. The crossbar has a large impact on area which increases the overall LUT usage. This can also be seen by investigating the average LUTs/PE when comparing HyCUBE to ADRES. For the non-shared DSP variant, we see an average of 1.25× and 1.4× speedup over the baseline for 16-bit and 32-bit datapaths, respectively, as seen in Fig. 8(e) and Fig. 8(f). From the same figure we see a speedup of 1.40× and 1.61× for 16-bit and 32-bit for the shared-DSP designs. This highlights the impact that DSP-sharing (which reduces the overall logic dramatically) has on architectures that suffer from long combinational paths, like HyCUBE. Architectures that suffer from longer combinational paths between registers appear to be more amenable to DSP-sharing.

Since our shared variant is configured with an initiation interval (II) of 1, we extrapolate that with an II greater than 1, the $Fmax$ of the shared-DSP variant will degrade noticeably. This conclusion is based on limitations in DSP run-time configuration [18] and is left for future work.

C. Intel Stratix 10

Resources used by ADRES for both unoptimized and optimized variants are reported in Table V. We show area savings of 30.9% and 30.5% as well as 25.0% and 24.3% ALMs/PE for 16-bit and 32-bit datapath designs, respectively for our non-shared DSP variants. Our shared-DSP variants display area savings of 31.1% and 30.6% for 16-bit and 32-bit, respectively. Similar to the results for US+, the increase in LUTRAMs is due to the LVT multi-port memory. We see a more modest speedup relative to the baseline with an average of 1.10× and 1.31× improvement in F_{max} for 16-bit and 32-bit designs, respectively, seen in Fig. 8(c) and Fig. 8(d) for our non-shared DSP variant. The shared-DSP variants display a similar speedup as the non-shared DSP variant, with an average of 1.09× and 1.33× improvement in F_{max} for 16-bit and 32-bit designs. Similar to the US+ ADRES implementation, the negligible difference in speedup can be attributed to the

TABLE IV
RESOURCE CONSUMPTION FOR 16-BIT (TOP) AND 32-BIT (BOTTOM) DATAPATH HYCUBE CGRA USING ULTRASCALE+. FP DENOTES FLOORPLANNING, NS-DSP DENOTES NO DSP SHARING, AND S-DSP DENOTES DSP SHARING.

16b	**Unopt w/o FP**	**Unopt w/ FP**	**Opt NS-DSP w/o FP**	**Opt NS-DSP w/ FP**	**Opt S-DSP w/o FP**	**Opt S-DSP w/ FP**
LUT	8308	8279	5879	5800	5108	5107
FF	2976	2976	2513	2513	2512	2512
DSP	16	16	48	48	32	32
LUTRAM	32	32	32	32	32	32
AVG LUT/PE	514	511	371	353	323	310

32b	**Unopt w/o FP**	**Unopt w/ FP**	**Opt NS-DSP w/o FP**	**Opt NS-DSP w/ FP**	**Opt S-DSP w/o FP**	**Opt S-DSP w/ FP**
LUT	17770	17673	12758	12539	11230	11069
FF	5472	5472	5009	5009	5008	5008
DSP	48	48	96	96	32	32
LUTRAM	64	64	64	64	64	64
AVG LUT/PE	1104	1032	802	767	705	674

register-to-register path being consistent for both variants. The speedup difference between the 16-bit and 32-bit datapath designs is also largely due to the naive FU implementation, where the DSP utilization is sub-optimal.

We also demonstrate area savings of 27.6% and 23.1% as well as 29.0% and 24.3% ALMs/PE reduction for 16-bit and 32-bit designs, respectively in Table VI for the HyCUBE non-shared DSP variant. Our DSP-sharing gives us an area savings of 27.8% and 23.4% as well as 29.0% and 24.4% ALMs/PE reduction for 16-bit and 32-bit designs, respectively. Fig. 8(g) and (h) shows a performance improvement of 1.59× and 1.85× in the 16-bit case and 32-bit case, respectively for no-DSP sharing. With DSP-sharing, we show a performance improvement of 1.68× and 1.93× in the 16-bit case and 32-bit case, demonstrating that architectures with more combinational logic are more amenable to DSP-sharing, resulting in higher $Fmax$. Prior to optimzation, we had to create design partitions for *every* module to allow the STA engine to be able to disable timing across the desired paths, which adds area overhead and more interconnect/logic delay.

D. Intel vs. Xilinx

We believe that the US+ is better suited to realize CGRAs as FPGA-overlays – at least for the ADRES and HyCUBE CGRAs considered here. With each of our optimizations, we see that Xilinx FPGAs are more capable of being area optimized, using less LUTs as MUXes and DSPs *per* OP. Intel FPGAs seem better equiped to handle logic for multi-ported memory, as the configuration of a MLAB is more optimized when compared to the Xilinx FPGAs (620 bits vs 512), although we noted that memory performance is not the bottleneck of our overlays. We believe this gap could be reduced if S10 had better DSP capabilities and the ability to handle wider OPs.

TABLE V
RESOURCE CONSUMPTION FOR 16-BIT (TOP) AND 32-BIT (BOTTOM) ADRES CGRA USING STRATIX 10. FP DENOTES FLOORPLANNING, NS-DSP DENOTES NO DSP SHARING, AND S-DSP DENOTES DSP SHARING.

16b	Unopt w/o FP	Unopt w/ FP	Opt NS-DSP w/o FP	Opt NS-DSP w/ FP	Opt S-DSP w/o FP	Opt S-DSP w/ FP
ALM	7504	7410	5239	5059	5221	5051
LUT	8144	8144	5276	5276	5276	5276
DSP	16	16	48	48	32	32
LUTRAM	40	40	360	360	360	360
AVG ALM/PE	298	294	228	216	228	216
AVG LUT/PE	336	336	251	251	251	251

32b	Unopt w/o FP	Unopt w/ FP	Opt NS-DSP w/o FP	Opt NS-DSP w/ FP	Opt S-DSP w/o FP	Opt S-DSP w/ FP
ALM	15495	15145	10700	10592	10674	10585
LUT	17373	17373	10808	10808	10808	10808
DSP	32	32	80	80	48	48
LUTRAM	80	80	720	720	720	720
AVG ALM/PE	654	637	492	485	492	485
AVG LUT/PE	772	772	551	551	551	551

TABLE VI
RESOURCE CONSUMPTION FOR 16-BIT (TOP) AND 32-BIT (BOTTOM) HYCUBE CGRA USING STRATIX 10. FP DENOTES FLOORPLANNING, NS-DSP DENOTES NO DSP SHARING, AND S-DSP DENOTES DSP SHARING.

16b	Unopt w/o FP	Unopt w/ FP	Opt NS-DSP w/o FP	Opt NS-DSP w/ FP	Opt S-DSP w/o FP	Opt S-DSP w/ FP
ALM	8165	8159	6140	5673	6128	5664
LUT	8536	8536	5736	5736	5736	5736
DSP	16	16	48	48	32	32
LUTRAM	40	40	40	40	40	40
AVG ALM/PE	486	486	359	330	359	330
AVG LUT/PE	525	522	349	349	349	349

32b	Unopt w/o FP	Unopt w/ FP	Opt NS-DSP w/o FP	Opt NS-DSP w/ FP	Opt S-DSP w/o FP	Opt S-DSP w/ FP
ALM	16951	16435	13030	12639	13006	12582
LUT	18680	18680	12229	12229	12229	12229
DSP	32	32	80	80	32	32
LUTRAM	80	80	80	80	40	40
AVG ALM/PE	1012	980	765	742	765	739
AVG LUT/PE	1150	1150	746	746	746	746

Consistent with both FPGAs, we see that ADRES has a better F_{max} than HyCUBE. When we compare them, we see a performance decrease of about 30% for the Ultrascale+ and 50% for the Stratix 10 on HyCUBE. This is due to the multi-hop interconnect style, and our mapping tool not accounting for these multi-hop paths, whereby it might realize a configuration that utilizes a longer, less optimal path. Additionally, in order to preserve the netlist names (for STA), Quartus requires design partitions for each module in the design hierarchy. If design partitions are not made, inputs to and outputs from a logic module (or port) will be converted into Quartus netlist notation. When this occurs, Quartus will not be able to recognize the names of the ports that we have set in our constraints that disables or sets as false paths. By creating design partitions, logic overhead is added to the design which affects the critical path. This is not an issue with Vivado, as the CAD tool maintains netlists with little overhead using the 'keep_hierarchy' synthesis attribute. Despite the decrease in performance, we believe that with tweaks to our mapping tool, we can realize more optimal configurations of HyCUBE.

VI. CONCLUSION

Presented here are two CGRA architectures, ADRES and HyCUBE, configured as overlays on two commercial FPGAs, Intel Stratix 10 and Xilinx Ultrascale+. We first synthesize the architectures naively, using CGRA-ME and determine the bottlenecks. We make flexible and architecture specific optimizations, utilizing hardened FPGA logic targetting area and performance. We then map various benchmarks on naive and optimized CGRAs, to determine the overall performance of each CGRA synthesized to each FPGA architecture, and discuss which FPGA is more amenable to these architecture changes.

REFERENCES

[1] 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 *IEEE ASAP*, pp. 184–189, 2017.

[2] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix," in *IEEE FPL*, pp. 61–70, 2003.

[3] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "HyCube: A CGRA with Reconfigurable Single-cycle Multi-hop Interconnect," in *ACM/EDAC/IEEE DAC*, pp. 1–6, 2017.

[4] S. Shukla, N. W. Bergmann, and J. Becker, "QUKU: A Two-Level Reconfigurable Architecture," in *IEEE ISVLSI*, pp. 6 pp.–, 2006.

[5] A. K. Jain, X. Li, S. A. Fahmy, and D. L. Maskell, "Adapting the DySER Architecture with DSP Blocks as an Overlay for the Xilinx Zynq," *SIGARCH Comput. Archit. News*, vol. 43, pp. 28–33, Apr. 2016.

[6] A. K. Jain, X. Li, P. Singhai, D. L. Maskell, and S. A. Fahmy, "DeCO: A DSP Block Based FPGA Accelerator Overlay with Low Overhead Interconnect," in *IEEE FCCM*, pp. 1–8, 2016.

[7] A. K. Jain, S. A. Fahmy, and D. L. Maskell, "Efficient Overlay Architecture Based on DSP Blocks," in *IEEE FCCM*, pp. 25–28, 2015.

[8] N. W. Bergmann, S. K. Shukla, and J. Becker, "QUKU: A Dual-layer Reconfigurable Architecture," *ACM Trans. Embed. Comput. Syst.*, vol. 12, pp. 63:1–63:26, Mar. 2013.

[9] V. Govindaraju, C. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing," *IEEE Micro*, pp. 38–51, 2012.

[10] Z. Poulos, Y.-S. Yang, J. Anderson, A. Veneris, and B. Le, "Leveraging Reconfigurability to Raise Productivity in FPGA Functional Debug," in *EDAC DATE*, pp. 292–295, 2012.

[11] A. Brant and G. G. Lemieux, "ZUMA: An Open FPGA Overlay Architecture," in *IEEE FCCM*, pp. 93–96, 2012.

[12] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures," in *IEEE FPT*, pp. 166–173, 2002.

[13] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *ACM/IEEE CGO*, p. 75, 2004.

[14] S. A. Chin and J. H. Anderson, "An Architecture-Agnostic Integer Linear Programming Approach to CGRA Mapping," in *ACM/ESDA/IEEE DAC*, pp. 1–6, 2018.

[15] S. A. Chin, K. P. Niu, M. Walker, S. Yin, A. Mertens, J. Lee, and J. H. Anderson, "Architecture Exploration of Standard-Cell and FPGA-Overlay CGRAs Using the Open-Source CGRA-ME Framework," in *ACM ISPD*, pp. 48–55, 2018.

[16] Xilinx, "Virtex UltraScale+ FPGA Data Sheet: DC and AC Switching Characteristics," Jan 2019.

[17] Intel, "Intel Stratix 10 Device Datasheet," Oct 2018.

[18] Xilinx, "UltraScale Architecture DSP Slice," Apr 2018.

[19] Intel, "Intel Stratix 10 Variable Precision DSP Blocks User Guide," Sep 2018.

[20] C. E. LaForest and J. G. Steffan, "Efficient Multi-Ported Memories for FPGAs," in *ACM FPGA*, pp. 41–50, 2010.

[21] D. Capalija and T. S. Abdelrahman, "Tile-based Bottom-up Compilation of Custom Mesh-of-Functional-Units FPGA Overlays," in *IEEE FPL*, pp. 1–8, 2014.