REVAMP: A Systematic Framework for Heterogeneous CGRA Realization

Thilini Kaushalya Bandara
thilini@comp.nus.edu.sg
National University of Singapore
Singapore

Dhananjaya Wijerathne
dmd@comp.nus.edu.sg
National University of Singapore
Singapore

Tulika Mitra
tulika@comp.nus.edu.sg
National University of Singapore
Singapore

Li-Shiuan Peh
peh@comp.nus.edu.sg
National University of Singapore
Singapore

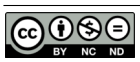
ABSTRACT

Coarse-Grained Reconfigurable Architectures (CGRAs) provide an excellent balance between performance, energy efficiency, and flexibility. However, increasingly sophisticated applications, especially on the edge devices, demand even better energy efficiency for longer battery life.

Most CGRAs adhere to a canonical structure where a homogeneous set of processing elements and memories communicate through a regular interconnect due to the simplicity of the design. Unfortunately, the homogeneity leads to substantial idle resources while mapping irregular applications and creates inefficiency. We plan to mitigate the inefficiency by systematically and judiciously introducing heterogeneity in CGRAs in tandem with appropriate compiler support.

We propose *REVAMP*, an automated design space exploration framework that helps architects uncover and add pertinent heterogeneity to a diverse range of originally homogeneous CGRAs when fed with a suite of target applications. *REVAMP* explores a comprehensive set of optimizations encompassing compute, network, and memory heterogeneity, thereby converting a uniform CGRA into a more irregular architecture with improved energy efficiency. As CGRAs are inherently software scheduled, any micro-architectural optimizations need to be partnered with corresponding compiler support, which is challenging with heterogeneity. The *REVAMP* framework extends compiler support for efficient mapping of loop kernels on the derived heterogeneous CGRA architectures.

We showcase *REVAMP* on three state-of-the-art homogeneous CGRAs, demonstrating how *REVAMP* derives a heterogeneous variant of each homogeneous architecture, with its corresponding compiler optimizations. Our results show that the derived heterogeneous architectures achieve up to 52.4% power reduction, 38.1% area reduction, and 36% average energy reduction over the corresponding homogeneous versions with minimal performance impact for the selected kernel suite.



ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02.
https://doi.org/10.1145/3503222.3507772

CCS CONCEPTS

• **Computer systems organization** → *Multicore architectures*; **Reconfigurable computing**; **Data flow architectures**.

KEYWORDS

Coarse Grained Reconfigurable Arrays (CGRAs), Heterogeneous CGRAs, CGRA design space exploration

ACM Reference Format:
Thilini Kaushalya Bandara, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. 2022. REVAMP: A Systematic Framework for Heterogeneous CGRA Realization. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3503222.3507772

1 INTRODUCTION

Resource-constrained edge devices rely on accelerators to deliver high performance at low power. Application Specific Integrated Circuit (ASIC) accelerators deliver the best power-performance, but are inflexible, leading to higher non-recurring engineering cost. Field Programmable Gate Arrays (FPGAs) are a popular alternative as reconfigurable accelerators. But bit-level reconfigurability in FPGAs leads to lower energy efficiency. Coarse-Grained Reconfigurable Architectures (CGRA) [24, 30] strike a balance between flexibility and efficiency by introducing word-level and per-cycle reconfigurability, making them ideal for acceleration at the edge.

Several academic CGRA architectures have been proposed in recent years, such as ADRES [26], Morphosys [36], HyCUBE [19, 38], and commercial ones like Samsung Reconfigurable Processor (SRP) [21], Wave DPU [27], DRP [13] and Plasticine [32]. Most existing CGRAs (e.g., SRP [21], Morphosys [36], ADRES [26]) adopt a regular structure where identical PEs and memory are connected with a uniform network. Designers have largely adhered to homogeneous CGRAs due to the lower hardware design effort, as well as the complexities of a software compiler when mapping onto irregular hardware. Such homogeneity, however, leads to idle resources as well as area and power inefficiencies. We thus see the need to introduce heterogeneity through automated design space exploration frameworks that can assist architects in delivering better power-performance, paving the way for wider adoption of CGRAs.

We propose *REVAMP* (Figure 1), a novel design space exploration framework comprising a set of micro-architectural optimizations

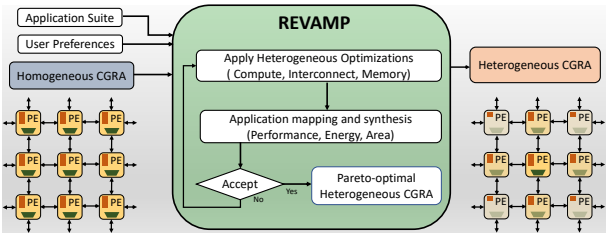


Figure 1: High-level overview of *REVAMP*

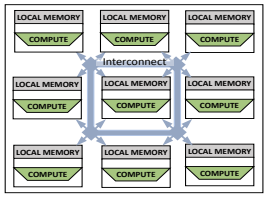


Figure 2: Homogeneous CGRA

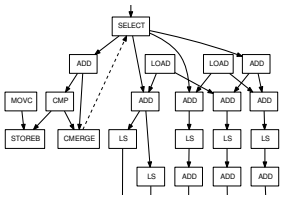


Figure 3: Snippet of DFG

and corresponding compiler support for *heterogeneous* CGRA realization for a suite of applications starting with a homogeneous architecture. *REVAMP* offers a trade-off between generalization and specialization by creating heterogeneous architecture for a specific domain with diverse kernels (e.g., edge computing for healthcare). Moreover, we are targeting CGRAs for acceleration at the edge amenable to domain- and product-class-specific specialization. Hence, there is generally prior knowledge of the workloads that *REVAMP* can use to derive a heterogeneous CGRA specialized across target workloads.

In contrast to prior heterogeneous CGRA architectures with localized heterogeneous optimizations, *REVAMP* introduces novelty to the heterogeneous CGRA design process by being generic, configurable, and scalable. *REVAMP* enables architects to explore heterogeneous optimizations for diverse homogeneous CGRAs and not a specific CGRA. We also introduce heterogeneity to a wider scope, considering compute, interconnect, and memory. Finally, we design the compiler to support the introduced heterogeneous features. Our concrete contributions are:

- We propose a novel framework *REVAMP* for architects, comprising a set of optimizations to automatically derive a heterogeneous CGRA with higher efficiency given any homogeneous architecture and an application suite. The framework is publicly available at [4]
- Our optimizations cover a wider scope of heterogeneity including compute, interconnect, and PE-local storage.
- We develop compiler optimizations to support near-optimal mapping on the derived heterogeneous architectures.
- We showcase *REVAMP* by deriving heterogeneous architectures from three prominent homogeneous CGRAs. The heterogeneous CGRAs provide average of 38.5% power and 29.8% area reduction (highest 52.4% and 38.1%) compared to the homogeneous counterparts. Application kernel execution on heterogeneous CGRAs shows average of 36% energy reduction with minimal impact on performance compared to the homogeneous versions, resulting in a 62% average increase in energy efficiency for evaluated kernel suite.

2 INEFFICIENCY OF HOMOGENEOUS CGRAS

We analyze several prominent homogeneous CGRA architectures to highlight their inefficiencies and motivate the need for the proposed heterogeneous optimizations.

2.1 Homogeneous CGRA

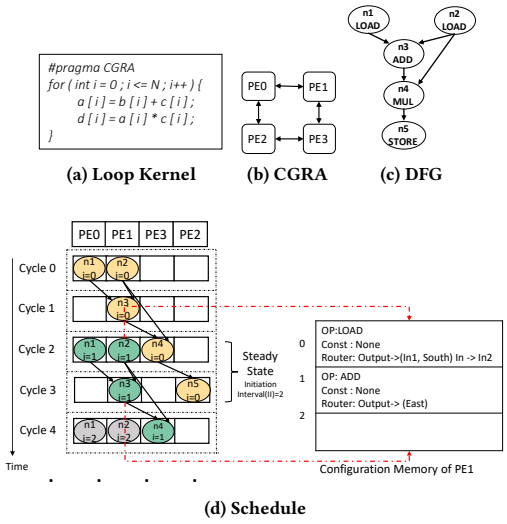
Figure 2 shows a canonical homogeneous CGRA. The CGRA consists of a uniform array of processing elements (PE), each with compute and memory elements (for configuration and data), interconnected with a network.

As CGRAs are fully software scheduled, the compiler plays a vital role. It extracts the compute-intensive loop kernel as a Data Flow Graph (DFG) (Figure 3), where the nodes represent the operations and the edges represent the data dependencies. The compiler is responsible for spatio-temporal mapping of the DFG onto the CGRA with both inter- and intra-loop iteration parallelism. The schedule gets repeated for each loop iteration, hence called a modulo schedule. The number of cycles between two successive iterations is known as the Initiation Interval (II), which is used as the performance metric. The lower the II value, the higher is the throughput of the loop execution. A compiler-generated CGRA modulo schedule contains hardware operations and the data routing details for each cycle. Prior to the execution, a binary representation of the modulo schedule is loaded to the configuration memory of the PEs. Input data is loaded to the on-chip data memory. During execution, each PE reconfigures compute and routing according to the configuration memory every cycle and repeats after II cycles. Figure 4 illustrates an example of such scheduling.

2.2 Homogeneity Leads to Inefficiency

Homogeneous CGRAs uniformly distribute the on-chip resources and provide full flexibility to the compiler. Identical resource distribution eases compilation. It also simplifies the placement and layout during chip design. Unfortunately in reality, the compiler may not be able to fully utilize the resources for certain applications, incurring unnecessary area, power overheads. CGRAs like SRP [21] introduce different power modes and clustered PEs to reduce this overhead by gating selected clusters. But complex power managements are at odds with resource-constrained devices and do not offer additional choices and fine-grained control like heterogeneity.

Compute : In a homogeneous CGRA, the PEs have identical hardware (e.g., ALU) supporting time-multiplexing of different operations. However, for most applications, only a subset of the operations are used throughout execution for each PE and the hardware for the remaining operations remain idle. In addition, a fully flexible compute unit requires more configuration bits, of which only a few carry useful reconfiguration information. On the other hand, the data dependencies in DFGs are inherently irregular (Figure 3), restricting the amount of parallelism that can be achieved and impacting resource utilization.



(a) Loop Kernel (b) CGRA (c) DFG

(d) Schedule

Figure 4: An example of CGRA scheduling. DFG (Figure 4c) of loop kernel (Figure 4a) is scheduled (Figure 4d) on to a $2x2$ CGRA (Figure 4b). Prior to the execution, the schedule is loaded to the configuration memory.

Interconnect : The interconnect delivers data from the source PE to the destination PE corresponding to each data dependency within the DFG. As the data flow is not uniform across the DFG, the bandwidth and latency requirements are also inherently non-uniform across the CGRA. Hence the available bandwidth cannot be fully utilized across all links. Moreover, a homogeneous interconnect handles different data types (e.g., operands and predicates) the same way when ideally they should be treated differently to save energy.

Local Memory : Configuration memory takes up significant power in architectures with spatial-temporal mapping given per-cycle reconfiguration [20]. Typically the opcode, constants, and router settings are encoded into a single configuration word in a homogeneous design. Yet, in reality, these components differ significantly. Constants rarely change, whereas router configurations can change each cycle. Also, identical configuration memory size for all PEs is not desirable as the fraction of useful configuration bits depends on how active a PE is. Thus for configuration memory, the inefficiency is caused by both intra- and inter-PE homogeneity.

2.3 Quantifying Inefficiency of Homogeneous CGRAs

We now quantify inefficiencies concretely with three prominent homogeneous CGRA architectures that serve as our baselines.

HM-ADRES in Figure 5a is inspired by the ADRES [26] template architecture with uniform PEs. Each PE contains a full-fledged ALU with an adder, comparator, multiplier, logic unit, and load-store unit (in PEs directly connected to the data memories).

Each PE has a local register file for storage of intermediate results and configuration memory. The PEs are placed in a grid and communicate with only the immediate neighbors using ALU MOV directive. All the PEs on a row or column are connected through a bus to offer additional connectivity.

Table 1: Power breakdown of homogeneous CGRAs.

		HM-ADRES	HM-HyCUBE	HM-Softbrain
Local Memory	**Configuration**	54.8%	57.7%	13.2%
	Register File	25.6%	-	30%
Compute		4.1%	3.8%	7.3%
Interconnect	**Switches**	1%	3.4%	9.9%
	Registers	-	12.47%	-

Table 2: Area breakdown of homogeneous CGRAs.

		HM-ADRES	HM-HyCUBE	HM-Softbrain
Local Memory	**Configuration**	41%	46.75%	3.45%
	Register File	27.05%	-	41.05%
Compute		19.04%	22.22%	29.28%
Interconnect	**Switches**	5.2%	13.63%	14.8%
	Registers	-	5.7%	-

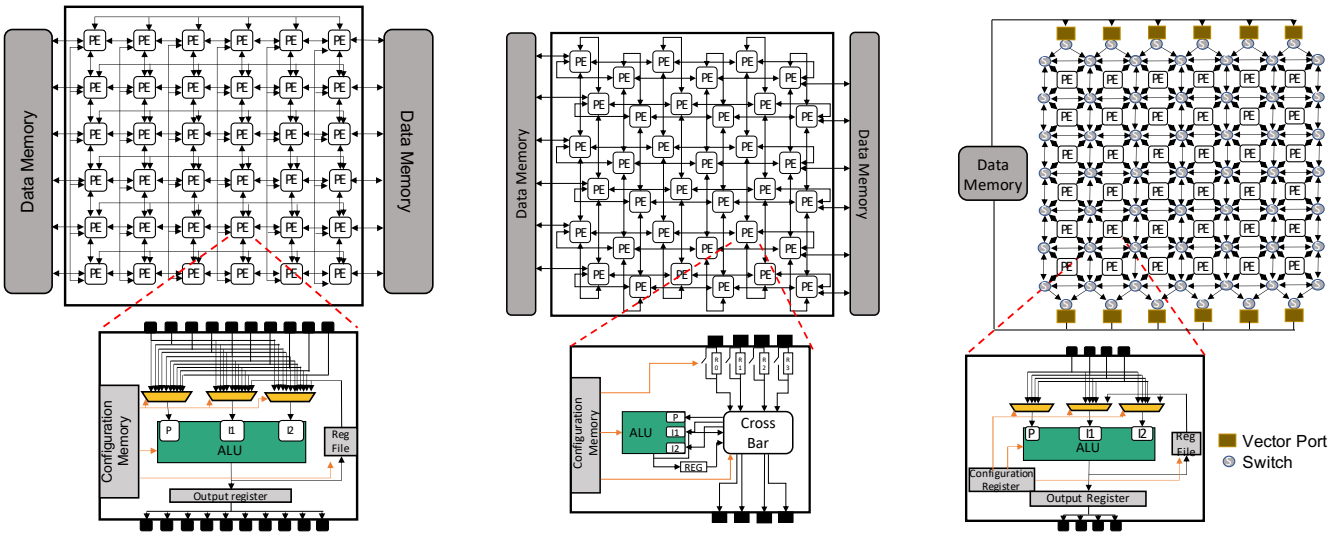
HM-HyCUBE in Figure 5b closely resembles HyCUBE [19, 38]. It has a full-fledged ALU just like HM-ADRES. We replace the mesh network in HyCUBE [19] with a uniform folded torus for a fair comparison. Rich SMART NoC [7] enables single cycle multi-hop connections. Any two PEs in this design have multiple possible paths, unlike HM-ADRES that has only one dedicated path between connected PEs. Each PE in HM-HyCUBE has a crossbar switch; so the PEs can simultaneously compute and communicate. HM-HyCUBE only has configuration memory, while the input side registers to the switches are used for intermediate data storage. Both these architectures are coupled architectures in that the PEs perform both memory access-related operations and computations.

HM-Softbrain (Figure 5c) is a Softbrain [28] style stream dataflow architecture with decoupled memory access and computations [40]. Each PE comprises a fully-fledged ALU with the same capabilities as HM-ADRES and HM-HyCUBE but without any load-store units due to decoupled memory. Input data flows to the PE array through input channels and the computed results flow to memory through output channels. Stream units ensure that there is a continuous data flow to the PE array. HM-Softbrain maps computations only spatially unlike spatio-temporal mapping of the other two CGRAs and hence does not need the configuration memory. A single register per PE holds the configuration data. Each PE has a local register file to store intermediate data values. The interconnect is a circuit-switched network fully pre-configured by the compiler.

A common characteristic across the three architectures is that they are homogeneous, with identical PEs of the same compute capabilities, storage, and interconnect bandwidth.

2.3.1 Power and Area Breakdown. Table 1 and Table 2 show the power and area breakdown of the different components within each architecture. We assume all the architectures have 32-bit data paths and configuration memory and register file to be 256B and 32B per PE (if present in the architecture). Total static and dynamic power with average switching activity is obtained using RTL synthesis with Synopsys Design Compiler on a commercial 22nm process.

For architectures with spatio-temporal mapping (HM-ADRES and HM-HyCUBE), the configuration memory consumes the most power as it is relatively large and accessed every cycle. The area of



(a) HM-ADRES: Resembles ADRES [26] (b) HM-HyCUBE: Resembles HyCUBE [19] (c) HM-Softbrain: Resembles Softbrain [28]

Figure 5: Baseline homogeneous CGRA architectures derived from prominent CGRAs in literature.

the configuration memory is also quite significant, along with any local register files.

A full-fledged ALU with a multiplier allows all compute options. If the memory component is not dominant in an architecture, the compute unit consumes a significant portion of power as is evident with HM-Softbrain. Area wise it is the second-highest contributor for all three architectures. A highly flexible compute unit also requires more configuration bits, increasing the storage cost.

In architectures with sophisticated interconnects (HM-ADRES and HM-Softbrain), the network cost is significant. Apart from any registers placed at the network, the muxes and the crossbar switches are the largest contributor.

2.3.2 Resource Utilization. The efficiency of a system is the amount of useful work done with the consumed resources. In CGRAs, the ideal case will be to achieve 100% utilization across applications. As the CGRA is fully software scheduled, the efficacy of the compiler has a significant impact on utilization. We employ several optimizations such as loop unrolling and loop fusion to maximize the utilization. We analyze the average utilization (Table 3) of the CGRA resources with five application kernels (GeMM, Convolution2D, Stencil2D, crs, and ellpack) from PolyBench [31] and MachSuite [34] benchmark suites that are compatible across all the three architectures. The methodology is detailed in Section 4. The utilization is very low due to uniform resources. For example, even though the configuration memory consumes the most power in HM-ADRES and HM-HyCUBE, the actual valid configuration bits are 12% and 21% of the total memory, respectively. The homogeneous nature of the configurations leads to many unused bits.

Summary : Homogeneous CGRAs lead to idle resources that impose power overheads. This motivates us to explore a comprehensive set of optimizations that can break the uniformity and create heterogeneous realization of the architectures that are more power-efficient. We choose optimizations that can be applied broadly

Table 3: Resource utilization for CGRA elements.

	HM-ADRES	HM-HyCUBE	HM-Softbrain
Compute Utilization	16.5%	28.2%	26.8%
Interconnect Utilization	1.6%	8.8%	13.1%
Valid Configurations	12.5%	21.5%	-

across architectures. *REVAMP* enables design space exploration for such heterogeneous architecture realization across a diverse landscape of CGRAs.

3 REVAMP: EXPLORING CGRA HETEROGENEITY

The *REVAMP* framework comprises micro-architecture level optimizations applicable across diverse CGRA architectures. It thus retains the general-purpose nature of the CGRAs, while optimizing their energy efficiency. We first elaborate on the optimizations for compute, network, and PE-local memory.

3.1 Compute Heterogeneity

Each PE in CGRA has a compute unit, typically a full-fledged ALU that can perform all arithmetic, logic, comparison, and memory operations supported by the ISA. A full-fledged ALU provides the most flexibility as every PE can execute every operation like a general-purpose processor. On the other hand, PEs can be specialized to support only a single compute operation, closer to an ASIC, at much lower power/area overheads.

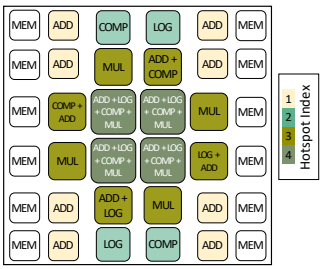
Homogeneous CGRAs have identical ALUs in every PE. Our proposed compute heterogeneity optimizations aim to preserve the performance advantages of full-fledged ALUs, while heterogeneously customizing the ALUs across PEs to save power. *REVAMP* seeks to accelerate a wide range of applications, rather than restricting the application scope. Specifically, given a suite of applications,

it derives the compute unit of each PE to support a subset of operations by judiciously searching the design space across all the PE. Converting full-fledged ALUs to more specialized heterogeneous compute units reduces both static and dynamic power.

- *Distribution of operation classes across the PE array* : A subset of operations is selected for each PE in correspondence with the operation distribution of the target application suite and the supported operations of the homogeneous architecture. We classify the individual operations into five operation types (see Table 6a). *REVAMP* correlates the frequency of occurrence of each class of operations in the PE array with that of the input application suite. For example, if ADD/SUB operation class has a higher frequency of occurrence compared to LOGIC class, more PEs will contain adders. The approach is summarized below.
  For particular operation class C, let f be the percentage occurrence of operations (in a given application suite) belonging to class C. If N_c is the number of PEs supporting class C and N is the total number of PEs,
  $$\frac{N_c}{N} \approx \alpha \times f, \ where\, \alpha \geq 1$$
  As a rule of thumb, we keep $\alpha = 2$ to provide sufficient flexibility. α can be set to explore multiple combinations.
  In a coupled architecture where both memory access and the computations are done in the PE array, a set of PEs will be dedicated for memory access operations [14, 32]. In addition, the area, power cost of different operation classes are considered. For example, multipliers or any special functions like square root units consume more resources compared to adders and comparators. Such bulky units are kept at a minimum to support the required operation.
- *Compute flexibility within a PE* : Compute flexibility in this context is the number of operation classes supported by each PE. Operator classes described above are unevenly distributed for better heterogeneity. We define a hotspot index for each PE. The PEs with higher hotspot indices can support more classes of operations, and thus have higher compute flexibility. Either the user can specify the hotspot index as an input (static mode) or let the framework decide (dynamic mode). In static mode, *REVAMP* spreads the operator classes according to the provided hotspot index, such that those with higher hotspot index will get higher flexibility. In dynamic mode, *REVAMP* maps a subset of applications onto the homogeneous CGRA to identify the potential hotspots and places the operation classes accordingly.
  Figure 6b shows an example of a derived heterogeneous compute architecture. Colour codes represent the statically assigned hotspot indexes where the middlemost PEs have the fullest compute capabilities and the capabilities gradually reduce when reaching the outermost PEs. As multipliers have a larger area footprint, they are usually associated with a higher index. When applying compute diversity, it is important to maintain the equilibrium of the design with respect to thermal characteristics and area.
  Essentially, heterogeneous compute units save resources in two ways. As each compute unit supports a limited subset of operations, power, area savings are realized. In addition, each PE now requires fewer configuration bits, thus also translating to lower configuration memory requirements.

Compute Type	Operations
Adder (ADD)	Add, Subtract
Multiplier (MUL)	Multiply
Logic (LOG)	AND, OR, NOR, XOR, Left Shift, Right Shift
Comparator (COMP)	Equal, Greater than, Less than
Memory (MEM)	Load, Store

(a) Operation subsets



(b) Derived CGRA

Figure 6: Example of heterogeneous compute optimization

Implications for CGRA Compiler. Following summarizes how the compute heterogeneity impacts the CGRA compiler.

- As the set of operations supported by each PE is not uniform, the compiler should know what operations are supported by each PE. This is fed as input to the compiler through the heterogeneous architecture description file (supportedOps in Algorithm 1).
- In mapping a particular operation to a PE node, we first need to identify the candidate set of PEs. Hence, we search in the set of supportedOps and consider only the PEs having the respective compute unit.
- Diversified compute elements leads to resource scarcity for some operations. We thus introduce *scarcity cost*. We categorize the nodes into classes of operations (ADD, MUL, LOG, COMP, and MEM) and obtain the number of nodes in each class. We define *scarcity cost* as the ratio between the number of unmapped DFG nodes of an operation class and the number of empty CGRA PEs supporting that particular class. During DFG node placement on a PE, we calculate *scarcity cost* for the remaining supported classes of that PE to discourages the current placement if the cost to another class is too high.

3.2 Network Heterogeneity

Interconnects play a significant role in CGRAs, supplying each PE's required operands when they are needed, so the PEs need not stall and the performance is maximized. A homogeneous interconnect provides uniform bandwidth to every PE, whereas in an ideal case, the interconnect should provide bandwidth on-demand, according to the application requirements. Again, on one end of the spectrum, we have general-purpose networks like those in multi-core processors, where data is transported through a homogeneous, fixed-width network. On the other end, ASICs transport data directly via point-to-point wires tailored to the specific data width needed. In *REVAMP*, we seek to support the general-purpose communications of diverse data types and traffic on customized datapaths between CGRA PEs to lower the overheads. The key thus

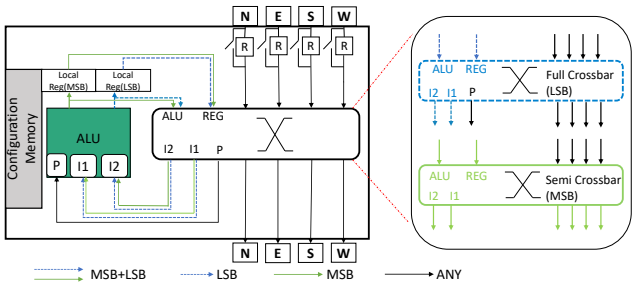


Figure 7: PE microarchitecture with heterogeneous interconnect

lies in the compiler being able to map diverse data flows onto the specific datapaths.

REVAMP explores two heterogeneous optimizations in the CGRA's on-chip network:

- Heterogeneity of the physical resources in the datapath.
- Heterogeneity in the bandwidth allocated for data transfer of each individual data element.

State-of-the-art homogeneous CGRAs have different types of interconnect networks like neighbour-to-neighbour [26], compiler-scheduled packet-switched [19] and circuit switched [28]. The interconnect datapath consists of crossbars or muxes, datapath registers, and wires. *REVAMP* optimizes each of these individual elements and introduces heterogeneity.

3.2.1 Heterogeneous Bit-Width Interconnects : To incorporate heterogeneity to the interconnect, we diversify the bit-width of the datapaths such that the compiler has a choice to select the most efficient one.

- *Internal vs External Datapaths*: We subdivide the datapaths of the interconnect as internal and external datapaths. Internal is the connection within the PE forming datapath between the compute unit and the switch boundary. External datapaths are the connections between PEs. Connections across internal and external datapaths are made heterogeneous by giving internal datapaths twice the number of wires compared to external datapaths. For example, if the architecture has an N-bit compute unit, internal datapaths will be N+1 bits while the external datapaths are N/2 +1 bits (The extra bit is the data valid bit).
- *PE microarchitecture with heterogeneous bit widths*: Figure 7 shows an example micro-architecture of a PE that handles multiple bit-width datapaths. The single crossbar is replaced with two sliced crossbars of half the bit-width. Internal datapaths are MSB and LSB combined, while external datapaths can carry either LSB or MSB. Registers in the datapath are used to establish the pipeline.
- *Saving Resources with Proposed Interconnect Heterogeneity*: As evident from Section 2, most of the interconnect power is contributed by the switches or the crossbar and datapath registers, if any. The proposed optimization can reduce these power consumption with minimal impact on performance, optimizing the overall network power.

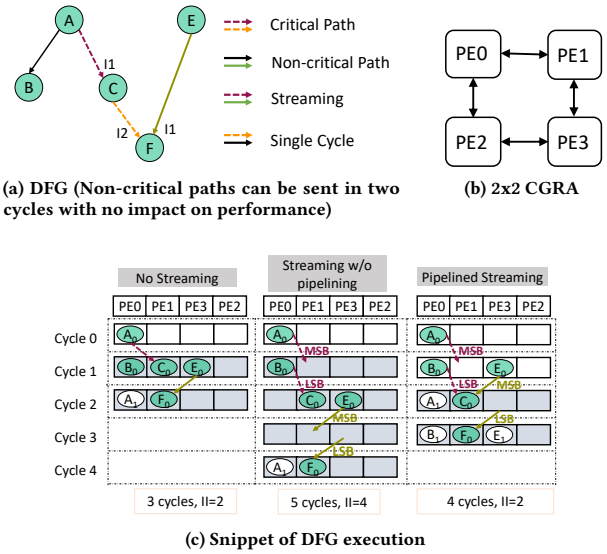
  Once the external datapath size is reduced by half, the registers on the datapath are also reduced by half which saves both static and dynamic power. Additionally, we can gate these datapath registers to save more power.
  In architectures with rich network connectivity, the switch is normally a crossbar. Crossbar is essentially a sea of muxes that takes up significant power and area that quadratically increases with the degree of connectivity. As shown in Figure 7, we slice the individual N bit crossbar to two N/2 bit crossbars. Crossbar channel slicing reduces the internal complexity of each individual crossbar, reducing the overall power and area compared to a single full-sized crossbar.

3.2.2 Implications for CGRA Compiler. With heterogeneous bit-width datapaths, the compiler needs to assign the dataflows accordingly to minimize the performance impact.

- *Single Cycle vs Streaming Data Transfers* : In order to send an N+1 bit data over N/2 +1 bit datapath, the original data word is split into equal LSB (Least Significant Bits), MSB (Most Significant Bits) and appended with a valid bit. If there is a critical latency requirement, such links are prioritized. If so, MSB and LSB can be sent simultaneously along two different paths if routes are available. Else, data is either streamed or sent on two paths at the same time depending on the availability of routes and the cost associated with the routes. If data transfer occurs on two different paths, the compiler selects two of the lowest cost paths and assigns the LSB and MSB transfers accordingly.
- *Pipelined Streaming* : There are scenarios where even critical path edges need to be streamed depending on the availability of routes. Increased latency can get accumulated along the DFG nearly doubling the execution time. We introduce pipelined streaming of data to hide the latencies. In a schedule, if the data for two consecutive computations traverse along independent paths, those transfers can be pipelined.In the example shown in Figure 8, a sample DFG is mapped onto three instances of 2x2 CGRAs. In case there is no streaming, data traversal takes place within single clock cycle. In case of streaming, data traversing between functional units take 2 clock cycles with LSB followed by MSB. If there is no pipelining, transferring output of E starts only after transferring on output of A is completed. In this case the execution of single iteration takes up 5 cycles and II=4 which is reduced to a latency of 4 cycles with II=2 with pipelining.
- *Different Data Types* : Heterogeneity can be further imposed depending on the type of data transferred. Once data is split into LSB and MSB, the MSB may or may not carry useful information. If this can be statically determined at the compilation time, the MSB transfer can be fully eliminated (e.g., predicates carry useful information in LSB only).

3.3 Heterogeneity of Local Memory

The local memories in a PE store configurations and intermediate data. *REVAMP* explores heterogeneity in the configuration memory microarchitecture, by decoupling the configuration memory structure for different hardware components (see Figure 9). Decoupled

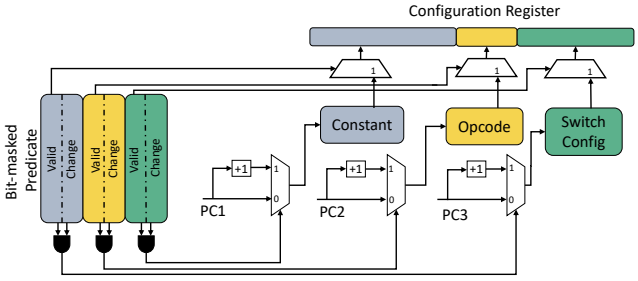


(a) DFG (Non-critical paths can be sent in two cycles with no impact on performance)

(b) 2x2 CGRA

(c) Snippet of DFG execution

Figure 8: Example DFG execution on 2x2 CGRAs with no streaming, with streaming but without pipelining, with pipelined streaming. Same II can be achieved with +1 cycle latency if the streamed MSB and LSB transfers are pipelined.



(a) Architecture of decoupled configuration structure

Figure 9: Proposed decoupled configuration structure

configuration structure removes redundant and invalid configurations, thus reducing the area and power overhead. The separate structures can be clock gated, as the configuration memory reads happen rarely with infrequent changes in configurations, reducing dynamic power.

3.3.1 Decoupled Configuration Structure. We dedicate separate configuration memories for constants, opcodes, switch, and store only the valid configurations. Each of them has separate program counter pointing to the latest read instruction. There is a common configuration register that holds the entire current configuration word. Program counters are incremented only if there is a change required in the corresponding portion of the configuration entry in the common configuration register. Common predicate structure captures the validity and the variability of the sparse configurations. If the valid bit of a particular element is false, it will directly update the common configuration register.

3.3.2 Time Shared Switch Configurations . If the crossbar has a higher degree, the size of the configuration becomes quite significant. For example, the crossbar in HyCUBE [19] accounts for 33% of the configurations. Yet, only a few of the crossbar ports are involved in actual data transfers. So, we subdivide the individual configuration word of the switch into two classes(based on crossbar direction) and allow only one class to change at a time. This restricts the routing flexibility but halves the configuration requirement. We also take into account the hotspot index when adding routing restrictions. If these restrictions lead to an unacceptable performance impact, we ease the restrictions for PEs with a higher hotspot index until a good balance is found.

3.3.3 Non-uniform Memory Sizes. We either pre-set the sizes in configuration or use the hotspot index to diversify the memory sizes allocated to each PE, allocating more memory for those with a higher index. This can be applied to both the configuration store and intermediate value store.

3.3.4 Implications for CGRA Compiler.

- *Resource aware mapping*: We extend the input heterogeneous configurations to add details about the sizes, such that the compiler can keep track of it. If it is not specified, we leverage the hotspot index of compute heterogeneity. The compiler updates the configuration resources and prevents any overflows by blacklisting the PEs whose configuration resources are full.
- *Improving configuration similarity*: The compiler needs to minimize the changes between consecutive configurations such that the individual structures can be compressed even further and switching is minimized. We introduce *Similarity Cost* (see Algorithm 1) to discourage toggling of configurations. *Similarity Cost* is calculated by comparing the last configuration already store and the prospective configuration from the current choice for node placement.
- If time shared switch configurations are enabled, the compiler adds more control into router selection such that it prevents overlapping of classes. The compiler knows the switch configurations belonging to the two classes and it checks if any class is active in a given cycle. If the prospective route does not belong to the active class, it is discarded. This check is done at each hop of the routing algorithm.

3.4 Heterogeneous CGRA Compiler

The compiler that maps the application kernels onto the CGRA is pivotal in determining the performance. As the CGRA architecture is tightly coupled with the compiler, the heterogeneous optimizations need to be partnered with compiler optimizations. Earlier, we have highlighted how heterogeneous architectural optimizations bring about challenging implications to the CGRA compiler. Here, we detail how we extend the HyCUBE [19] mapper and compiler to accommodate these optimizations related to heterogeneity.

Figure 10 illustrates the compilation flow. All the heterogeneous CGRA optimizations are included in the mapping stage. Hence the DFG generation is similar to the state-of-the-art approaches.

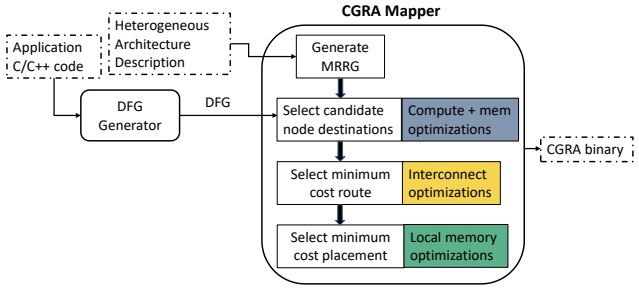


Figure 10: Heterogeneous CGRA compilation flow

The CGRA mapper takes as input a DFG and the architectural description, which details the heterogeneous microarchitectural optimizations (i.e., any heterogeneous CGRA instance generated by REAVMP), and generates the modulo schedule for execution in binary format. The objective of the mapping algorithm is to achieve the minimum possible Initiation Interval (II). The mapper initially takes the minimum II as the maximum of resource minimum II (ResMII) and recurrence minimum II (RecMII). Minimum II is used to create a time-extended version of the CGRA resource graph known as the Modulo Routing Resource Graph (MRRG) [8]. The mapper takes an iterative approach, where the II value is increased by one in each iteration, to obtain a valid mapping of the DFG onto the MRRG taking all the heterogeneous optimizations into account. For each II value, the sorted DFG nodes are first placed in candidate locations followed by routing. It uses heuristic cost models to determine the best placements and routes.

Problem Definition. Given a DFG $D = (V_D, E_D)$ where V_D correspond to nodes of the graph and E_D to the edges of the graph and a heterogeneous CGRA instance, the problem is to obtain the minimally time extended MRRG $M_{II} = (V_M, E_M)$ which has a valid mapping $\phi = (\phi_V, \phi_E)$ of D on M_{II}.

Compute heterogeneity: Let us define O_v as the set of operations supported by v where $v \in V_M$ and $O_v \subset O$, O is the set of all supported operations. In an architecture with heterogeneous compute units, $\forall u \in V_D, if \phi_V(u) = v, u_{op} \in O_v$ where u_{op} is the operation of the node u. As illustrated in Figure 10, the compute constraints are taken into account and *scarcity cost* updated when selecting the set of candidate destinations for the node placement.

Interconnect heterogeneity: The problem statement is redefined to accompany the proposed heterogeneous interconnect. Given a DFG $D = (V_D, E_D)$, we create a edge-splatted DFG $D' = (V_D, E'_D)$, $\forall e \in E_D, e' = \{e_{msb}, e_{lsb}\}$, where e_{msb}, e_{lsb} are equal splits of e. If there exist a valid mapping $\phi' = (\phi'_V, \phi'_E)$ of D' on M_{II}, $\forall e' \in E'_D, \phi'_E(e') = \{\phi'_E(e_{msb}), \phi'_E(e_{lsb})\}$. If $e' \in$ {set of single cycle paths}, $\phi'_E(e_{msb}) == \phi'_E(e_{lsb})$. The set of single cycle paths is determined statically depending on whether any valid data is carried on MSB or whether data transfer is within the same PE. Edges carrying predicates is an example of such single cycle paths. Interconnect related optimizations are applied in the route selection stage of the mapper.

Heterogeneity of local memory: Memory resource availability is checked during the candidate PE selection stage and the candidate is discarded if sufficient resources are not available. *Similarity Cost* to capture frequent toggling of configurations is added at the minimum cost selection phase.

Algorithm 1: Mapping single DFG node onto MRRG

Data: DFG node,Partially mapped MRRG
Result: Candidate destinations with minimum cost for placement and routing, Route

```
1  InitialCandi = getEmptyNodes(MRRG);
2  ResAvaiCandi = checkResources(InitialCandi);
3  ComputeAvaiCandi = checkCompute(ResAvaiCandi);
4  MappedParentChild = getMapped(DFGNode);
5  foreach dest in ComputeAvaiCandi do
6      foreach node in MappedParentChild do
7          isSingleCycle = isSingleRoute(DFGNode, node, dest);
8          if isSingleCycle then
9              {Route,Cost} = getMinCostRoute(DFGNode,node,dest);
10         else
11             isCriticalPath = isCriticalPath(DFGNode, node);
12             if isCriticalPath then
13                 dePrioritizeStreaming(true);
14             {Route_LSB, Cost_LSB} =
                getMinCostRoute(DFGNode, node, dest);
               {Route_MSB, Cost_MSB} =
               getMinCostRoute(DFGNode, node, dest);
               Cost = Cost_LSB + Cost_MSB;
               Route = {Route_LSB, Route_MSB};
15     Cost+ = scarcityCost(node, MRRG) +
         similarityCost(node, MRRG, Route);
16 return {MinimumCostDest, Route};
17 Function getMinCostRoute(DFGNode,RelNode,dest)
18     routesWithCost = getRoutes(RelNode, dest);
19     if timeSharedSwitchConfig then
20         removeClassConflicted(routesWithCost);
21     minCostRoute = getMinCost(routesWithCost);
22     return minCostRoute;
```

Algorithm 1 shows how heterogeneous optimizations are applied in mapping one of the DFG nodes. Given an unmapped DFG node, first, the possible placement destinations are filtered considering memory heterogeneity (Line 2) and compute heterogeneity (Line 3). The algorithm then iterates over possible placement destinations estimating the routing cost to already mapped parents or children. In the case of streaming, the cost is calculated for both LSB and MSB transfers (Line 14). It will consider the configuration memory availability in determining the minimum cost route (Line 19-20). Finally, the total cost for the placement destination is calculated considering similarity cost and scarcity cost (Line 15).

4 EXPERIMENTAL EVALUATION

4.1 Experimental Methodology

We apply *REVAMP* framework on three prominent but distinctly different homogeneous CGRAs to derive heterogeneous architectures with the proposed optimizations. The derived heterogeneous architectures along with the homogeneous versions are implemented in System Verilog RTL [1] and synthesized on a commercial 22nm process using Synopsys Design Compiler to obtain the power and area values. The compiler optimizations for *REVAMP* are implemented

Table 4: Application kernels used in the evaluation

Application Kernel	Description	Domain	Nodes	Edges
GeMM	General Matrix Multiply	Linear Algebra	61	77
Conv2d	2-dimensional convolution	Deep Learning	59	72
CRS	Compresses Sparse Row, SPMV	Graphic Processing	27	33
ellpack	SPMV ELLPACK format	Graphic Processing	41	50
stencil2d	2-dimensional stencil computation	Image Processing	27	33

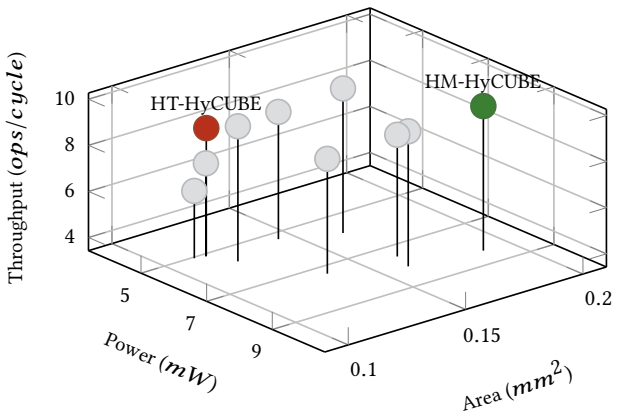


Figure 11: Design space exploration of heterogeneous variants of HM-HyCUBE with *REVAMP*, arriving at HT-HyCUBE.

in C++, extending an existing CGRA compiler [19] to take in as inputs a comprehensive architectural description in JSON format and the dataflow graph (DFG) corresponding to the loop kernel of an application. The LLVM [10] based DFG generator is used for generating the DFG from C/C++ application code of the kernels.

Benchmark Applications. Table 4 shows the set of application kernels used with our framework. These kernels, selected from MachSuite [34] and PolyBench [31] benchmark suites, are computationally intensive and suitable for acceleration on CGRAs covering widely used application domains.

4.2 Derived Heterogeneous Architectures

Figure 11 illustrates the design space exploration by *REVAMP* on HyCUBE [19] (HM-HYCUBE). We select the architecture instance with the best balance in terms of power, area, and throughput as the derived heterogeneous design. Similarly, corresponding heterogeneous architectures are derived for ADRES [26] (HM-ADRES) and Softbrain [28] (HM-Softbrain). Note that the PE array size (6×6), clock frequency (100 MHz), and the granularity of operations (32-bit) are kept constant across the three architectures to make a fair comparison. Table 5 summarizes the heterogeneous optimizations added by the framework on the derived architectures. Let us name the derived heterogeneous architectures as HT-ADRES, HT-HyCUBE, and HT-Softbrain. Figure 12 provides a visual depiction of the derived heterogeneous architectures and Table 6 details their specifications.

Compute heterogeneity is applied to all three architectures. As shown in Table 6, the compute units of HT-ADRES and HT-HyCUBE are identical because their execution models are similar with close coupling of memory and compute operations, resulting in a similar distribution of the operations. As Softbrain decouples the memory

Table 5: Heterogeneous optimizations applied on the selected homogeneous architectures

		HM-ADRES	HM-HyCUBE	HM-Softbrain
Heterogeneous Compute		✓	✓	✓
Heterogeneous Interconnect		✗	✓	✓
Local Memory	**Configuration**	✓	✓	✗
	Data	✓	✗	✓

Table 6: Derived heterogeneous architectures specifications

		ADRES		HyCUBE		Softbrain	
		HM	HT	HM	HT	HM	HT
Compute Units	**Adders**	36	36	36	36	36	36
	Multipliers	36	7	36	7	36	16
	Comparators	36	6	36	6	36	7
	Logic	36	19	36	19	36	13
	Load/Store	12	12	12	12	-	-
Datapath (bit)		33	33	33	33 - 17	33	33 - 17
Config Memory per PE (B)		256	66.5/87.8	256	75.5 /105.7	9	9
Local Memory (B)		16	8/16	-	-	16	8/16

address generation operations from the PE array, the multiplications become significant for most applications. Hence, a higher number of multipliers are used to reduce the impact on performance.

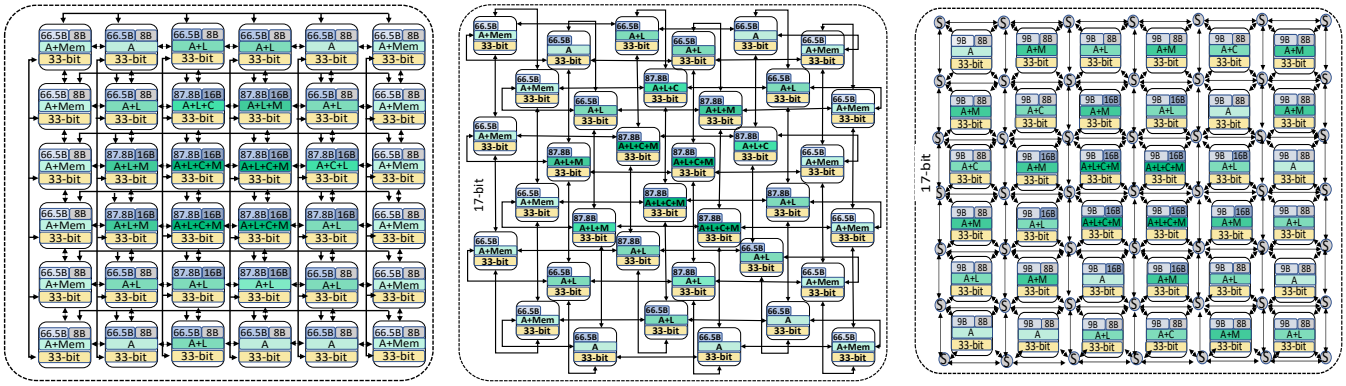
Both HT-HyCUBE and HT-Softbrain end up with heterogeneous interconnect as the rich network architectures in their homogeneous versions allow greater options for heterogeneity. But ADRES has a more restricted network where each PE has dedicated connections with every other PE in the same row/column. Hence two PEs are connected by at most one path and diagonal PEs are not connected. This narrows the scope for applying interconnect heterogeneity. Moreover, HM-ADRES interconnect does not have complex switches or registers.

Only architectures with spatio-temporal mapping require configuration memory. This eliminates configuration memory-related optimizations in HT-SoftBrain. In addition to the decoupled configuration structure, for HT-ADRES and HT-HyCUBE we vary the memory size in each PE, where hotspot PEs have twice the memory of other PEs.

We now analyze the derived heterogeneous architectures to quantify the power/area gains with heterogeneity and how it impacts the performance. We use power, area utilization, performance (in terms of II), average energy, and power efficiency of the PE array to evaluate the efficiency of the architecture.

4.3 Power-Performance Trade-off

REVAMP aims to explore the heterogeneous optimizations to improve the power efficiency of CGRAs while minimizing the impact on performance compared to the homogeneous versions. We quantify the performance of an architecture by the II (Initiation Interval) values for selected application kernels mapped on that architecture through our compiler. The lower the II value, the higher the throughput of the application on that architecture. Table 7 summarizes the obtained II values for derived heterogeneous architectures and the homogeneous versions. We use the best unrolling factor for each of the three baselines and the same DFG across homogeneous and heterogeneous versions.



(a) HT-ADRES (b) HT-HyCUBE (c) HT-Softbrain

Figure 12: Derived Heterogeneous Architectures (The letters A, M, L, C and Mem stands for Adder, Multiplier, Logic unit, Comparator and Memory unit respectively. The local memories are specified in Bytes(B)).

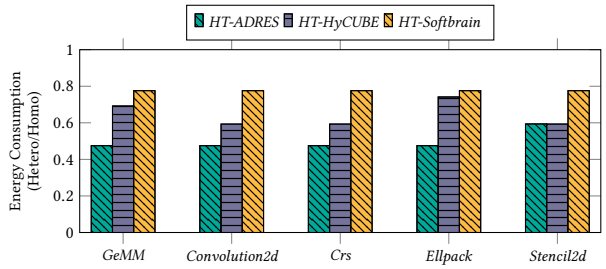


Figure 13: Energy reduction with heterogeneity.

Table 7: Initiation Interval(II) obtained for homogeneous and derived heterogeneous architectures.

	ADRES		HyCUBE		Softbrain	
	HM	HT	HM	HT	HM	HT
GeMM	5	5	6	7	1	1
Convolution2d	4	4	4	4	1	1
CRS	6	6	4	4	1	1
ellpack	5	5	4	5	1	1
Stencil2d	4	5	4	4	1	1

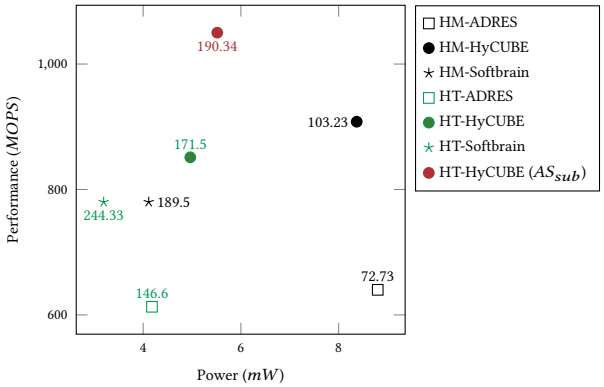


Figure 14: Performance-power comparison of the PE array. Node labels indicate the power efficiency in MOPS/mW. HT-HyCUBE (AS_{sub}) is derived using subset of the application suite (GeMM, Convolution2d and Stencil2d), for which optimal point is different.

The slight increase in II for two kernels in HT-HyCUBE is because of some critical paths (backedges) missing single cycle routes, while in HT-ADRES it is the limited connections to heterogeneous compute. However, we show that the energy consumption in heterogeneous CGRA reduces significantly to provide an excellent energy-performance trade-off in battery-operated devices.

Figure 13 illustrates the total energy for execution (normalized w.r.t homogeneous) of the kernels. We obtain 36% average energy reduction, while the maximum is 50.1% in HT-ADRES.

We compare the power efficiency of the PE array for homogeneous architectures and the derived heterogeneous architectures averaged across all the kernels. As we are highlighting the heterogeneity impact, we consider only the PE array in this analysis assuming conflict-free data flow from the external memories. Figure 14 shows that heterogeneity increases the power efficiency of the PE array by 1.5x. The ability of *REVAMP* in exploring the effect of different application suites on power-performance of heterogeneous CGRA optimizations is also highlighted by the design point HT-HyCUBE (AS_{sub}) – an architecture derived from a subset of the kernels (GeMM, Convolution2d, Stencil2d) partnered with heterogeneous optimizations can have substantially higher performance.

4.4 Application-Level Evaluation

We showcase the design-space exploration with *REVAMP* on a few applications at the edge.An application comprises one or more compute-intensive kernels, which are suitable for acceleration. We select five applications (from Mibench [16], CortexSuite [37] benchmark suites and C++ implementation of MobileNet V1 [18] and LeNet-5 [22]) and extract the compute-intensive kernels to form the kernel suite for each domain. These kernel suites are then fed as input to *REVAMP* along with HM-HyCUBE as the homogeneous

Table 8: Selected applications

Application	Domain	Kernels
Lenet-5 [22], Mobilenet V1 [18]	Deep Learning	Convolution, GeMM, Average Pooling
GSM [16]	Telecommunication	APCM_inverse_quantization, APCM_quantization, Autocorrelation, Gsm_coder, Gsm_decoder, Weighting_filter
Motion Detection [37]	Computer vision	GrayscaletoRGB, RGB2Grayscale, Taylor_app
Rijndael [16]	Security	encrypt, decrypt

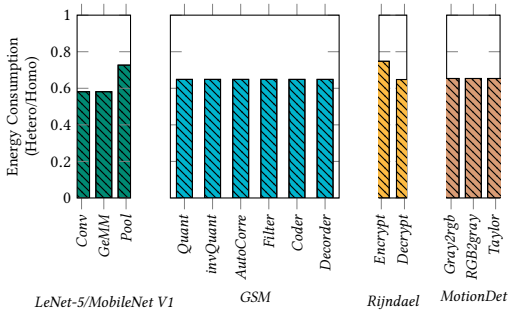


Figure 15: Energy comparison for applications running on derived HT-HyCUBE architectures.

CGRA input. Table 8 summarises the selected applications, their domain, and the extracted kernel suite.

REVAMP performs the design space exploration converging to 04 Pareto-optimal heterogeneous architectures for each application domain. Figure 15 shows normalized energy consumption (against the HM-HyCUBE) when kernels execute on individually generated heterogeneous architectures. We compare the energy for kernel execution as the remaining execution and data transfers are identical in both homogeneous and heterogeneous versions. Every kernel demonstrates a reduction in energy consumption, implying application-level energy savings.

4.5 Resource Utilization and Power/Area Breakdown

We showed in Section 2 that there is a significant resource under utilization in the homogeneous architectures. We now analyze how the proposed optimizations impact resource utilization. We define compute utilization as the ratio between the total number of compute operations in the DFG and the number of available compute units (number of compute units in the PE array multiplied by the II value of the corresponding DFG on the architecture). For a more precise comparison, we consider the adder, multiplier, logic, comparator, and memory elements within a PE as separate individual compute units in contrast to Table 3 where we simply assumed one compute unit per PE. We calculate the interconnect utilization as the percentage of used links with respect to the total available links (number of links in the PE array multiplied by the II value for the DFG). In terms of configurations, we take the percentage of valid configuration bits from the total amount of configuration bits used (this includes valid configurations and redundant/NOP configurations). As there are only II configurations in each PE, a portion of the memory will be empty in most of the cases. In a decoupled structure, this void occurs only in bit-masked predicate (Figure 9) , which reduces empty memory size by nearly 7x.

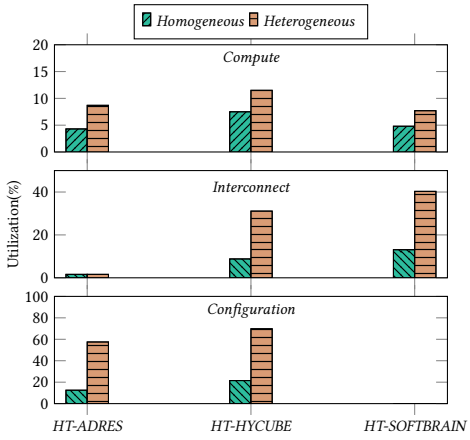


Figure 16: Resource utilization comparison.

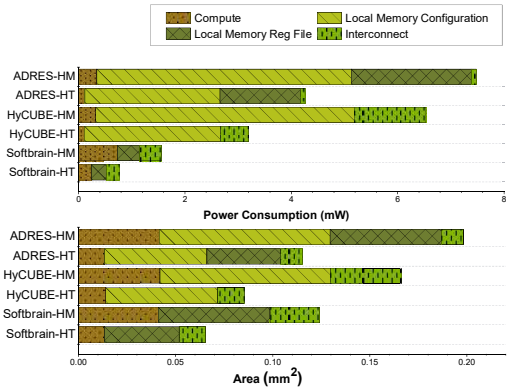


Figure 17: Power and Area Distribution

Figure 16 shows the resource utilization for the derived heterogeneous architectures compared with the homogeneous versions. There is a significant improvement in the resource utilization for interconnect and configuration memory. The compute utilization is improved by around 1.8x, and interconnect, configuration utilization are improved by 3.1x and 3.9x, respectively.

4.6 Analysis of Subcomponent Impact

Figure 17 shows the power and area breakdown across individual components of heterogeneous architectures compared with the homogeneous architectures. Still, the distribution in heterogeneous architecture is quite similar to that of homogeneous architecture. If individual components are considered, for spatio-temporal architectures, the configuration memory component produces the most significant reduction in terms of power and area. This is evident for HyCUBE and ADRES in figure 17. When the memory component is less significant, for example in spatial Softbrain architecture, compute heterogeneity provides significant reduction especially in terms of power. Similarly, the impact of network heterogeneity is most significant when the starting point is a homogeneous CGRA with sophisticated networks consuming substantial resources. Throughput can be affected by microarchitecture and mapping

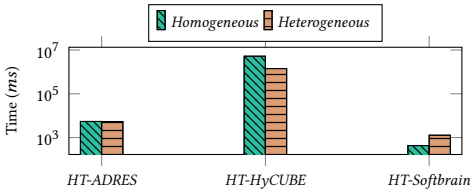


Figure 18: Average compilation time

quality of individual sub-components or a combination of sub-components. The increase in II value (Table 7) for Stencil2d execution in HT-ADRES is from the compute heterogeneity. For GeMM and ellpack, the increase of II value in HT-HyCUBE is mainly due to the combination of compute and network heterogeneity.

4.7 Compilation Time

To analyze how heterogeneity affects the original compiler performance, we compare the compilation time averaged over the set of kernels. Figure 18 shows that HT-ADRES takes the same compilation time as HM-ADRES. In contrast, compilation time is faster in HT-HyCUBE, because the restrictions of the search space end up reducing the search time. In the case of Softbrain, the compilation is much faster (few ms) than HyCUBE or ADRES, as the decoupling of memory access from compute simplifies the mapping. There is a slight increase in compilation time for HT-Softbrain versus HM-Softbrain, mainly because of additional search due to network heterogeneity. Still, the average compilation time is around 1 sec.

5 RELATED WORK

Some prior CGRAs introduced a limited amount of heterogeneity, essentially just compute heterogeneity, in an ad-hoc fashion to a specific CGRA architecture. *REVAMP*, on the other hand, covers a broader range of heterogeneity across compute, storage and interconnect, and is a design framework that explores and judiciously introduces heterogeneity for a variety of CGRA architectures.

Compute: Compute heterogeneity proposed in the literature can largely be classified as those with a variety of compute units that support different, specialized operations or units with varying precision. PACT-XPP [5] and Plasticine [32] combine two types of functional units one for memory based operations and the other for ALU operations. BilRC [3], DPU [27], PPA [29] and SNAFU [14] further separate the multipliers from the ALU. RASP [12] functional units are augmented with floating-point dividers and real datatype adders. But, BilRC [3] and RASP [12] report 68% and 43.9% average utilization, respectively. EGRA [2] decomposes the single ALU functionality into four different units as logic only, logic+shift, logic+add, and logic+shift+add to form clusters with different combinations. Apart from ALU clusters, there are memory tiles and multiplier tiles. EGRA does the mapping at the expression level instead of operator level that is more suited for regular applications. However, the general architecture derived from the template has ALUs with full functionality (logic+shift+add).

Among prior works that propose different precision levels among the functional units, TRANSPIRE [33] supports a combination of integer and floating-point functional units, while on-the-fly CGRA [6] incorporates heterogeneous compute units each with diverse levels of approximation.

Interconnect: There has been little prior CGRA research into heterogeneous interconnects. ADRES [26] and Morphosys [36] have regular neighbor-to-neighbor mesh with additional links connecting all the elements in a row and column. Homogeneous torus, switch-based and NoC-based interconnects are present in TRANSPIRE [33], Softbrain [28] and HyCUBE [19], respectively. Apart from the inherent heterogeneity in meshes (where the edges of the mesh have less traffic), the only heterogeneity seen in the interconnect is the hierarchical layout when the architecture is clustered, for example in Morphosys [36].

Memory: Memories in PEs include register files for intermediate data storage and memories for the storage of reconfiguration words. Most prior literature addresses overhead from memories independently and proposes different compression techniques [11, 35]. These works still maintain the uniformity of the design, rather than moving to a heterogeneous architecture.

CGRA compiler: There are prominent CGRA compilers based on simulated annealing (DRESC [25]), graph theory (EPIMap [17], Graph Minor [8]), Graph Neural Networks (LISA[23]), hierarchical mapping (HiMaP[41]), ILP (CGRA-ME[9]), and heuristics (HyCUBE [19]). Usually these basic approaches are customized to cater for specific architectures. Specially, for the heterogeneous architectures discussed earlier, the compiler is customized. For example EGRA [2] does the mapping at expression level. Plasticine [32] maps based on patterns.

CGRA design tools: As the compiler is tightly coupled with the architecture, the design process of CGRAs is quite complex. A few prior works (CGRA-ME [9], DSAGEN [39] and Pillars [15]) address this issue by proposing end-to-end design frameworks for CGRAs. These tools perform the simulation and RTL generation for a provided architecture. However, they do not support heterogeneity.

6 CONCLUSION

CGRAs are at a nascent stage of development, with a few commercial products, and many exciting research CGRA architectures proposed in the last decade. Just as the widespread adoption of FPGAs is enabled by extensive toolchain support, design tools for CGRAs can help pave the way for their wider adoption. Design tools are particularly critical for CGRAs given the closely coupled hardware-software interface, with CGRA hardware completely scheduled by the CGRA compiler. Here, we propose *REVAMP*, a design space exploration framework for heterogeneous CGRA realization. The derived heterogeneous architectures achieve 38.5% average reduction in core power and 29.8% reduction in core area with 36% total energy reduction.

ACKNOWLEDGMENTS

We would like to thank our shephard and the anonymous reviewers for their valuable feedback. This research is partially supported by the National Research Foundation, Singapore under its Competitive Research Programme Award NRF-CRP23-2019-0003.

A ARTIFACT APPENDIX

A.1 Abstract

REVAMP artifact includes the complete framework comprising the heterogeneous architecture generator, heterogeneous CGRA mapper, parameterized RTL and scripts for power, area calculation. We elaborate on the *REVAMP* tool flow with an example of generating a pareto-optimal heterogeneous CGRA from a 4x4 homogeneous CGRA targeting five application kernels.

A.2 Artifact Check-List (Meta-Information)

- **Algorithm:** *REVAMP* architecture generator, Heterogeneous CGRA mapper
- **Program:** Application kernels included in DFG (data-flow graph) format
- **Compilation:** cmake 3.5 or higher, C++ 8.X or 9.X for CentOS 8.X, Synopsys Design Compiler (tested on R-2020.09, Proprietary)
- **Model:** Technology library for synthesis (Proprietary)
- **Run-time environment:** CentOS 8.X with cmake, C++, nlohmann json, and Synopsys DC installed. Python virtual environment included. Root access will be needed in setting up prerequisites.
- **Hardware:** CPU
- **Metrics:** Generated heterogeneous architectures and design space exploration in terms of power efficiency, area efficiency, and performance.
- **Output:** Numerical performance estimations from compiler, architecture power and area estimations from RTL synthesis
- **Experiments:** Python scripts along with a README file elaborating on the workflow.
- **How much disk space required (approximately)?:** ~ 500MB
- **How much time is needed to prepare workflow (approximately)?:** ~ 10 min
- **How much time is needed to complete experiments (approximately)?:** ~ 5 hrs
- **Publicly available?:** yes
- **Code licenses (if publicly available)?:** MIT License
- **Workflow framework used?:** Workflow automated using python scripts
- **Archived (provide DOI)?:** 10.5281/zenodo.5848404

A.3 Description

A.3.1 How to Access. Download the tool package from Zenodo.

Unzip and run all the below mentioned commands inside the `./REVAMP` folder.

```
cd REVAMP
```

A.3.2 Hardware Dependencies. Our experiments were run on a server with 10 Intel Xeon cores running at 2.70GHz and 2GB RAM. Synthesis using commercial tools will take a substantial amount of time; hence we recommend a server with comparable memory and core counts.

A.3.3 Software Dependencies. CentOS 8.X with cmake 3.5 or higher and c++ 8.X are installed. The architecture generator and the CGRA mapper are developed in C++ and require nlohmann json as a prerequisite. Synopsys Design Compiler(DC) R-2020.09 is used to estimate the power and area of the generated heterogeneous architectures. Python scripts are used to automate the flow and the required python virtual environment configurations are provided.

A.3.4 Data Sets. We include application kernels in DFG (data-flow graph) format generated with the Morphor DFG generator (https://github.com/ecolab-nus/Morpher_DFG_Generator.git).

A.3.5 Models. We use proprietary standard cells on a commercial technology node for synthesis and hence they are not included herewith. We provide scripts for synthesis with Design Compiler where one can modify the scripts to include an available technology node. Details will be given in the next section and in the README.

A.4 Installation

A.4.1 Prerequisites.

- Check cmake(3.5 or newer) and C++ versions(8.X or 9.X for CentOS)
  The framework was tested on CentOS 8.5 and C++ 8.5.
- Installing nlohmann JSON:
  `chmod +x install_json.sh`
  `./install_json.sh` (Will require root access)
- Check for synthesis tools:
  Run the following command to ensure design compiler is configured properly.
  `which dc_shell`
- Activate python virtual environment

```
virtualenv revamp_env
source revamp_env/bin/activate
pip3 install -r py_environment.txt
```

A.4.2 REVAMP Toolchain Installation. Execute the following command to build the heterogeneous architecture generator and the heterogeneous CGRA mapper.

```
python3 scripts/installation.py
```

A.5 Experiment Workflow

We demonstrate the *REVAMP* workflow by performing heterogeneous design space exploration with a 4x4 homogeneous CGRA and set of application kernels as inputs.
The workflow comprises 4 main steps.

- **Heterogeneous architecture generation:**
  Inputs:
  - Path to folder containing application kernels in DFG format (`./APPLICATIONS/applications/`)
  - Homogeneous architecture description in JSON format (`hycube_original_4x4_torus.json in ./HOMOGENEOUS_ARCHITECTURE/`)
  - User configurations (`./ARCHITECTURE_GENERATOR/config.json`)

  Tool:
  - REVAMP Architecture Generator (`./ARCHITECTURE_GENERATOR/src/build/generator`)

  Outputs:
  - Generated heterogeneous architectures in JSON format (`./ARCHITECTURE_GENERATOR/generated_architectures`)
  - Heterogeneous parameter configurations for RTL (`./ARCHITECTURE_GENERATOR/generated_architectures_RTL_config`)

The file naming suggests the applied heterogeneity
Ex: `heterogeneous_CGRA_compute_L_network_config` means all three elements of heterogeneity, i.e compute, network and memory are applied.
To execute: `python3 scripts/revamp_run_generator.py`

- **Heterogeneous CGRA compiler:**
  Inputs:
  – Application kernels in DFG format
  – Generated heterogeneous architecture description in JSON format
  Tool:
  – REVAMP heterogeneous mapper (Developed as an addon for Morphor CGRA mapper, https://github.com/ecolab-nus/Morpher_CGRA_Mapper.git) (`./HETEROGENEOUS_MAPPER/src/build/heterogeneous_compiler*`)
  Outputs:
  – Performance in terms of II (Initiation Interval) of the mapped kernel and throughput in MOPS. (`./mapper.log`, `./HETEROGENEOUS_MAPPER/throughput.rpt`)
  To execute: `python3 scripts/revamp_run_mapper.py`
- **RTL synthesis:**
  Inputs:
  – Parameterized RTL (`./HOMOGENEOUS_ARCHITECTURE/RTL/`)
  – Generated heterogeneous RTL parameter configurations
  – Scripts to run Design Compiler (`./synthesis_scripts`)

  NOTE: Update ADDITIONAL_SEARCH_PATH variable with path to your technology db files and TARGET_LIBRARY_FILES with technology db name in ./synthesis_scripts/common_setup.tcl
  Tool:
  – Synopsys Design Compiler (Proprietary)
  Outputs:
  – Power and area estimates (`./synthesis_scripts/power.rpt`, `./synthesis_scripts/area.rpt`)
  To execute: `python3 scripts/revamp_run_synthesis.py`
  NOTE: This step will take ~4 hrs. To skip this step for the given example please copy power.rpt and area.rpt in ./synthesis_scripts/synthesis_output to ./synthesis_scripts and move to analysis of the results.
- **Design analysis and converging to Pareto optimal:**
  Analysis of performance, power, and area to decide on the Pareto-optimal heterogeneous architecture.
  To execute: `python3 scripts/revamp_run_analysis.py`
  output: `./analysis_reports`

Complete workflow can be executed automatically by running the following command. Please ensure the synthesis script is updated with available cells and process technology files.

```
chmod +x revamp_run.sh
./revamp_run.sh
```

To run without synthesis (if access is restricted for Synopsys tools),

```
chmod +x revamp_run_wo_synthesis_ex.sh
./revamp_run_wo_synthesis_ex.sh
```

NOTE: This will take ~5 hours to complete.
We provide detailed steps and commands in the README provided herewith.

A.6 Evaluation and Expected Results

We demonstrate the functionality of the artifact with a smaller CGRA (4x4) and with a restricted search scope. To derive the pareto optimal heterogeneous architecture from the generated architectures, we analyze the power, area and performance.

A summary of the results is saved in `./analysis_reports`.

- comparetoHomogeneous.rpt : Ratio between derived architectures vs baseline architecture(ex: power(derived architecture)/power(baseline architecture)) for throughput, power, and area
- dse.rpt: optimal design choices in terms of power efficiency and area efficiency, and the impact on throughput
- dse.png: Visualizes the design space of the considered architectural instances.

Expected analysis reports from a trial run of the above example is added in `./analysis_reports_expected/4x4_CGRA`.

We have added the analysis reports for design space exploration corresponding to Figure 11 of the paper in `./analysis_reports_expected/6x6_CGRA`.

Though the tool does not restrict CGRA size, we use a smaller CGRA size(4x4) and a more constrained search space for faster validation of the tool functionality.

A.7 Experiment Customization

We elaborate on how *REVAMP* supports customization in the README provided. Users can experiment on different homogeneous architectures, different search scopes and application DFGs by customizing the user inputs to the framework, prompting *REVAMP* to derive different heterogeneous CGRAs.

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/\artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] 2018. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018). https://doi.org/10.1109/IEEESTD.2018.8299595
[2] G. Ansaloni, P. Bonzini, and L. Pozzi. 2011. EGRA: A Coarse Grained Reconfigurable Architectural Template. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems.* https://doi.org/10.1109/TVLSI.2010.2044667
[3] Oguzhan Atak and Abdullah Atalar. 2013. BilRC: An Execution Triggered Coarse Grained Reconfigurable Architecture. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems.* https://doi.org/10.1109/TVLSI.2012.2207748
[4] Thilini Kaushalya Bandara, Dhananjaya Wijerathna, Tulika Mitra, and Peh Li-Shiuan. 2021. *REVAMP: A Systematic Framework for Heterogeneous CGRA Realization.* https://doi.org/10.5281/zenodo.5748656

[5] Volker Baumgarten, G. Ehlers, Frank May, Armin Nückel, Martin Vorbach, and Markus Weinhardt. 2003. PACT XPP—A self-reconfigurable data processing architecture. *The Journal of Supercomputing*. https://doi.org/10.1023/A:1024499601571

[6] M. Brandalero, A. C. S. Beck, L. Carro, and M. Shafique. 2018. Approximate On-The-Fly Coarse-Grained Reconfigurable Acceleration for General-Purpose Applications. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. https://doi.org/10.1109/DAC.2018.8465930

[7] C. O. Chen, S. Park, T. Krishna, S. Subramanian, A. P. Chandrakasan, and L. Peh. 2013. SMART: A single-cycle reconfigurable NoC for SoC applications. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. https://doi.org/10.7873/DATE.2013.080

[8] Liang Chen and Tulika Mitra. 2012. Graph minor approach for application mapping on CGRAs. In *2012 International Conference on Field-Programmable Technology*. https://doi.org/10.1145/2655242

[9] S. Chin and Jason Anderson. 2018. An architecture-agnostic integer linear programming approach to CGRA mapping. In *DAC Design Automation Conference 2018*. https://doi.org/10.1109/DAC.2018.8465799

[10] Lattner Chris and Adve Vikram. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. https://doi.org/10.1109/CGO.2004.1281665

[11] S. Das, K. J. M. Martin, and P. Coussy. 2019. Context-memory Aware Mapping for Energy Efficient Acceleration with CGRAs. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. https://doi.org/10.23919/DATE.2019.8715288

[12] Fan Feng, Li Li, Kun Wang, Feng Han, Baoning Zhang, and Guoqiang He. 2016. Floating-point Operation Based Reconfigurable Architecture for Radar Processing. *IEICE Electronics Express*.

[13] Taro Fujii, Takao Toi, Teruhito Tanaka, Katsumi Togawa, Toshiro Kitaoka, Kengo Nishino, Noritsugu Nakamura, Hiroki Nakahara, and Masato Motomura. 2018. New Generation Dynamically Reconfigurable Processor Technology for Accelerating Embedded AI Applications. In *2018 IEEE Symposium on VLSI Circuits*. https://doi.org/10.1109/VLSIC.2018.8502438

[14] Graham Gobieski, A. Atli, K. Mai, Brandon Lucia, and Nathan Beckmann. 2021. SNAFU: An Ultra-Low-Power, Energy-Minimal CGRA-Generation Framework and Architecture. https://doi.org/10.1109/ISCA52012.2021.00084

[15] Yijiang Guo and Guojie Luo. 2020. Pillars: An Integrated CGRA Design Framework.

[16] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. https://doi.org/10.1109/WWC.2001.990739

[17] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. 2012. EPIMap: Using Epimorphism to map applications on CGRAs. In *DAC Design Automation Conference 2012*. https://doi.org/10.1145/2228360.2228600

[18] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *ArXiv* (2017).

[19] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. 2017. HyCUBE: A CGRA with Reconfigurable Single-Cycle Multi-Hop Interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017*. https://doi.org/10.1145/3061639.3062262

[20] Manupa Karunaratne, Cheng Tan, Aditi Kulkarni, Tulika Mitra, and Li-Shiuan Peh. 2018. DNestMap: Mapping Deeply-Nested Loops on Ultra-Low Power CGRAs. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. https://doi.org/10.1109/DAC.2018.8465833

[21] C. Kim, M. Chung, Y. Cho, M. Konijnenburg, S. Ryu, and J. Kim. 2012. ULP-SRP: Ultra low power Samsung Reconfigurable Processor for biomedical booktitle=2012 International Conference on Field-Programmable Technology, applications. https://doi.org/10.1109/FPT.2012.6412157

[22] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* (1998). https://doi.org/10.1109/5.726791

[23] Zhaoying Li, Dan Wu, Dhananjaya Wijerathne, and Tulika Mitra. 2022. LISA: Graph Neural Network based Portable Mapping on Spatial Accelerators. In *28th IEEE International Symposium on High-Performance Computer Architecture*.

[24] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. 2019. A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications. (2019). https://doi.org/10.1145/3357375

[25] Bingfeng Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. 2003. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *2003 Design, Automation and Test in Europe Conference and Exhibition*. https://doi.org/10.1109/DATE.2003.1253623

[26] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Haris Man, and Rudy Lauwereins. 2003. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. https://doi.org/10.1007/978-3-540-45234-8_7

[27] Chris Nicol. 2017. A Coarse Grain Reconfigurable Array (CGRA) for Statically Scheduled Data Flow Computing.

[28] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-Dataflow Acceleration. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. https://doi.org/10.1145/3079856.3080255

[29] Hyunchul Park, Yongjun Park, and Scott Mahlke. 2009. Polymorphic Pipeline Array: A Flexible Multicore Accelerator with Virtualized Execution for Mobile Multimedia Applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*.

[30] Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. 2020. A Survey on Coarse-Grained Reconfigurable Architectures from a Performance Perspective. (2020). https://doi.org/10.1109/ACCESS.2020.3012084

[31] Louis-Noel Pouchet. [n. d.]. PolyBench/C. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[32] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Patterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. https://doi.org/10.1145/3079856.3080256

[33] R. Prasad, S. Das, K. J. M. Martin, G. Tagliavini, P. Coussy, L. Benini, and D. Rossi. 2020. TRANSPIRE: An energy-efficient TRANSprecision floating-point Programmable archItectuRE. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*. https://doi.org/10.23919/DATE48585.2020.9116408

[34] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for Accelerator Design and Customized Architectures. In *Proceedings of the IEEE International Symposium on Workload Characterization*.

[35] Mohammad Hossein Sargolzaei and Siamak Mohammadi. 2018. Energy Efficient Configuration Unification and Compression for CGRAs. In *Microprocessors and Microsystems*. https://doi.org/10.1016/j.micpro.2018.06.010

[36] H. Singh, Ming-Hau Lee, Guangming Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho. 2000. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. In *IEEE Transactions on Computers*. https://doi.org/10.1109/12.859540

[37] Shelby Thomas, Chetan Gohkale, Enrico Tanuwidjaja, Tony Chong, David Lau, Saturnino Garcia, and Michael Bedford Taylor. 2014. CortexSuite: A Synthetic Brain Benchmark Suite. In *International Symposium on Workload Characterization (IISWC)*. https://doi.org/10.1109/IISWC.2014.6983043

[38] Bo Wang, Manupa Karunarathne, Aditi Kulkarni, Tulika Mitra, and Li-Shiuan Peh. 2019. HyCUBE: A 0.9V 26.4 MOPS/mW, 290 pJ/op, Power Efficient Accelerator for IoT Applications. In *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. https://doi.org/10.1109/A-SSCC47793.2019.9056954

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

[40] Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunarathne, Anuj Pathania, and Tulika Mitra. 2019. CASCADE: High Throughput Data Streaming via Decoupled Access-Execute CGRA. *ACM Trans. Embed. Comput. Syst.* (2019). https://doi.org/10.1145/3358177

[41] Dhananjaya Wijerathne, Zhaoying Li, Anuj Pathania, Tulika Mitra, and Lothar Thiele. 2021. HiMap: Fast and Scalable High-Quality Mapping on CGRA via Hierarchical Abstraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021). https://doi.org/10.1109/TCAD.2021.3132551