R-Blocks: An Energy-Efficient, Flexible, and Programmable CGRA

BARRY DE BRUIN, Tampere University, Tampere, Finland and University of Technology Eindhoven Faculty of Electrical Engineering, Eindhoven, Netherlands
KANISHKAN VADIVEL, IMEC, Eindhoven, Netherlands and University of Technology Eindhoven Faculty of Electrical Engineering, Eindhoven, Netherlands
MARK WIJTVLIET, ASMPT, Beuningen, Netherlands
PEKKA JÄÄSKELÄINEN, Tampere University, Tampere, Finland
HENK CORPORAAL, University of Technology Eindhoven Faculty of Electrical Engineering, Eindhoven, Netherlands

Emerging data-driven applications in the embedded, e-Health, and internet of things (IoT) domain require complex on-device signal analysis and data reduction to maximize energy efficiency on these energy-constrained devices. Coarse-grained reconfigurable architectures (CGRAs) have been proposed as a good compromise between flexibility and energy efficiency for ultra-low power (ULP) signal processing. Existing CGRAs are often specialized and domain-specific or can only accelerate simple kernels, which makes accelerating complete applications on a CGRA while maintaining high energy efficiency an open issue. Moreover, the lack of instruction set architecture (ISA) standardization across CGRAs makes code generation using current compiler technology a major challenge. This work introduces R-Blocks; a ULP CGRA with HW/SW co-design tool-flow based on the OpenASIP toolset. This CGRA is extremely flexible due to its well-established VLIW-SIMD execution model and support for flexible SIMD-processing, while maintaining an extremely high energy efficiency using software bypassing, optimized instruction delivery, and local scratchpad memories. R-Blocks is synthesized in a commercial 22-nm FD-SOI technology and achieves a full-system energy efficiency of 115 MOPS/mW on a common FFT benchmark, 1.45× higher than a highly tuned embedded RISC-V processor. Comparable energy efficiency is obtained on multiple complex workloads, making R-Blocks a promising acceleration target for general-purpose computing.

CCS Concepts: • **Computer systems organization → Reconfigurable computing**; • **Hardware → Hardware-software codesign;**

Additional Key Words and Phrases: Coarse-grained reconfigurable architecture, HW/SW co-design, code generation, Energy efficiency

Authors' addresses: B. de Bruin, Tampere University, Tampere, Pirkanmaa, Finland, Electronic Systems, University of Technology Eindhoven Faculty of Electrical Engineering, Eindhoven, Noord-Brabant, Netherlands; e-mail: e.d.bruin@tue.nl; K. Vadivel, IMEC, Eindhoven, Netherlands, Electronic Systems, University of Technology Eindhoven Faculty of Electrical Engineering, Eindhoven, Noord-Brabant, Netherlands; e-mail: k.vadivel@tue.nl; M. Wijtvliet, Center of Competence, ASMPT, Beuningen, Netherlands; e-mail: mark.wijtvliet@asmpt.com; P. Jääskeläinen, Tampere University, Tampere, Pirkanmaa, Finland; e-mail: pekka.jaaskelainen@tuni.fi; H. Corporaal, Electronic Systems, University of Technology Eindhoven Faculty of Electrical Engineering, Eindhoven, Noord-Brabant, Netherlands; e-mail: h.corporaal@tue.nl.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1936-7406/2024/05-ART34
https://doi.org/10.1145/3656642

ACM Reference Format:
Barry de Bruin, Kanishkan Vadivel, Mark Wijtvliet, Pekka Jääskeläinen, and Henk Corporaal. 2024. R-Blocks: An Energy-Efficient, Flexible, and Programmable CGRA. *ACM Trans. Reconfig. Technol. Syst.* 17, 2, Article 34 (May 2024), 34 pages. https://doi.org/10.1145/3656642

1 INTRODUCTION

In the embedded, **digital healthcare (e-Health)**, and **internet of things (IoT)** domains all kinds of data-driven signal analysis applications are being deployed on processing nodes with a limited energy budget and an **ultra-low power (ULP)** envelope (<10 mW). Examples of these applications can be found in the e-Health domain, e.g. continuous detection of complex epileptic seizures using a wearable battery-operated EEG headset [15], in the industrial computer vision domain, e.g. fast visual servoing using camera-based OLED center detection [25], and also in intelligent speech interfaces or mobile voice assistants [20]. For many of these applications, the algorithms are still under development and are not fixed, the computational requirements change when the input data dimensions change, and the computations are very diverse. Accelerating these applications through an **application-specific integrated circuit (ASIC)** or an **application-specific instruction processor (ASIP)** has a very high manufacturing cost and has a high risk of quickly becoming obsolete due to future algorithmic developments. This motivates the need for a flexible and reconfigurable general-purpose computing platform while maintaining a high energy efficiency (>100 MOPS/mW).

Within the context of ULP signal processing platforms, **coarse-grained reconfigurable architectures (CGRAs)** have been proposed as a good compromise between flexibility and energy efficiency for embedded vision, bio-medical and signal processing applications [8, 14–19, 23, 30, 45, 47, 57, 63], and are often coupled to a single-core or multi-core RISC processor as a loop nest accelerator, or they operate standalone. A CGRA consists of a grid or array of **functional units (FUs)** or **processing elements (PEs)** that are interconnected through a configurable switching fabric or **network-on-chip (NoC)**. Different from **field-programmable gate arrays (FPGAs)**, which also have coarse-grained units, like DSP blocks, BRAM memories, and recently even complete **very long instruction word (VLIW)** cores [66], the unit of reconfiguration of CGRAs should be well above the bit-level, of an FPGA, i.e. the minimum unit of reconfiguration could be the arithmetic or memory operation of PEs. Compared to CPUs (including vector and VLIW cores), where instructions reconfigure the complete datapath every cycle, in CGRAs part of the datapath (i.e. PEs, interconnect, or both) reuses the same configuration for most of the program (e.g. for the dominant loop nests and kernels), which lowers the number of instruction bits required to execute an application [64]. These aspects enable CGRAs to achieve a much higher energy efficiency than FPGAs and CPUs [2, 29, 59, 64].

Many CGRAs fit in this classification, but there is still a wide spectrum of design decisions on the trade-off curve between energy efficiency and flexibility. CGRAs with the highest energy efficiency combine a reconfigurable interconnect with static PEs that employ a modulo-scheduled **data-flow graph (DFG)** mapping of the kernel under consideration [8, 18, 19, 23, 29, 47]. A major limitation of the above-mentioned works is that the modulo-scheduled DFG execution model is too inflexible for more irregular and complex workloads, which requires manually splitting the application or executing irregular parts of the application on a host processor, which leads to mapping inefficiencies [23] and under-utilization of the CGRA fabric [18]. Recently, there has been a trend towards using CGRAs for more general-purpose computing, which includes off-loading complete kernels and applications, including control flow, onto the fabric [14–17, 23, 30]. To improve the flexibility many CGRAs consist of a fabric with programmable PEs and interconnect to

explicitly schedule PE operations and data transports between PEs while bypassing a centralized **register file (RF)**, which is also referred to as software bypassing [52]. To maintain energy efficiency, the interconnect is typically simplified to a nearest-neighbor communication network or a static reconfigurable 2D-mesh network [14, 19, 30, 57, 63].

Unfortunately, most of these works lack a compiler and do compare manually written assembly with compiler-generated baselines [15, 17, 63]. Additionally, recent works that present a CGRA compiler are generally target-specific with a very specific execution model and only present code generation for one specific instance [14, 23, 30, 57], which makes reuse of existing compiler developments challenging. Finally, most CGRAs limit themselves to the exploitation of **instruction-level parallelism (ILP)**, and CGRAs that also exploit **data-level parallelism (DLP)** require specialized vector processing units [18, 19, 40, 43, 45, 65], which reduce the flexibility of the fabric. In this work we build upon the Blocks CGRA tool-flow, as presented in [62], and present R-Blocks. R-Blocks is a ULP CGRA with reconfigurable data and control network and programmable FUs. The fabric can be used to overlay compiler-programmable VLIW-SIMD cores with an arbitrary vector width, due to the reconfigurable control network that can broadcast instructions to multiple FUs, thereby exploiting DLP without excessive specialization by means of dedicated vector units. We present a HW/SW co-design tool-flow built on top of the OpenASIP toolset [27] and demonstrate efficient code generation for application-specific VLIW-SIMD cores for 15 complex benchmarks. The main contributions of this work are:

(1) The R-Blocks CGRA; a customizable fabric to overlay compiler-programmable VLIW-SIMD cores with software bypassing support (Section 3). To the best of our knowledge, R-Blocks is the first compiler-programmable CGRA with support for flexible SIMD-processing (Section 4).
(2) An accompanying HW/SW co-design tool-flow based on the OpenASIP toolset [27] (Section 5). This tool-flow is able to compile 15 complex benchmarks written in the C language, whose code quality approaches the performance of manual mappings (1.39-2.14×, Section 6.3), while reducing the RF traffic using software bypassing (0.23-0.43×, Section 6.2.2), compared to a highly tuned embedded RISC-V processor.
(3) An in-depth CGRA evaluation in terms of energy and area efficiency. Compared to the state of the art (Section 7), we obtain an energy efficiency of 115 MOPS/mW on a common FFT benchmark, which is 1.45× better than a highly tuned embedded RISC-V processor (Section 6.4). Overall, we report an area penalty of 2.66-5.14× compared to ASIPs with dedicated SIMD units and fixed datapath (Section 6.5).

The rest of this paper is organized as follows: Section 2 discusses related work on ULP CGRAs and HW/SW co-design tool-flows, Section 3 introduces the R-Blocks physical architecture. In Section 4 the R-Blocks virtual architecture is explained. Section 5 details the R-Blocks HW/SW co-design tool-flow. A detailed experimental evaluation is provided in Section 6, followed by a comparison with the prior art in Section 7. Concluding remarks are provided in Section 8.

2 RELATED WORK

2.1 Energy-efficient Signal Processing Using CGRAs

A recent ULP CGRA that achieves extreme energy efficiency is [47], which achieves 584.9 GOPS/W[1] peak energy efficiency on an FIR filter and is able to execute several other kernels (e.g. FFT) with extremely high energy efficiency. The basic idea is that both the PEs and NoC are

[1]All executed operations on the CGRA fabric are included in this energy efficiency value. To compare energy efficiency between computing platforms for a given workload, independent of implementation quality, it is more representative to

statically configured using a spatially mapped DFG, and that the fabric executes 16-bit operations, which is sufficient precision for specific application domains. Another recent example [8] of an SoC with CGRA that is able to achieve an extremely high energy efficiency of 538 GOPS/W[1] (16-bit operations). Again, both PEs and NoC are statically configured, but the CGRA supports dynamic partial run-time reconfiguration to improve PE utilization for complete applications. The CGRA is able to accelerate vision applications using the Halide **domain-specific language (DSL)** and compiler, which limits its use for other application domains. Riptide [23] combines static PE assignment with a reconfigurable interconnect that supports control flow operations. This enables execution of arbitrary C-code without wasting PEs on control flow operations. Using a custom compiler with constrained programming mapper, Riptide obtains an energy efficiency of 180 MOPS/mW (32-bit integer operations) on a matrix multiplication kernel, and comparable results on several signal processing and linear algebra workloads.

A major limitation of the above-mentioned works is while CGRAs with spatial mappings can be extremely energy-efficient for certain application domains, the execution model is too inflexible for more irregular and complex workloads. As such, the control flow, memory address generation, and other irregular parts of an application are traditionally executed on a host processor, and only the inner loops are off-loaded to the CGRA fabric, which can lead to sub-optimal application-level energy savings due to under-utilization of the CGRA [18]. Additionally, if temporal computation is not possible, a computation graph might not fit on the fabric, which requires manual effort to split the graph [22]. Recently, there has been a trend towards using CGRAs for more general-purpose computing, which includes off-loading complete multi-kernel applications, including control flow, to the fabric [14–17, 30].

To improve the flexibility many CGRAs consist of a fabric with programmable PEs and NoC. To maintain energy efficiency, the interconnect is typically simplified to a static nearest-neighbor network or reconfigurable 2D-mesh network. The efficiency of these CGRAs is typically slightly lower at the kernel-level, but time-sharing enables mapping of more complex applications. For example, ULP-SRP achieves an energy efficiency of 59.4 MOPS/mW[1] on an FFT kernel and accelerates an ECG heartbeat detection application[30]. The fabric is able to switch between different modes, i.e. 4-issue VLIW mode or 16-issue CGRA mode, depending on the amount of ILP within a kernel. ρ-VEX [7] is another example of a reconfigurable VLIW processor that dynamically adapts to the available ILP and **task-level parallelism (TLP)**. However, the centralized RF is a major energy bottleneck, which emphasizes the need for architectures with distributed RFs and software bypassing, such as R-Blocks. Another recent work [14] is able to obtain an average energy efficiency of 142 MOPS/mW[1] while executing signal processing kernels and a smart visual surveillance application using a MIMD execution model with a lightweight global synchronization mechanism. However, the flexible execution model limits the energy efficiency. Blocks [63] is a CGRA with programmable PEs and a reconfigurable interconnect using the VLIW execution model. To reap the benefits of spatial computation, zero-overhead hardware loops are included to allow for spatial computation of inner loops where the instruction delivery remains static. The Blocks fabric accelerates complete applications and is able to reconfigure the network at run-time with a tolerable reconfiguration penalty on a complex epileptic seizure detection algorithm [15]. Blocks achieves 164.9 MOPS/mW[1] while executing an FFT kernel. R-Blocks optimizes the Blocks template with a reconfigurable instruction cache, improved memory subsystem, and support for inter-lane communication in the ALU, as well as a compilation tool-flow to enable application-scale acceleration. Some smaller changes were also made, i.e. the oversized $32b \times 32b \rightarrow 64b$ multiplier in the MUL

only count operations that are intrinsic to the workload, e.g. N^3 Multiply-Accumulate-Shift operations for a $N \times N$ fixed-point matrix multiplication, as will be discussed in more detail in Section 7.

unit was replaced by a 32b × 32b→32b multiplier to save energy, and the switchbox topology was changed from fully-connected (which scales very poorly with an increasing number of tracks) to a more area-optimized Wilton structure.

It follows from the above-mentioned works that there is an energy efficiency gap between the static and programmable CGRAs, but that programmable CGRAs are more suitable for acceleration of complete applications. Application-level evaluations are rather rare, and most efforts are limited to small kernels, probably motivated by the lack of a proper compiler, or use DSLs to target specific application domains. Some works report on application-level energy savings but are lacking a compiler and do therefore compare manually written assembly towards compiler-generated baselines [15–17, 63]. These comparisons aid in understanding how these complex architectures should be programmed, but it remains unclear to what degree a compiler is able to approach these results. One work reports compiler-generated application-level savings and reports 10× energy savings over a RISC-V processor [14]. However, it remains unclear why exactly their CGRA saves that much energy, and whether the baseline architecture is properly tuned. In this work, we perform a detailed analysis of the quality of our compiler-generated mappings and compare them against a RISC-V processor with a comparable operation set, optimized memory hierarchy, and highly tuned algorithmic mappings.

2.2 HW/SW Co-design for CGRAs

Recent works that present a CGRA compiler are generally target-specific, and only present compilation results for one specific instance [23, 30, 57]. In contrast, we present a CGRA HW/SW co-design tool-flow based on the open-source retargetable OpenASIP compiler [27], and demonstrate efficient code generation for application-specific VLIW-SIMD cores for 15 complex benchmarks. Due to the lack of a well-defined execution model, most CGRAs limit themselves to the exploitation of ILP through the mapping of an unrolled or modulo-scheduled DFG [18, 19, 57]. Some CGRAs also exploit DLP, but the amount of DLP (i.e. the vector width) is decided at design-time using dedicated **single instruction, multiple data (SIMD)** hardware [18, 19], which makes the CGRA very specialized. Outside the ULP domain, there are some high-performance CGRAs that include support for exploiting DLP, but they also require specialized SIMD hardware, only support modulo-scheduled DFG mappings, or have limited energy efficiency [40, 43, 65]. R-Blocks exploits ILP and DLP with flexible SIMD support, compiles arbitrary C programs using a well-defined VLIW-SIMD execution model with support for software bypassing, and achieves state-of-the-art energy efficiency within a ULP budget. The R-Blocks fabric also supports TLP through multi-processing, but this feature is currently not exploited due to the lack of software support and efficient hardware primitives for inter-core communication and synchronization.

To enable rapid CGRA **design space exploration (DSE)**, a fast retargetable compiler and performance estimation model that can deal with all architectural configurations is tremendously important. Recently, some frameworks for (automated) DSE of CGRAs have been proposed. Recent examples include CGRA-ME [10], DSAGEN [58], OpenCGRA [49], and REVAMP [3]. These tool-flows focus on fast compilation and simulation of small kernels on a highly customizable CGRA template and often include rapid energy and area estimation models to quickly evaluate many design points. Hardware generation tooling is often included to implement the optimal design points. The scheduler in CGRA-ME is based on **integer programming (IP)**, which is too slow and restricted for DSE for larger CGRAs and applications. DSAGEN proposes to iteratively update the hardware mapping using a schedule repair method to speed up DSE, and not completely map every design point from scratch every time a variation in the architecture is made. OpenCGRA uses a fast analytical regression model for area and power estimates, similar to the approach presented in [61], and a heuristic-based mapper, inspired by the approach presented in [29], for fast

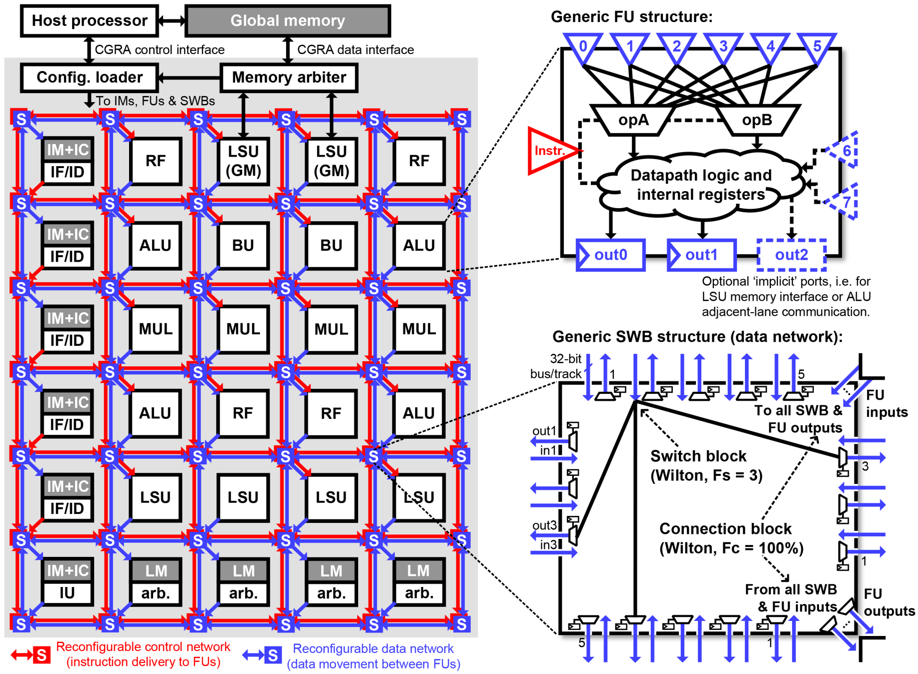


Fig. 1. R-Blocks system example, consisting of a CGRA fabric, global memory, and host processor.

performance estimates. The authors mention that a complete DSE run finishes within a couple of minutes, but is heavily dependent on the mapping algorithm. REVAMP presents a DSE framework for heterogeneous CGRAs with respect to compute, interconnect, and local storage. The authors of [37] propose a framework for exploration of the CGRA interconnect, which is commonly seen as a major source of inefficiency. DSE for more programmable architectures is AEx [26], however, AEx is limited to automated DSE of ILP architectures, and doesn't optimize for DLP.

3 R-BLOCKS CGRA & TOOL-FLOW

This section presents the R-Blocks CGRA fabric, its execution model, and discusses a mapping example from high-level C-code to assembly using the R-Blocks tool-flow.

3.1 Architecture Overview

The R-Blocks CGRA or **physical architecture (physArch)** is a fabric to overlay multi-issue VLIW-SIMD cores with software bypassing support. These cores are called virtual cores or **virtual architectures (virtArchs)**. An example R-Blocks system, which includes a CGRA fabric, is depicted in Figure 1. R-Blocks enables the processor designer to construct ASIPs that support the VLIW or SIMD execution model, as well as a combination of both. R-Blocks consists of a grid of programmable FUs and a reconfigurable interconnect with two switchbox networks (for data and control), similar to an FPGA but with less overhead due to the word-level granularity of the FUs and interconnect. The interconnect is typically reconfigured once per dominant loop nest or kernel program, and remains static during program execution to minimize reconfiguration overhead.

FUs can be customized to support different instructions. Having heterogeneous FU types allows for reuse of the instruction encoding space across FUs, keeping the instruction words small. A

processor designer is able to add his own FUs and instruction-set extensions to provide more heterogeneity, highlighting the flexibility of the fabric. A summary of the FUs (excluding the RF unit) used for this paper:

- **BU:** it supports control flow operations and generates the program counter for a virtual core. It includes an **instruction cache (IC)** controller. Multiple BUs enable execution of parallel processors on the same fabric.
- **LSU:** it supports memory operations. **Load-store units (LSUs)** either connect to LM units over the reconfigurable data network to create scratchpad memory or connect to the **global memory (GM)** arbiter.
- **ALU:** it supports basic (bitwise) logic, arithmetic, and comparison operations. Also, some constant logical and arithmetic shifts are supported. It includes 'implicit' adjacent-lane communication ports for vector processing.
- **MUL:** it supports (fixed-point) multiplication operations as well as some constant arithmetic shifts.
- **IU:** it supports loading of long (32-bit) immediate values and requires wider **instruction memories (IMs)** (33-bit) than the regular IMs (13-bit) of other FUs.

FUs are programmed using **instruction fetch and decode units (IF/IDs)** that can be connected to the instruction port of one or more FUs over the reconfigurable control network. These IF/IDs are controlled by the CGRA itself using a **branch unit (BU)** that broadcasts the **program counter (PC)** to one or multiple IF/IDs over the reconfigurable data network. The idea of having an interconnect with separate control and data networks for instruction delivery and data movement is adopted from the Blocks architecture [62]. Having a reconfigurable control network allows us to broadcast an instruction from an IF/ID to multiple FUs to emulate SIMD-execution without dedicated vector units (more details follow in Section 4.1). This provides more flexibility for the processor designer, as FUs can be used to increase either the issue width or the vector width. This approach has been demonstrated to outperform traditional VLIW and SIMD processors in the original Blocks architecture [63], due to its ability to create custom processors with a varying VLIW issue width and a suitable amount of SIMD-parallelism per application.

3.2 Execution Model

The execution model of the R-Blocks virtArch consists of VLIW-SIMD cores with support for software bypassing, which allows the compiler to directly schedule data transfers between FUs and avoid a centralized RF. An example of an R-Blocks virtArch with the corresponding programming model is depicted in Figure 2 (left). The reconfigurable data network connects up to N different bypass sources to FU input ports, where N is limited by the number of operand selection bits in the instruction word. Several types of bypass sources are supported: FU output port registers, RF outputs, LSU outputs from a **local memory (LM)** unit or the GM. The unit mix and bypass sources per unit are specified in the virtArch description file, as depicted in Figure 2 (right). Processor designers can design a processor with an arbitrary number of FUs and define a custom datapath in the virtArch description file. These kind of parallel processors with software-controlled data movement are also referred to as **exposed datapath architectures (EDPAs)** [52].

Code generation for this type of architecture has been challenging, mostly due to the constrained connectivity between FUs and the lack of a global RF. In this work we exploit the OpenASIP tool-flow to generate code for arbitrary VLIW-SIMD cores with support for software bypassing. In the next section we will walk through the mapping of an example application using the R-Blocks framework.

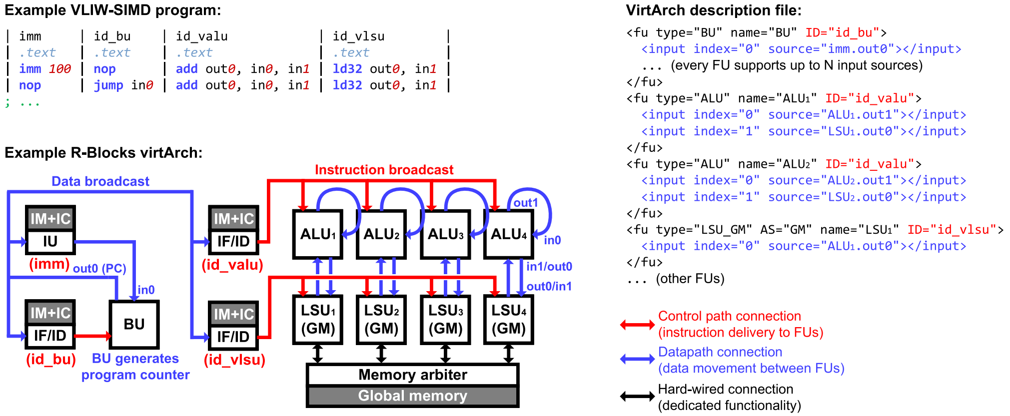


Fig. 2. Example of an R-Blocks VLIW-SIMD core with 4 issue slots and 4-wide 'virtual' vector units and its programming model.

3.3 Compilation & Mapping Overview

Figure 3 shows the source code of a vectorized *binarization* kernel, which is compiled for a 7-issue VLIW-SIMD machine with 4-wide vector units. The kernel loads four pixels from the `input_samples` array in the GM address space (`AS_GM`) using the scatter-gather memory interface, binarizes the data based on a `THRESHOLD` constant, and writes the resulting binary pixels back to GM. The resulting parallel assembly of the inner loop computation is depicted in Figure 3. All seven issue slots share the same program counter and operate in lock-step, in a VLIW fashion. The first four columns correspond to the scalar datapath, while the right-most three columns correspond to the vector datapath.

While this implementation of the *binarization* kernel is merely an illustrative example, it can be observed that one loop iteration takes six cycles (lines 6–11) to binarize a single vector. To further improve performance the vector width can be increased and loop unrolling or software pipelining can be used to overlap multiple loop iterations. This can be accomplished by modifying the virtArch and application code. Further improvements include the use of more complex load/store instructions with automated address generation, and the use of zero-overhead loops to reduce loop overhead. However, compiler support for these features is currently very preliminary. The resulting assembly code and virtual architecture can be simulated using a fast cycle-accurate simulator.

To execute this application on real hardware, the virtArch need to be mapped to the fabric or physArch, which describes the physical resources in terms of FUs and interconnect topology in more detail and can be used to generate HDL source files. This step is performed using a heuristic-based placement and routing tool. In this step, logical IMs and FUs are mapped onto physical instances and connections are routed over the reconfigurable switchbox networks. Depending on the size of the virtArch and the available resources within the physArch, the placement and routing step takes typically less than a few minutes. It should be noted, however, that functionality and application performance can already be simulated based on the virtArch. Finally, the assembly is converted to a binary that contains the code segments of every instruction stream, including a header to indicate which physical IM it is mapped to. A bitfile contains the configurations of all FUs and switchbox networks, which were generated by the placement and routing tool.

The host processor initiates a new CGRA acceleration request via the CGRA control interface using the configuration loader. The loader retrieves the program binary and/or bitfile from GM,

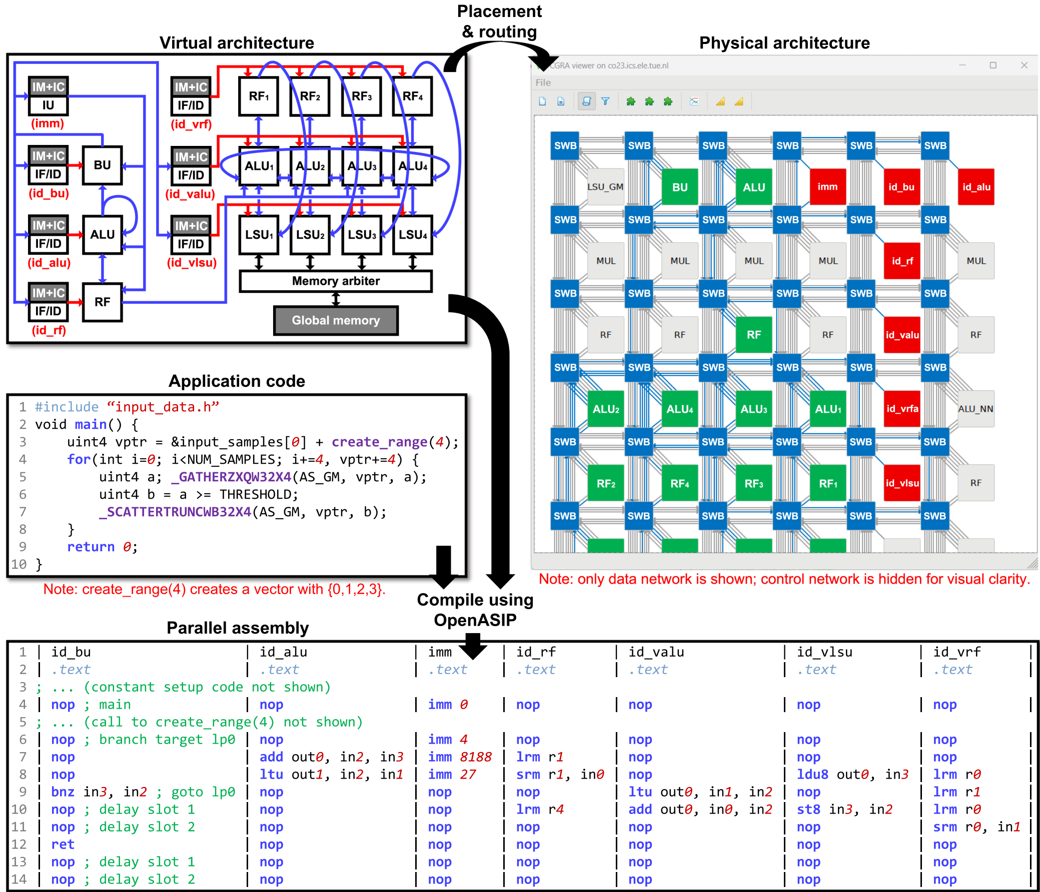


Fig. 3. An illustrative example of a vectorized *binarization* kernel on an R-Blocks CGRA fabric. The C-code is compiled for the provided virtual architecture using the retargetable OpenASIP compiler, assembly is generated, and the virtual architecture is mapped onto a physical architecture using a placement and routing tool.

copies the instruction streams from the binary to their designated IMs, and configures the FUs and switchbox networks using the bitfile. Once a valid configuration is loaded, the host processor starts CGRA program execution and will be notified upon its completion. Successive CGRA acceleration requests can reuse the loaded binary and bitfile to reduce CGRA reconfiguration overhead.

4 R-BLOCKS VIRTUAL ARCHITECTURE

In this section, we present an overview of the **instruction set architecture (ISA)** extensions and micro-architectural optimizations to optimize the energy efficiency and enable flexible SIMD-processing on the R-Blocks CGRA.

4.1 Instruction Delivery

Within a virtual core, a **branch unit (BU)** generates the **program counter (PC)** and executes control flow instructions based on the program and configuration that is loaded by the configuration

[2]Based on post-synthesis simulations of R-Blocks CGRA-L instance with 512-line instruction memory and 64-line cache array (see also Section 6.1).

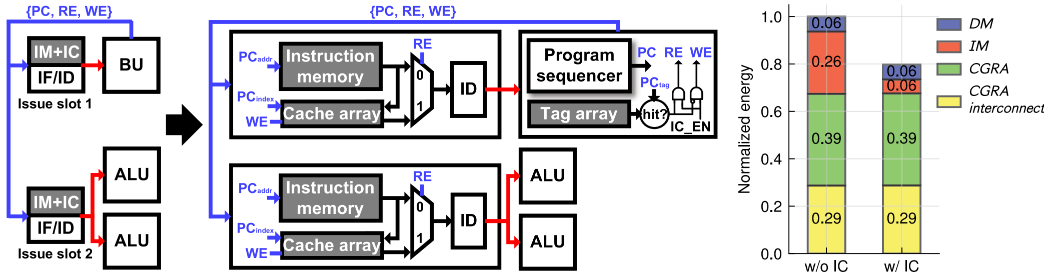


Fig. 4. R-Blocks instruction delivery organization with **instruction cache (IC)** that supports a reconfigurable number of issue slots (left) and normalized energy consumption of *matmul* benchmark on R-Blocks CGRA-L instance[2] with and without IC (right).

loader into the CGRA. This PC signal is broadcast to all issue slots over the reconfigurable data network, i.e. the instruction fetch and decode units (IF/IDs). To reduce the instruction fetch energy overhead for data-parallel applications, the decoded instructions from IF/IDs are broadcast over the reconfigurable control network to multiple FUs. This approach can reduce the system-level energy by 22% on average [62]. To further reduce the instruction delivery energy, CGRAs often have a small context memory, loop buffer, or instruction buffer [12, 29, 48, 53] to support loop acceleration for a pre-defined maximum loop size or initiation interval (II). However, for CGRAs that target acceleration of complete applications and support a VLIW style of execution, like [30] and R-Blocks, a more generic solution is required. As such, we implement a direct-mapped instruction cache that supports a CGRA with a reconfigurable number of issue slots, inspired by [53] and illustrated in Figure 4 (left). The tag array and cache logic are implemented in the BU and reused by all issue slots in a virtual core. Besides the PC signal, two additional control signals, i.e. **write enable (WE)** and **read enable (RE)**, are used to fetch from the IMs, copy instructions to the cache arrays in case of a miss, and to fetch from the cache arrays in case of a hit. The instruction cache coherence is software-managed using the CGRA configuration interface. Prior to loading a new program, the CGRA must be in an idle state. After loading the program, the old cache content will be invalidated.

4.2 Vector Memory Subsystem

into R-Blocks several FUs can be combined into a 'virtual' vector unit using instruction broadcasting, which allows us to design VLIW-SIMD processors. As such, these cores typically consist of a scalar and vector datapath, each with its own memory interface. Some LSUs are connected to GM, while other LSUs connect to private LM units over the reconfigurable data network. To communicate between the scalar and vector datapath, scalar values can be broadcast to vector units over the data network. This feature is commonly used for vectorized load/stores and operations with a scalar and vector operand. More complex communication patterns, such as inserting/extracting vector elements, use the so-called vector memory subsystem, which is accessible by both the scalar and vector datapath.

The vector memory subsystem consists of N 32-bit memory banks with N+1 LSUs; The scalar memory interface is used for data transfers between GM and the local vector memory, while the other N LSUs operate as a vector memory interface, as illustrated in Figure 5 (top). Due to the absence of dedicated SIMD logic in the R-Blocks fabric, the vector memory interface only supports aligned vector accesses and does not provide scatter-gather support to other LM units. In practice the scalar memory interface copies data to the local vector memory, whereafter the vector datapath processes on this data. In the current system, the scalar memory interface is used by the scalar

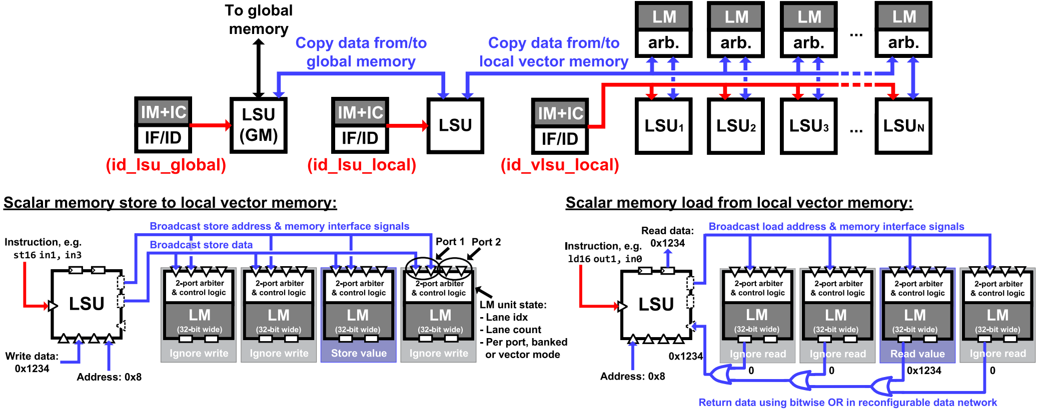


Fig. 5. R-Blocks vector memory subsystem (top) and example of load and store using the scalar memory interface (bottom).

datapath, but in the future this part could be optimized using a dedicated **direct-memory access (DMA)** controller to hide the latency of memory transfers.

Each LM unit on the CGRA fabric consists of a memory macro, a two-port arbiter, and reconfigurable state registers. These state registers store the lane index and the total numbers of lanes or lane count of the 'virtual' vector memory. Each of the two arbiter ports can operate in banked or vector mode. In banked mode, an LSU is connected to multiple LM units. A location in the local vector memory is defined by a mapping from byte-address to a tuple of (*lane index*, *line index*, *byte index*). The storage scheme for banked mode, where the LSU is connected to multiple local memory banks, is calculated in the LM unit as follows (assuming a 32-bit LM bank width):

$$lane\ index = \lfloor addr/4 \rfloor \bmod L \qquad line\ index = \lfloor addr/(4 \cdot L) \rfloor \qquad byte\ index = addr \bmod 4$$

for a given load/store address *addr* and local vector memory with L lanes. This scheme can be efficiently implemented using bit-slicing, given that L is a power of 2. When a store is performed in banked mode, the scalar memory interface broadcasts the data and address to all connected LM units. The stored lane index is compared with the lane index in the address, and the store operation is performed in the bank where the id matches. Similarly, read requests are also broadcast to LM units. The LM unit with a matching lane index in the load address outputs the data in the next cycle, while the other lanes output zero. The output values from all lanes are combined using bitwise OR-reduction, as shown in Figure 5 (bottom). This OR-reduction functionality is implemented in the reconfigurable data network of R-Blocks, where switchboxes incorporate 2-to-1 bitwise OR functionality, as depicted in the switchbox structure in Figure 1.

The LSUs of the vector memory interface are connected to an LM unit port that is configured in vector mode, the translation from a load/store address to a memory bank address is simple, and is defined as follows:

$$lane\ index = always\ valid \qquad line\ index = \lfloor addr/L \rfloor \qquad byte\ index = addr \bmod 4$$

In future iterations of R-Blocks we envision multiple virtual cores running on the R-Blocks CGRA fabric in parallel. The functionality of the LM units could be extended with support for local inter-core communication. For example, the LM units could be extended with support for FIFO functionality.

4.2.1 *Fast Data Copies from/to Vector Memory.* In applications that operate on vector data types with elements smaller than the native LM bank width (32-bit words), such as bytes

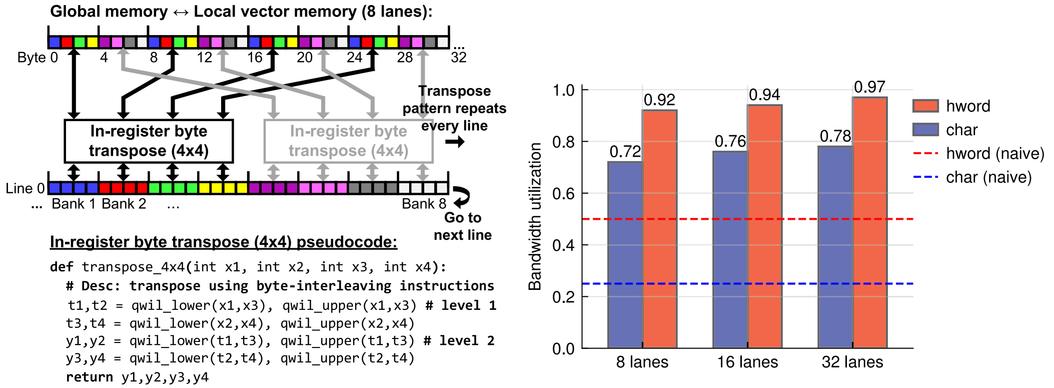


Fig. 6. Fast data copies between GM and local vector memory using byte-interleaving instructions for an 8-lane virtArch (left) and GM bandwidth utilization of 2 kB memcpy to vector memory with a varying number of lanes, compared to a naive approach (right).

and hwords, it is necessary to transfer data between GM and the local vector memory in an interleaved/de-interleaved manner. This prevents consecutive vector elements from being placed in the same bank. A naive way of copying these elements using the 32-bit GM interface would be limited to 1 byte/cycle or 1 hword/cycle, lowering the GM bandwidth utilization to only 25% or 50%, respectively. However, by performing an in-register data layout transformation, we can sustain a much higher data copy bandwidth. We present a solution to this problem that achieves close to peak GM bandwidth, independent of the number of vector lanes (powers of two, larger or equal to the bank width, i.e. 4 in our case, are supported), using a fixed data shuffle pattern for both interleaving/de-interleaving of byte and hword arrays. We extend the R-Blocks ALU FUs with four extra instructions, i.e. `qwil_upper/lower` for byte interleaving of two 32-bit operands and `hwil_upper/lower` for hword interleaving of two 32-bit operands, similar to the approaches described in [39, 46]. Figure 6 illustrates the memcpy implementation using byte-interleaving instructions, as well as some benchmarking results of performing a 2 kB memcpy between GM and the vector local memory. For an 8-lane machine (similar to Figure 5), we achieve a sustained bandwidth utilization of 72% for bytes and 92% for hwords. Increasing the number of lanes reduces the loop overhead and improves the utilization to a near-optimal 78% for bytes (the minimum initiation interval is five cycles per four words) and 97% for hwords.

4.3 Inter-lane Communication

In order to efficiently perform data permutations in vectors without relying on the vector memory interface to emulate shuffles in memory, we extend the ALU FU with additional ports for adjacent lane communication, as is depicted in Figure 7. This approach is inspired by the nearest-neighbor network presented in [56]. While arbitrary shuffle patterns can be emulated in software using adjacent lane communication, it would be excessively costly. However, commonly used shuffle patterns such as 1D/2D sliding windows and image boundary handling can be efficiently executed using adjacent lane communication. Even commonly used reduction patterns can be effectively performed, as demonstrated in [55]. To support these operations, we extend the ALU FU with four instructions: `roel` and `roer` for rotating vector elements left or right, and `selra` and `serla` for shifting elements left or right while appending a scalar operand at the start of end of the vector. The `roel` and `roer` instructions move the `opA` operand from the adjacent lanes to an output port register, whereas the `selra` and `serla` instructions will either forward operand `opA` from adjacent lanes or append a scalar operand `opB` if it's the first or last lane in the vector.

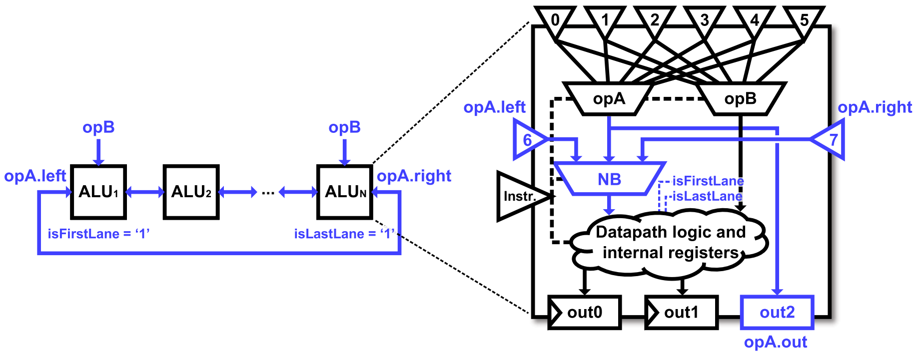


Fig. 7. Extensions to ALU FU (in blue) to support adjacent lane communication in R-Blocks.

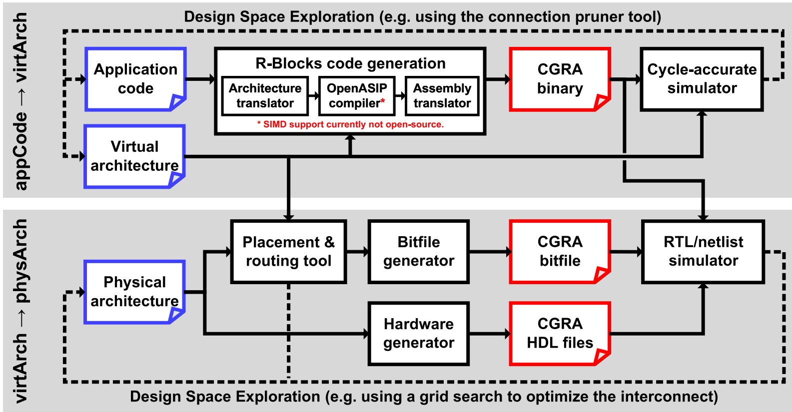


Fig. 8. R-Blocks code generation and hardware generation tool-flow.

5 R-BLOCKS HW/SW GENERATION TOOL-FLOW

To generate efficient code and hardware for the R-Blocks architectural template, an overview of the HW/SW generation tool-flow is presented in this section. This tool-flow maps high-level **application C-code (appCode)** to the R-Blocks CGRA fabric, and presents a framework for hardware generation and **design space exploration (DSE)**. For the code generation aspects in this paper, we extend the open-source OpenASIP toolset [27] with an architecture translator to convert to the OpenASIP **architecture description file (ADF)** format, assembly translator to convert OpenASIP assembly to R-Blocks assembly, and modified the OpenASIP back-end with some R-Blocks specific instruction patterns. Figure 8 depicts the overall structure of our HW/SW generation tool-flow. The mapping process is performed in two primary steps: appCode(s) —(1)⟶ virtArch(s) —(2)⟶ physArch, i.e.

(1) Application C-code is compiled to assembly based on an XML-based R-Blocks virtArch description, which contains the processor structure and other architectural details. The generated CGRA binary can be simulated using a cycle-accurate simulator. After this step, the virtArch can be further optimized by increasing the computational resources to exploit more ILP and DLP, reducing the number of bypass connections to minimize the instruction size and interconnect complexity, and modifying the application code to increase performance. The fast OpenASIP compiler and cycle-accurate simulator allows us to quickly explore many design points.

(2) After finding a suitable virtArch, the resulting virtArch instance is mapped onto the fabric or physArch. This step is performed using a heuristic-based placement and routing tool. In this step, logical instruction memories and FUs are mapped onto physical instances and connections are routed over the reconfigurable switchbox networks. When the placement and routing tool fails to find a valid mapping, the available computational or interconnect resources in the physArch can be increased, or the virtArch resource usage could be reduced.

5.1 Designing and Optimizing Virtual R-Blocks Cores

5.1.1 OpenASIP Toolset & Compiler. The OpenASIP toolset can be used to design and program customized processors based on an energy-efficient **transport-triggered architecture (TTA)** template. TTAs are a kind of processor that can model multi-issue VLIW processors with a software-controlled datapath. Different from traditional **operation-triggered architectures (OTA)**, in TTAs the instructions represent the data transport; operations are executed as a side effect of these data transports. This programming model allows for more scheduling freedom, compared to the more conventional OTAs. The open-source OpenASIP toolset provides a customizable TTA-based processor template, simulator, and retargetable C-compiler. Since the OpenASIP processor template is very flexible, it allows us to model the execution model of the R-Blocks CGRA and generate efficient code for arbitrary virtArchs that all execute on the same physArch. A major advantage of the OpenASIP toolset is that it can deal with EDPAs with distributed RFs and sparsely connected datapaths, as is the case for R-Blocks which lacks a centralized RF, FUs can directly communicate to other FUs, and FUs do not always have a direct connection to an RF. More details about TTAs and OpenASIP can be found here [11, 27, 28]. In this work, we use an internal version of OpenASIP which includes a SIMD back-end, as was presented in [50].

5.1.2 Architecture Translation & Code Generation. We model the R-Blocks virtArch as a TTA core with VLIW connectivity where every FU operand port has its own private bus, see e.g. [51], and a fully-connected operand bypassing network to all other FUs and RFs. The centralized RF bottleneck in traditional VLIW processors is being avoided by having multiple distributed 1R/1W RFs. R-Blocks FUs are directly translated to TTA FUs with the same operation set and registered output ports. The R-Blocks virtual SIMD units and connectivity are translated to dedicated SIMD units and vector buses in the TTA core, using the template presented in [50]. To increase the scheduling freedom, all FUs support a `copy` operation, which allows the OpenASIP scheduler to move values to output registers of idle FUs to reduce the amount of unit-blocking, where an FU is blocked until the output buffer value is consumed. After code generation, the generated TTA assembly is converted back to R-Blocks assembly. To do so, we must ensure that all operand moves and the trigger of a single operation are scheduled in the same cycle. This scheduling constraint is easily enforced without changing the OpenASIP compiler, as the TTA cores derived from R-Blocks virtArchs do not support FU input port buffering and all FU operand moves can always be scheduled within the same cycle due to the VLIW connectivity.

5.1.3 Design Space Exploration. A good starting point for a processor designer is to start with a virtArch with similar resources to a single-issue RISC core, consisting of an ALU, MUL, LSU, BU, and one or two RF units. At this point, we put no restriction on the number of bypass sources per FU to ease code generation; our cycle-accurate simulator can simulate an arbitrary number of bypasses. From this design point, you could expand the FU mix and prune the connectivity to create an efficient multi-issue VLIW core. Additionally, the processor designer can vectorize the code using SIMD intrinsics and add virtual SIMD units to the virtArch of any vector width, as shown in Figure 2. Furthermore, local scratchpad memories (LM units) and FUs with custom instructions

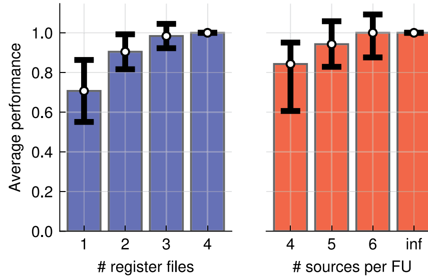


Fig. 9. Average performance (and minimum and maximum performance in error bar) of 15 benchmarks on the CGRA-L physArch with a varying number of RFs (left) and with a different number of FU bypass sources (right). In the final implementations, we generally use two RFs for most benchmarks and more for a few with a lot of state (*conv1d*, *vg*). It turns out that six FU bypass sources are sufficient to obtain performance close to the 'optimal'. Note that the heuristic-based OpenASIP compiler is sometimes able to find better solutions for constrained architectures, compared to a virtArch with full bypass connectivity.

can be added and simulated as well. The quality of the designed virtArch can be quickly evaluated using the simulator and a first-order area and energy model [61].

Besides computational resources, the number of RFs and connectivity between FUs are important parameters to achieve high performance and an efficient implementation. To illustrate this point, Figure 9 (left) depicts the impact of the number of RFs in a virtArch with full bypass connectivity on the average application performance for the 18-issue CGRA-L architecture that will be introduced in Section 6.1.1. As can be seen from the figure, having just a single 1R/1W RF for a multi-issue VLIW-SIMD core will cripple the average performance to only 70% of the 'optimal' performance. However, adding a second RF puts most applications already within 10% of their 'optimal' performance, after which adding more RFs only has a limited effect on performance. This highlights a primary advantage of EDPAs compared to a traditional VLIW core with centralized RF; most of the RF accesses can be bypassed using FU output buffers, as will be evaluated in Section 6.2.2. While a virtArch with full bypass connectivity eases DSE, the number of FU bypass sources needs to be reduced to the maximum supported by the physArch, a design-time constant. Increasing this constant increases the instruction width, FU operand multiplexer size, and possibly the number of tracks in the interconnect. Therefore, this should be avoided if possible. To automate the tedious process of finding a solution that fits on the physArch, we designed an iterative connection pruner tool that greedily removes FU bypass sources with the lowest impact on performance until the bypass sources of every FU are pruned to fit the physArch. From Figure 9 (right) it follows that having only six bypass sources per FU approaches the 'optimal' performance compared full bypass connectivity. The current pruner is somewhat slow; it requires up to several hours for a single virtArch. However, in practice this pruning step is only required once, as a last step before the optimized virtArch is mapped to the R-Blocks fabric.

5.2 Mapping Virtual Cores to the R-Blocks Fabric

5.2.1 Placement & Routing. Mapping a virtArch onto the physArch is performed using a heuristic-based placement and routing tool. We follow a conventional approach that is widely adopted by the FPGA and CGRA community [5, 21, 29, 32, 33, 37, 60], of using simulated annealing [6] for placement and the Pathfinder [21, 36] algorithm for routing. Other, more optimal mapping approaches based on **integer programming (IP)** or boolean **satisfiability (SAT)** exist [10, 22, 23], but these generally take significantly more time. Depending on the routing complexity, our heuristic-based router finds a solution within several minutes, given that all

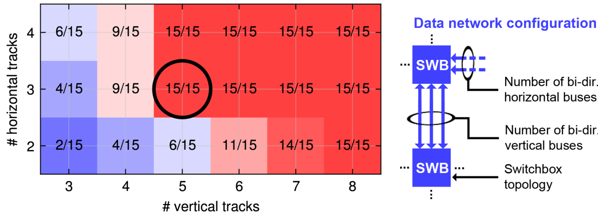


Fig. 10. Routability of 15 benchmarks on the CGRA-L physArch for data network with a different number of horizontal/vertical tracks. All implementations assume a Wilton switchbox with connectivity degree 3 and fully-connected connection box to the FU input and output ports. The circle indicates the configuration that was used for the experiments in Section 6.

resource conflicts can be resolved. If resource conflicts remain, either the available computational or interconnect resources in the physArch must be increased, or the virtArch resource usage and connectivity must be reduced.

5.2.2 Hardware Generation. A hardware implementation is automatically generated based on an XML-based physArch description file, detailing the location and types of all CGRA resources, as well as the interconnect topology. The supported instructions with the corresponding control word for every FU type are listed in an XML-based ISA description file. The control words for every instruction are encoded in a variable-length opcode based on a Huffman tree to minimize the FU instruction width, using a tool that automatically generates the instruction decoder hardware. The FU hardware is manually implemented using a HDL template that follows the standardized interface as depicted in Figure 1. The interconnect hardware is generated using an interconnect generator, which has options for the switchbox topology, as well as the number of horizontal and vertical tracks for both the reconfigurable control and data networks.

5.2.3 Design Space Exploration. We perform an automated grid search on the required number of tracks in the reconfigurable control and data networks. Figure 10 plots the routability of all 15 benchmarks that will be introduced in the next section for a different number of horizontal and vertical tracks in the data network. It follows that three horizontal tracks and five vertical tracks is the smallest network configuration that can support all benchmarks. Note that the control network (not shown) requires fewer routing resources; onlytwo horizontal tracks and one vertical track are required.

6 EXPERIMENTAL RESULTS

This section provides experimental results on performance, energy- and area efficiency. We compare the CGRAs against highly tuned benchmarks on a state-of-the-art RISC-V core, as well as ASIPs to evaluate the cost of reconfigurability.

6.1 Experimental Setup

We perform our evaluation using three reference architectures: 1) an open-source scalar RISC-V processor[3], comparable to an ARM Cortex-M4; a popular choice for low-power embedded signal processing (RV32IM), 2) a small CGRA with fixed datapath whose processing resources roughly correspond to the RISC-V core (CGRA-S), to quantify differences between the RISC-V and OpenASIP compilers and benefits of software bypassing, and 3) a large CGRA with reconfigurable

[3]The RI5CY core from the PULPino platform (https://github.com/pulp-platform/pulpino) with a private IC (https://github.com/pulp-platform/hier-icache).

Table 1. Summary of the Baseline RISC-V Processor and R-Blocks CGRAs

	Baseline	**R-Blocks CGRA**	
Component	RV32IM	CGRA-S	CGRA-L
GM (global memory) [kB]	64	64	64
LM (local memory) [kB]	-	-	9
IM (instruction memory) [kB]	32	6.9	17.1
IC (instruction cache) [kB]	1	0.9	2.1
#ALU	1	1	10
#LSU	1	1+0	1+9[1]
#MUL	1	1	9
#RF	1[2]	2[3]	18[3]

[1] 1 LSU is connected to GM, 9 LSUs are connected to LM units. [2] 32 × 32-bit 3R/2W register file (extra read/write ports required for DSP extensions). [3] 16 × 32-bit 1R/1W register file.

datapath (CGRA-L) to investigate the energy efficiency benefits from parallel processing. Additionally, ASIPs with a fixed datapath are instantiated per benchmark application based on the CGRA-L mappings to quantify the cost of reconfiguration. Each ASIP directly implements the R-Blocks virtArch by having fixed wiring for instruction distribution and data communication, instead of the reconfigurable interconnect. Units that share the same instruction stream are merged into dedicated SIMD units with an application-specific number of vector lanes and unused units are removed. CGRA-S, CGRA-L, and the ASIPs are all instantiated using the R-Blocks framework.

Other popular parallel processing platforms for ULP signal processing are RISC multi-core processors, e.g. [4, 18, 44], and vector processors, e.g. [9, 24, 54, 56]. However, multi-cores do not improve energy efficiency over a single-core solution without further optimizations, such as voltage-frequency scaling [44] and ISA extensions [9], which would also benefit R-Blocks. Vector processors specialize towards data-parallel workloads, but often suffer from a major vector RF area and energy bottleneck [24, 54, 56], and are unable to match the issue width and vector width depending on the application requirements, which was evaluated in [9, 63].

6.1.1 Reference Architectures. All architectures (including the RISC-V processor) contain a single-ported SRAM-based GM of 64 kB, large enough to contain the input and output data of all benchmarks. The program of the RISC-V processor is initially stored in a 32 kB single-ported SRAM-based instruction memory (IM), and a 256-word direct-mapped instruction cache (IC) is utilized to reduce the instruction fetching energy overhead. The CGRAs use a 512-line SCM-based instruction memory and a 64-line direct-mapped instruction cache (line size depends on the number of issue slots in a virtArch). In general, the instruction words of the CGRA are significantly larger than the RISC-V core's 32-bit instructions. Additionally, CGRA-L and the ASIPs have multiple 1 kB LM units that can be used for energy-efficient and parallel access of data with sufficient reuse. To make the comparison as fair as possible, we have experimented with adding a small data cache or scratchpad memory (≈1 kB) to the RISC-V processor as well. However, we were unable to save energy due to a sub-optimal hit rate (≈80%), which led to a significant performance and energy penalty. The RISC-V processor and R-Blocks CGRA configurations are summarized in Table 1.

The instantiated R-Blocks CGRA (CGRA-L) that is reused for all benchmarks contains a superset of all required FUs for the ASIPs with fixed datapath; its structure is visualized in Figure 11. The architecture essentially scales the smaller CGRA-S architecture up with sufficient resources for an additional 8-lane vector datapath, local memories for fast parallel access and energy-efficient processing, and reconfigurable switchbox networks for both instructions and data. Once a

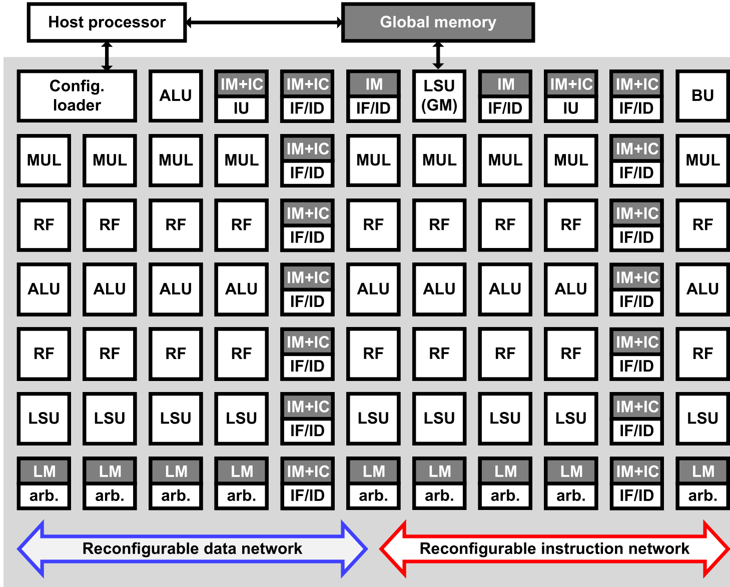


Fig. 11. R-Blocks architecture instance (CGRA-L) that was used for the evaluation. For clarity, the switchbox networks and connections between units are not drawn.

configuration context (bitfile and/or program binary) is loaded into the fabric by the host processor, the CGRA operates in standalone mode. For all benchmarks the input data is preloaded in the shared GM, so data copies to local memories are part of the execution time and energy consumption. All R-Blocks architectures support up to six bypass sources per FU or RF, and the reconfigurable architecture has 2/1 (horizontal/vertical) 16-bit bidirectional tracks for the control network, and 3/5 32-bit bidirectional tracks for the data network. Both networks use area-optimized Wilton switchboxes [6] with switch block flexibility $F_s = 3$ and connection block flexibility $F_c = 100\%$, as shown in Figure 1.

6.1.2 Implementation. All architectures are implemented in RTL and synthesized in a commercial 22-nm FD-SOI technology (including compiled SRAM and SCM memories) using an 8-track regular V_{th} (RVT) library and Cadence Genus 21.10.000 in the worst-case corner (100 MHz, 0.72 V, -40 C). Power analysis was performed using back-annotated netlist simulations using Cadence Incisive 15.20.038 in the typical corner (100 MHz, 0.80 V, 25 C). We perform post-synthesis simulations to validate functional correctness and retrieve the execution time and switching activity for power estimations. For CGRAs with the switchbox interconnects we have to cut the combinational loops in the interconnect and manually constrain timing paths to allow for static timing analysis.

The maximum operation frequency (post-synthesis) of the R-Blocks CGRA (CGRA-L) excluding the reconfigurable switchbox interconnect delay is approximately 425 MHz. The timing bottlenecks are the instruction memories and the multiplier FU latency of 2.3 ns. The switchboxes add an additional delay of approximately 0.25 ns per hop (the critical path of the largest switchbox in CGRA-L). The interconnect reduces the operation frequency based on the maximum number of hops for a given workload mapping. For the evaluated benchmarks on CGRA-L, the maximum number of hops was between 10 and 14, which reduces the operating frequency to approximately 206 MHz and 170 MHz, respectively. The maximum operation frequency of R-Blocks can be significantly increased by adding pipeline registers to the interconnect, similar to [8, 23, 29],

Table 2. Overview of Evaluated Benchmarks

Benchmark	Description
matmul	32×32 16-bit fixed-point MatMul
conv1d	2184-point 8-tap 32-bit integer FIR filter
conv2d	128×64 image 8-bit 3×3 Gaussian blur filter
fft	256-point 16-bit fixed-point complex FFT
binarization	thresholding on 128×64 gray-scale image
erosion	3×3 logic-AND filter on 128×64 binary image
projection	summation of each row and column on 128×64 binary image
ffos	pipeline of *binarization*, *erosion* and *projection* as part of vision application
bpf	fixed-point 5-stage biquad band-pass filter on 2×128 16-bit samples
dwt	fixed-point filter-based db4 wavelet decomposition on 2×256 16-bit samples
apen	index mergesort followed by vector comp. loop with early exit on 2×256 16-bit samples
sampen	index mergesort followed by vector comp. loop with early exit on 2×256 16-bit samples
vg	16-bit NVG/HVG Node Degree slope-following D&C algorithm on 2×256 16-bit samples
mfcc	16-bit fixed-point Mel-Frequency Cepstral Coefficient (MFCC) pipeline, based on [20], on 256 16-bit audio samples (16 ms), repeated 99× (10 ms overlap/window, 1 s audio clip) for *cnn* input.
cnn	8-bit/16-bit (weights/activations) fixed-point CNN (2 convolutional and 2 fully-connected layers) with Max-pooling and ReLU activations on *mfcc* output for keyword classification.

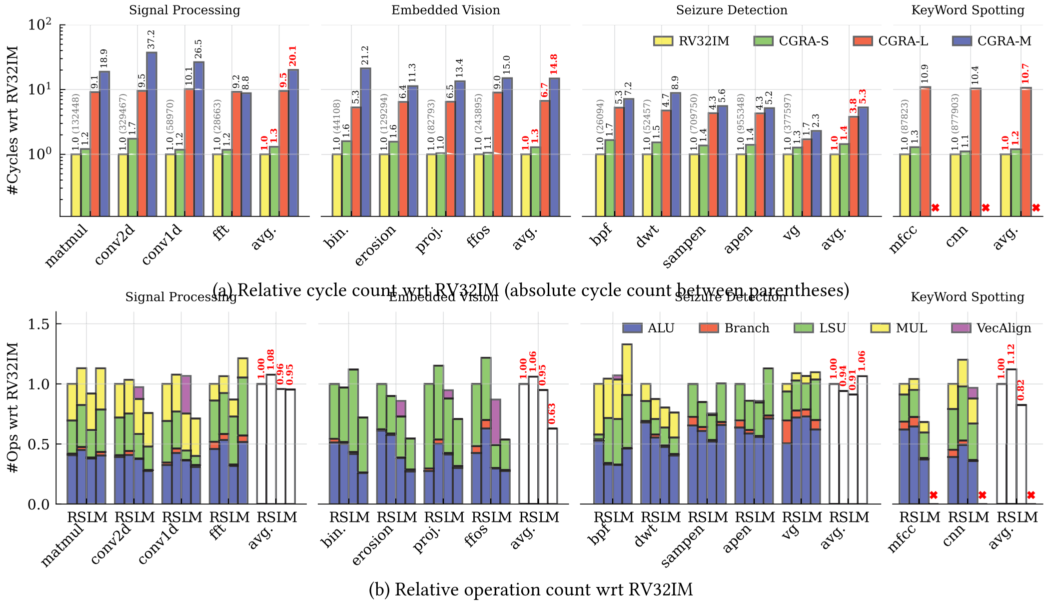
which we leave for future optimization. The RISC-V processor is able to run up to approximately 475 MHz before the LSU to GM becomes the bottleneck. For a fair comparison, we evaluate the RISC-V processor, R-Blocks CGRAs, and R-Blocks ASIPs at a relaxed operation frequency of 100 MHz.

6.1.3 Benchmarks. We evaluate the reference architectures on a set of benchmarks from the signal processing domain and three representative data-driven applications in the embedded/e-Health/IoT domain, as is detailed in Table 2. The aim of this benchmark set is to showcase the performance and energy benefits of R-Blocks for application-scale acceleration, rather than only considering small inner loops of kernels. The considered benchmarks are:

— **Signal Processing:** four common signal processing benchmarks that are used in many embedded applications: *matmul*, *conv1d*, *conv2d*, and *fft*.
— **Embedded Vision:** four benchmarks from an industrial embedded computer vision application [25, 56]: *binarization*, *erosion*, *projection*, and *ffos*.
— **Seizure Detection:** five benchmarks from an embedded EEG-based NCSE seizure detection application [15]: *bpf*, *dwt*, *sampen*, *apen*, and *vg*.
— **Keyword Spotting”** two benchmarks from a keyword spotting application [20] trained on the Google Speech Commands dataset: *mfcc*, and *cnn*.

All three applications could potentially execute in real-time on an embedded RISC processor, like the RV32IM platform. However, the architectural benefits of the R-Blocks CGRA, i.e. the parallel and flexible VLIW-SIMD execution model with software bypassing, in combination with voltage-frequency scaling makes R-Blocks an attractive parallel processing platform for throughput- or energy-constrained applications.

6.1.4 Compilation & Mapping. For the RISC-V core, we compile the benchmarks using the RV32IM instruction set with GCC 5.2.0 and -O3 and -funroll-loops optimization flags. The R-Blocks C-compiler is based on an internal version of the OpenASIP compiler tool-flow with support for SIMD-processing, and is compiled with LLVM 14.0. For CGRA-S and CGRA-L,



(a) Relative cycle count wrt RV32IM (absolute cycle count between parentheses)

(b) Relative operation count wrt RV32IM

Fig. 12. Performance of CGRA-S and CGRA-L compared against the RV32IM baseline and CGRA manual mappings (CGRA-M). The performance of the manual mappings is derived from [15, 63]. ✕ indicates that an implementation was not available.

the C-code is annotated with Clang pragmas to unroll hot loops and compiled using the -O3 optimization flag. Additionally, CGRA-L benchmarks are manually vectorized using intrinsics. Also, we included handwritten assembly mappings for most CGRA benchmarks to evaluate the code quality of the current compiler (CGRA-M).

For a fair comparison with the RISC-V core, the R-Blocks instruction set is based on a standard 32-bit RISC operation set (very similar to RV32IM). One difference is that the CGRA compiler has support for if-conversion and a conditional move instruction. Also, byte/hword interleaving instructions to facilitate fast data copies between GM and local vector memory were added (Section 4.2.1), as well as the SIMD-specific instructions for inter-lane communication (Section 4.3). Both the RISC-V core and CGRAs also support DSP extensions and hardware loops, but the current R-Blocks compiler cannot exploit those, which is why these features were disabled for all architectures.

6.2 Performance Evaluation

We evaluate the speedup of the R-Blocks CGRAs compared to the RISC-V processor in terms of cycle count, and include the number of executed operations to emphasize the similarities between the mappings. Second, we discuss some inefficiencies in code generation between RV32IM and R-Blocks architectures by comparing the mapping quality. Finally, we quantify the impact of software bypassing and instruction broadcasting, and conclude with an in-depth discussion on where we could further enhance the compiler by investigating manually implemented assembly mappings.

6.2.1 Speedup & Operation Breakdown. Figure 12(a) plots the cycle count improvements of the compiler-generated mappings on CGRA-S and CGRA-L compared to the RV32IM baseline. For each group of benchmarks, we compute the geometric mean. At the same operation frequency, CGRA-S is 1.2-1.4× faster on average compared to the RISC-V processor, depending on the application

Table 3. Computational Resources used by Different Benchmarks on CGRA-L Instance

Benchmark	#LM	#ALU	#LSU	#MUL	#RF	DLP[1]
matmul	8	10	1+9	9	18	8
conv2d	8	10	1+9	9	18	8
conv1d	0	9	1+0	8	12	8
fft	8	10	1+9	8	18	8
binarization	8	9	1+9	0	10	8
erosion	8	10	1+9	0	18	8
projection	8	10	1+9	0	10	8
ffos	8	10	1+9	0	10	8
bpf	2	3	1+3	4	6	2
dwt	2	7	1+3	4	6	2
sampen	4	7	1+5	0	6	2
apen	4	7	1+6	0	6	2
vg	2	3	1+2	1	3	1
mfcc	8	10	1+9	9	18	8
cnn	8	10	1+9	9	18	8
available (Table 1)	9	10	1+9	9	18	8

[1]Vector width of virtual SIMD units used in the mapping.

domain, while executing a similar number of operations for both implementations: 0.94-1.12×, as is shown in Figure 12(b). The primary reason for this performance degradation is that the 4-stage RISC-V core contains a significant number of data hazards, which reduces the average **instructions per cycle (IPC)**. The CGRA-L mappings are 3.8-10.7× faster on average compared to the RV32IM baseline. The benchmarks in the Signal Processing, Embedded Vision, and Keyword Spotting applications use virtArchs that utilize roughly all resources, while the Seizure Detection benchmarks utilize approximately half of the computational resources available on the fabric, following the manual mappings as presented in [15, 63]. For reference, the resource usage for every benchmark is summarized in Table 3. The FU mix of each virtArch was manually selected based on the application characteristics, but this process could be automated in the future, similar to [49]. However, the FU bypass connectivity of every virtArch was automatically constrained using the connection pruner tool that was introduced in Section 5.1.3.

The performance scaling of *matmul* and *conv2d* on CGRA-L is limited by the data transfer overhead between GM and LM units. A significant portion of the cycles is spent on data loading (≈30%) from/to GM, as all data movement is managed by the cores themselves. Smaller kernels like *binarization*, *erosion* and *projection* face a similar problem, but it follows from *ffos* that this issue is alleviated for larger applications with more data reuse via LM units. The Seizure Detection and Keyword Spotting benchmarks suffer less from this problem as they are compute-bound.

It should be emphasized that the instruction sets of the RISC-V core and R-Blocks are very similar, and the benchmarks were implemented using equivalent C-code with aggressive loop unrolling to maximize performance. This also follows from the number of executed operations in Figure 12(b), which in general is very comparable between RV32IM, CGRA-S, and CGRA-L. If we consider the code density or utilization of the CGRA-L mappings, computed using the number of executed operations of the number of active FUs divided by the peak number of operations, i.e. $\texttt{Utilization} = \texttt{Ops}/(\texttt{\#FUs} \cdot \texttt{Cycles})$, it ranges from 24% to 54% (35% on average), which is competitive with other CGRAs [42].

Despite these efforts to make the mappings between platforms as comparable as possible, there are some interesting differences between the RV32IM and CGRA-S mappings, which highlight opportunities for further improvements in code generation. One notable inefficiency is that the current R-Blocks instruction format doesn't support memory and control flow operations with three operands (or an immediate operand). Additionally, some code generation differences between RV32IM and CGRA-S are also visible. We make the following observations:

— Memory operations with an immediate offset field are not supported, which results in additional address calculation operations. This issue is visible for the memory-dominated *projection* benchmark.
— Branching operations with a relative (immediate) offset field are not supported, which results in additional address calculation operations for every branch. This bottleneck is visible in the branch-heavy *vg* benchmark.
— The RV32IM mapping of *bpf* stores the filter delay line in an RF, while the R-Blocks compiler does not identify this opportunity, which leads to more memory operations.
— The RV32IM mappings of *sampen*/*apen* execute significantly more branching operations, as the R-Blocks compiler applies if-conversion due to the support of a conditional move instruction.

Overall, most of these inefficiencies (except for the if-conversion) favor the RV32IM code but can be used as guidelines to further improve R-Blocks CGRA code generation in the future. When comparing the number of executed operations in Figure 12(b) between CGRA-L and CGRA-S mappings, we make the following observations:

— Fewer ALU and branching operations are being executed for mappings that exploit SIMD-processing (e.g. *fft*, *mfcc*, and the Embedded Vision benchmarks), as these operations are only executed on the scalar datapath.
— Fewer LSU operations are being executed due to scalar broadcasts to the vector datapath (e.g. *matmul*, *convolution*, and *cnn*) or the use of vector manipulation operations (VecAlign) to prevent reloads (e.g. *conv1d*).
— Explicit data copies between GM and LM units result in more LSU operations, which is especially visible in the memory-dominated benchmarks with limited compute like *binarization* and *erosion*.
— While the VecAlign operations are used in 9/15 benchmarks, a significant increase in executed operations is generally avoided. Exceptions are *conv2d*, *conv1d*, and *erosion*, which use VecAlign operations for sliding windows to trade LSU operations for RF accesses. Furthermore, *ffos* scatters the vector results back to GM using the scalar memory interface, which is costly.

6.2.2 Impact of Software Bypassing. Software bypassing allows the compiler to directly schedule data transfers between FUs, which is essential to reduce the centralized RF energy bottleneck found in traditional vector processors [7, 24, 54]. Figure 13 depicts the normalized number of RF accesses for the different benchmarks. Compared to the RV32IM baseline, the CGRA-S core is able to reduce the number of RF accesses to 0.43-0.60× on average, depending on the application domain. These results highlight the advantage of processors with support for explicit bypassing. The number of RF accesses between RV32IM and CGRA-L is even further reduced to 0.23-0.43× on average, depending on the application domain. We identify several reasons why RF accesses are further reduced for CGRA-L over CGRA-S:

— Load/store addresses and filter coefficients are reused by vector operations with a scalar operand.

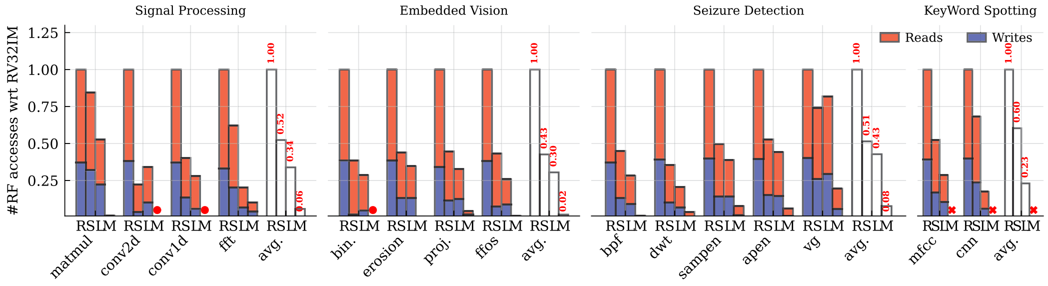


Fig. 13. Normalized RF accesses of CGRA-S (S) and CGRA-L (L) compared against RV32IM baseline (R) and CGRA manual mappings (M). The RF accesses of the manual mappings are derived from [15, 63]. ✕ indicates that an implementation was not available. ● indicates that there were no RFs in the implementation.

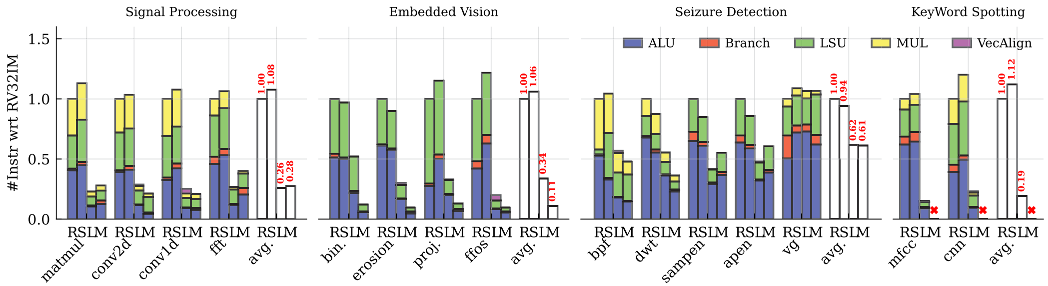


Fig. 14. Normalized number of executed instructions of CGRA-S (S) and CGRA-L (L), compared against RV32IM baseline (R) and CGRA manual mappings (M). The number of executed instructions of the manual mappings are derived from [15, 63]. ✕ indicates that an implementation was not available.

— Address calculations on the scalar datapath can be computed in parallel with the vector operations, which allows the compiler to schedule the address calculations as late as possible and bypass the RF.
— Scalar loads and vector loads can be performed in parallel, which allows the compiler to schedule both as late as possible and bypass the RF for some benchmarks such as *matmul* and *cnn*.

Exceptions are *conv2d*, *conv1d*, and *vg* where the number of RF accesses increases, or reduces less than expected. *conv2d* and *conv1d* reduce memory operations for an increase in RF traffic to keep the sliding window part of the filters in the RFs, which we consider a good choice, as also is visible in Figure 12(b). The main loop in *vg* consists of a very large code segment, which leads to high register pressure.

6.2.3 Impact of SIMD-processing. The reconfigurable control network of R-Blocks allows construction of vector datapaths with an application-specific number of lanes. Figure 14 plots the normalized number of fetched instructions compared to RV32IM. The benefits of SIMD-processing on the R-Blocks CGRA become apparent when we consider the number of fetched instructions for the CGRA-L implementations. On average, the number of executed instructions of CGRA-L is 0.19-0.62× the RV32IM baseline. The Seizure Detection benchmarks benefit less because these mappings process only 2 lanes in parallel while utilizing more FUs (i.e. more ILP) for a single lane, compared to the other benchmarks which have a vector datapath with up to 8 lanes. One notable exception is *vg* which doesn't exploit any SIMD-parallelism due to its data-dependent loops.

6.3 Compiler vs. Manual Mappings

In the previous section, we evaluated the performance of the compiler-generated mappings between different platforms. In this section, we investigate the performance differences of parallel CGRA mappings between the compiler-generated (CGRA-L) and handwritten assembly (CGRA-M). It should be emphasized that the assembly mappings were created before a compiler was available, and do use some architectural features that the current compiler cannot utilize. Also, the mapping of the benchmarks is sometimes slightly different. For example, the *conv2d* manual mapping implements a spatial 3x3 convolution with an adder reduction tree using a very specialized datapath that is controlled by a single instruction stream, while the compiler implementation follows a more conventional vectorized mapping. However, for all benchmarks, the input data and algorithm functionality are equivalent.

The speedup of CGRA-M compared to CGRA-L is, on average, 1.39-2.14×. In general, the number of executed operations, as depicted in Figure 12(b), is very similar between CGRA-L and CGRA-M for most benchmarks. However, there are still some differences due to compiler limitations and architectural changes:

— Most CGRA-M mappings utilize address generators in the LSUs, which reduces the number of ALU operations for address calculations. It supports 1D strided loads and stores, a common feature in recent CGRAs [13, 32, 35, 38, 43, 59, 63]. This feature is currently not supported by the R-Blocks compiler but can be used using intrinsics.
— Some benchmarks use a significant amount of vector alignment operations (VecAlign). In the manual mappings these vector manipulation operations are implicit in the switchbox network. This is a feature that, to the best of our knowledge, is only found in the Blocks CGRA [63].
— The CGRA-M mappings use a vectorized scatter-gather GM interface, while CGRA-L uses a simpler scalar GM interface. This implies that data is copied from/to LM units before/after processing, which increases the number of LSU operations significantly. For compute-bound benchmarks with sufficient data reuse this is not an issue, but for memory-bound benchmarks, such as *binarization*, *erosion*, and *projection*, the additional LSU operations are significant. This vectorized scatter-gather GM interface can be used in R-Blocks using intrinsics (see Figure 3), but is on the critical path when the number of lanes increases, which is why we left it out in CGRA-L.

Some differences in favor of the compiler-generated mappings are also visible in the operation breakdown of Figure 12(b). More specifically, the assembly mappings of *matmul* and *fft* are parallelized differently, and *sampen*/*apen* use fewer LSU operations due to some loads that are hoisted out of the inner loop. These kinds of optimizations are very cumbersome to exploit while manually writing assembly, but having a compiler makes it easy. The remaining performance differences between CGRA-L and CGRA-M mappings cannot be explained by the difference in operation count, but are related to limitations of the heuristic-based OpenASIP instruction scheduler and architectural features of R-Blocks that the compiler cannot exploit. For all benchmarks (except *conv1d*, due to the additional VecAlign operations), the inner loop performance is <2× of the manual assembly, which is mostly due to the lack of software pipelining in the compiler-generated versions. The need for explicit data copies between GM and LM units in the CGRA-L mappings is also a bottleneck for smaller benchmarks (e.g. *binarization*, *erosion*, and *projection*), but less pronounced for bigger benchmarks that are compute-bound, such as *ffos*, *sampen*, *apen*, *vg*, *mfcc* and *cnn*.

The number of RF accesses for the manual mappings is significantly lower than the compiler-generated mappings, as is depicted in Figure 13. The CGRA-M mappings have only 0.02-0.08× the RF accesses of the RV32IM baseline, which is also 5.3-15× lower than the compiler-generated

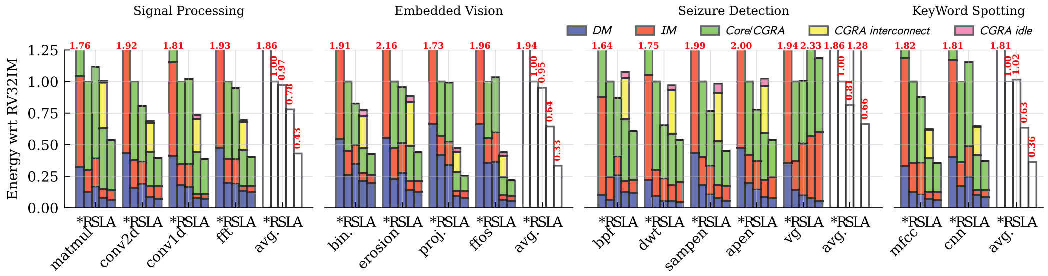


Fig. 15. Energy improvement of CGRA-S (S) and CGRA-L (L) CGRAs, ASIPs with fixed datapath and dedicated SIMD units (A) that were generated by the R-Blocks framework, compared against the RV32IM baseline (R) across different workloads. For completeness, we have also plotted the energy results of an RV32IM baseline without DM and IM optimization (*), as is commonly used in literature.

CGRA-L mappings. This highlights the limitations of the fast, but heuristic-based OpenASIP scheduler. Within the OpenASIP framework, a scheduler based on integer programming [1] was able to reduce the number of register reads and writes up to 33% and 18%, respectively, but this approach increased the compilation time from less than a second to minutes, even for small kernels. A recent work [51] improves the fast heuristic-based software bypassing algorithm in OpenASIP and reduces the number of register reads and writes by 26% and 37%, respectively. This approach should be used for R-Blocks in the future as well, as a heuristic-based OpenASIP scheduler with reasonable mapping quality allows for faster DSE over an exhaustive algorithm, like [1].

6.4 Energy Evaluation

In this section, we evaluate the energy savings of CGRA-L compared to the RISC-V processor (RV32IM) and CGRA-S. Furthermore, we discuss why CGRA-L is able to save energy compared to RV32IM and CGRA-S. Finally, we compare CGRA-L to ASIPs with dedicated SIMD units and a fixed datapath to quantify the cost of reconfiguration.

6.4.1 How much energy do we save? Figure 15 plots the normalized energy consumption of the reference architectures. CGRA-S consumes approximately the same amount of energy as RV32IM (0.81-1.01×), due to the number of executed instructions being very similar, as was previously discussed. For CGRA-L the results are split into two categories: Signal Processing, Embedded Vision, and Keyword Spotting all perform very well, consuming only 0.63-0.78× the RV32IM energy. On the other hand, Seizure Detection consumes 1.28× the energy of RV32IM, primarily due to the poor *vg* mapping due to the lack of high-ILP loops. The ASIPs with dedicated SIMD units and hardwired datapath in general save a significant amount of energy, namely 0.33-0.63×, showcasing the benefits software bypassing and SIMD-processing. When the ASIPs are compared with CGRA-L, the CGRA interconnect remains a major energy bottleneck. When comparing the ASIPs with CGRA-L, the energy price of reconfiguration is 1.75-1.94×, on average. Most of the ASIP savings are due to the lack of a reconfigurable interconnect, and the remaining savings (i.e. *Core/CGRA*) are due to the ASIPs being instantiated with dedicated SIMD units and the removal of unused FUs. The ASIPs could potentially be further optimized by removing logic of unused operations, by minimizing RF size, and by optimizing loop performance using handwritten assembly, but we consider these optimizations outside the scope of this work. This emphasizes the importance of further optimization of the interconnect, as was recently explored in [37]. Additionally, it should be stressed that further architectural optimizations, such as voltage-frequency scaling, make the parallel CGRA still an attractive solution for applications that are not throughput-constrained.

Table 4. Post-synthesis Energy Breakdown (in nJ) for the *fft* Benchmark on RV32IM Baseline and R-Blocks CGRAs

	Baseline	**R-Blocks CGRA**	
Component	RV32IM	CGRA-S	CGRA-L
DM	35.23	34.32	24.51
Global memory	35.23	34.32	4.00
Local memory	-	-	20.51
IM	34.44	34.74	8.00
Instruction memory	2.68	5.15	0.99
Instruction cache	31.77	29.59	7.01
Core/CGRA	109.57	100.97	51.85
Instr. fetch & decode	43.21	31.17	9.55
Functional units	32.09	44.40	35.66
Register files	25.68	9.01	3.70
Remaining	8.60	16.39	2.94
CGRA interconnect	-	-	39.78
Total energy (increase)	179.25 (1.00×)	170.03 (0.95×)	124.14 (0.69×)

For completeness, Figure 15 includes a design point of the same RISC-V processor without the instruction cache and without aggressive clock gating on the data memory (see *). This platform better reflects the reference platforms in many recent works [15, 17, 22, 23, 63]. Compared to our RV32IM baseline, this platform is 1.81-1.94× worse. This stresses the importance of a well-optimized baseline, as it can skew the results significantly in favor of the CGRA.

6.4.2 Why do We Save Energy? Table 4 presents a detailed energy breakdown of the *fft* benchmark running on CGRA-L. For the global and local **data memory energy (DM)**, it can be observed that exploiting the small LM units has a clear energy benefit, despite the overhead of data copies. The DM energy is decreased from 35.23 nJ to 24.51 nJ, a reduction of 30.4%. Going from CGRA-S to CGRA-L shows the benefits for SIMD-processing on instruction delivery overhead, reducing the instruction memory (IM) and instruction fetch and decoding overhead from 65.91 nJ to 17.55 nJ, a reduction of 73.4%. This is in line with the reduced number of instruction fetches due to SIMD-processing, as was discussed in Section 6.2.3. Also, the benefits of software bypassing, as was discussed in Section 6.2.2, are clearly visible in the breakdown. Going from RV32IM baseline to CGRA-S reduces the energy consumption of the RFs from 25.68 nJ to 9.01 nJ, a reduction of 64.9%. From CGRA-S to CGRA-L, the number of RF accesses is significantly reduced, leading to a further energy reduction from 9.01 nJ to 3.70 nJ, another 59.0%. Despite the number of RF accesses of the compiler-generated *fft* being significantly higher than the manual mapping, the impact on the total energy consumption of the RF accesses is limited to approximately 3.0%. This is much less than traditional vector processors [24, 54] (i.e. in these works ≈20-50% of total system energy is spent on RF accesses), which highlights the efficiency of the R-Blocks fabric and compiler. It should be noted that approximately 32.0% of the total energy is spent in the CGRA interconnect, a significant fraction, but in line with other works [23, 57, 63].

We provide a breakdown of individual architecture components on energy consumption, which is depicted in Table 5. It follows that the cost of reconfiguration, i.e. the switchbox networks, increases the energy consumption by 29%. However, instruction broadcasting with flexible SIMD-processing, software bypassing, and local memories all improve the energy consumption significantly, leading to a final energy consumption of 0.69× compared to RV32IM. The energy savings

Table 5. Summary of System-level Energy Savings for the *fft* Benchmark

Platform	Relative energy	Energy difference
Baseline – RV32IM	1.00×	-
R-Blocks – CGRA-S	0.95×	-5%
+ Reconfiguration[1]	1.22×	+29%
+ SIMD-processing[2]	0.88×	-27%
+ Software bypassing[3]	0.76×	-14%
+ Local memories[4]	0.70×	-8%
R-Blocks – CGRA-L	0.69×	-2%

Note: relative energy is computed using Table 4. The breakdown compares (CGRA-L wrt RV32IM): [1]CGRA interconnect energy. [2] IM + instr. fetch & decode energy. [3]RF energy. [4] DM energy.

Component	Baseline RV32IM	R-Blocks CGRA CGRA-S	CGRA-L
Local memory	-	-	0.044
Instruction memory	0.052	0.027	0.069 (2.51×)
Instruction cache	0.019	0.014	0.035 (2.50×)
Core/CGRA	0.013	0.012 (0.96×)	0.101 (7.93×)
CGRA interconnect	-	-	0.236
Total (excl. GM)	0.083	0.054	0.486 (9.05×)

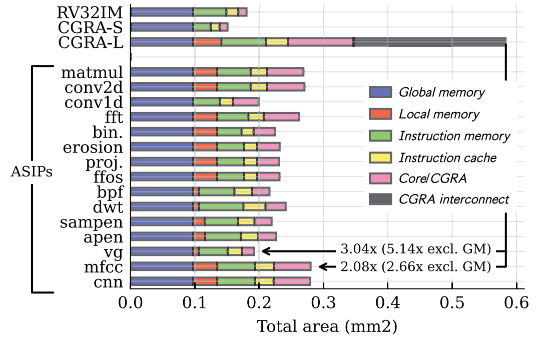


Fig. 16. Post-synthesis area breakdown (in mm^2) on RV32IM baseline, R-Blocks CGRAs, and R-Blocks ASIPs.

on the *fft* benchmark translate to similar savings over all benchmarks. On average, the use of LM units reduce the DM energy by 45.1% (vs. 30.4% for *fft*). Additionally, the instruction delivery overhead is reduced by 47.2% (vs. 73.4% for *fft*). The RF energy from RV32IM baseline to CGRA-L is reduced by 73.3% (vs. 64.9% for *fft*), and the impact of RF accesses on the total energy consumption is around 3.7% (vs. 3.0% for *fft*). Finally, the CGRA interconnect contributes to 35.1% (vs. 32.0% for *fft*) of the total energy.

6.5 Area Evaluation

Figure 16 depicts the area breakdown for all platforms. The area of all platforms is dominated by the different memories. If we consider the core-only area (*Core/CGRA* in figure), the CGRA-S and CGRA-L cores are 0.96× and 7.93× the RISC-V core, respectively. We consider the area increase of CGRA-L reasonable, as the performance scaling of the applications that utilize all fabric resources was between 6.7-10.7× (see Figure 12(a)). One interesting observation is that the area of the instruction memory and instruction cache between CGRA-S and CGRA-L is only increased by 2.51×, while the total area difference is 9.05× (excluding *Global memory*), which highlights the benefits of instruction broadcasting. Overall, the area cost of CGRA-L is 2.66-5.14× compared to ASIPs when we consider the CGRA fabric only (excluding *Global memory*), and 2.08-3.04× for the whole system.

7 COMPARISON WITH PRIOR ART

Many CGRAs have been proposed over the last few decades for acceleration of a wide variety of applications. [34, 64] provide a recent overview. Most of these works focus on the fast and efficient mapping of modulo-scheduled DFGs onto a CGRA fabric with a constrained NoC or static

Table 6. Comparison of R-Blocks CGRA with State-of-the-art ULP CGRAs

Metric	This work	ULP-SRP [30, 31]	HyCUBE0+- [57, 57]	ComplexStream [47]	SNAFU [22]	Blocks [15]	VWR2A [16]	Riptide [23]
Year	2023	2013	2019	2020	2021	2021	2022	2022
Execution model	VLIW-SIMD	VLIW	DFG	DFG	DFG	VLIW-SIMD	VLIW	DFG with CF
Program support	C-code	C-code	C-code	Assembly	Assembly	Assembly	Assembly	C-code
Acceleration scope	Full applications	Full applications	Inner loops, limited by context memory size	Inner loops, limited by #PEs on fabric	Inner loops, limited by #PEs on fabric	Functions, limited by context memory size	Functions, limited by context memory size	Functions, limited by #PEs on fabric
Applications (domains)	Various (embedded, e-Health, IoT)	Heart-beat detection (e-Health)	Various (embedded, e-Health, IoT)	Signal processing	Signal processing and linear algebra	Seizure detection (e-Health)	CW monitoring (e-Health)	Signal processing and linear algebra
Heterogeneous PEs	Yes	No	No	No	Yes	Yes	Yes	Yes
SW-bypassing support	Yes	Limited	Yes	Yes	Yes	Yes	Limited	Yes
Flexible SIMD support	Yes	No	No	No	No	Yes	No	No
Technology	22-nm FD-SOI	40-nm CMOS	40-nm CMOS	28-nm CMOS	22-nm FinFET	28-nm FD-SOI	40-nm CMOS	22-nm FinFET
Implementation	Post-synthesis	Chip measurement	Post-layout	Chip measurement	Post-synthesis	Chip measurement	Post-synthesis	Post-synthesis
PE architecture	ALU: 32b, MUL: 32b × 32b⟶32b	ALU: 32b, MUL: 32b × 32b⟶32b*	ALU: 32b, MUL: 32b × 32b⟶32b**	ALU: 16b, MUL: 16b × 16b⟶32b	ALU: 32b, MUL: 32b × 32b⟶32b	ALU: 32b, MUL: 32b × 32b⟶64b	ALU: 32b, MUL: 32b × 32b⟶32b	ALU: 32b, MUL: 32b × 32b⟶32b
Fabric dimensions	11 × 7 PEs	3 × 3 PEs	4 × 4 PEs	12 × 12 PEs	6 × 6 PEs	6 × 5 PEs	4 × 2 PEs	6 × 6 PEs
Fabric size	$0.486\,mm^2$	$1.443\,mm^{2\dagger}$	$1.404\,mm^{2\dagger}$	$3.9\,mm^2$	$0.27\,mm^2$	$0.487\,mm^{2\dagger}$	N/A	$0.25\,mm^2$
Supply voltages	0.8 V	logic: 0.5 V, memory: 0.7 V	1.1 V	0.6 V	N/A	logic: 0.46 V, memory: 0.85 V	N/A	N/A
Operation frequency	100 MHz	7 MHz	853 MHz	72 MHz	50 MHz	5 MHz	80 MHz	50 MHz
Power dissipation	3.99 mW	0.546 mW	72 mW	38.7 mW	0.74 mW	0.156 mW	5.41 mW	0.52 mW
FFT throughput	461 MOPS§	12 MOPS§	6483 MOPS	4500 MOPS	71 MOPS	8 MOPS	259 MOPS§	62 MOPS
FFT efficiency	115 MOPS/mW	22 MOPS/mW	90 MOPS/mW	116 MOPS/mW	97 MOPS/mW	51 MOPS/mW	48 MOPS/mW	117 MOPS/mW
energy/FFT	0.12 µJ	0.66 µJ	0.16 µJ (est. using §)	0.12 µJ (est. using §)	0.21 µJ	0.28 µJ	0.30 µJ	0.17 µJ

† CGRA fabric size area, including local scratchpad memories but excluding other on-chip SRAM memories, manually derived from annotated chip layout picture. § For the fixed-point *fft* benchmark we consider $14 \cdot (N/2)\log_2(N)$ for $N = 256$ arithmetic operations per execution. * [30] mentions that ULP-SRP is based on ADRES, a CGRA with 32-bit PEs. * [3] mentions that HyCUBE closely matches their re-implementation with 32-bit PEs.
R-Blocks accelerates full applications with great energy efficiency without excessive specialization.

mesh network. As such, their main optimization criteria are based on performance or compilation speed [10, 59, 60], and not on energy efficiency. Some works do focus on energy efficiency, but only present relative energy savings or use first-order energy models, which prohibits an accurate comparison between CGRAs in different works [3, 18, 19, 33, 35, 38, 43, 58]. Other works present absolute energy numbers, but only report metrics that provide an indication of the energy efficiency in terms of the number of operations that are executed (e.g. loads/stores, logic & arithmetic, branching, etc.) on the CGRA fabric per unit of energy (e.g. Million Instructions per Second per milliWatt or MIPS/mW) [14, 15, 30, 31]. This metric becomes meaningless between different CGRAs when the number and types of instructions to perform the same algorithm are very different, due to differences in ISA and mapping quality. We argue that the most important metric for energy efficiency is the workload efficiency or full-system energy cost to perform a fixed amount of work. In the absence of being able to run exactly the same benchmark, this efficiency can be approximated in **Million Operations per Second per milliWatt (MOPS/mW)**, where only the essential arithmetic operations are counted for a benchmark, e.g. $3 \cdot N^3$ operations per $N \times N$ fixed-point MatMul (1 multiply-shift-accumulate equals 3 operations) or equivalently energy per execution in µJ.

Table 6 provides an overview of recent programmable ULP CGRAs. For a fair comparison, we consider CGRAs that optimize for energy-efficient acceleration of a variety of compute-intensive benchmarks in the embedded, e-Health, or IoT domain within an ultra-low power budget. We compare the CGRAs in terms of architectural features and compiler support and compare the system-level energy efficiency on a commonly used FFT benchmark. We observe a trend from executing only inner loops on the CGRA fabric towards off-loading complete functions, including control flow. However, there are two primary limitations to most approaches: 1) for spatial DFG mappings the maximum DFG size is limited by the number of PEs, and 2) when cycle-level reconfiguration of PEs is supported using a context memory (i.e. initiation interval $II \geq 1$), the size

of this context memory is kept small (e.g. ≤ 32 instruction lines), as it has a significant impact on system-level energy efficiency. R-Blocks avoids these limitations by employing a reconfigurable instruction cache, which makes the system-level energy efficiency relatively insensitive to the context memory size. Furthermore, R-Blocks supports temporal processing using the well-defined VLIW-SIMD execution model, which allows us to reuse existing compiler developments of the last few decades. Additionally, this approach is highly flexible; any application can be executed on any virtArch, independent of the number of FUs, using the retargetable OpenASIP compiler.

In terms of architectural features, R-Blocks avoids the centralized RF bottleneck that is commonly found in CGRA and VLIW architectures by supporting full software bypassing. The degree of software bypassing support can fluctuate, i.e. CGRAs that map modulo-scheduled DFGs often do support direct communication with other DFG nodes over the interconnect, but they rely on control flow and/or memory operations being executed on an external host processor, which limits their energy efficiency. Alternatively, ULP-SRP utilizes a centralized RF and only supports limited software bypassing by means of nearest-neighbor connections. Similar to many recent CGRAs, R-Blocks supports heterogeneous PEs which increases compiler scheduling complexity but improves area and energy efficiency by reducing idle resources. Finally, R-Blocks is the only CGRA with compiler support for flexible SIMD-processing without dedicated SIMD units within an ultra-low power budget. There are some interesting CGRAs with some degree of SIMD support that are not mentioned in Table 6, but these utilize dedicated SIMD units [18, 19, 40, 43] or target the high-performance domain and have limited energy efficiency [41, 65].

In terms of energy efficiency, the R-Blocks CGRA-L instance is competitive on a representative 256-point complex FFT benchmark, compared to [15, 16, 22, 23, 30, 31, 47, 57]. As mentioned before, we report the energy efficiency in MOPS/mW and energy per execution, counting only the essential arithmetic operations (i.e. excluding loads/stores, register file operations, branching instructions, and immediates), while considering the full-system power dissipation, which is consistent with the approach in other works, such as [15, 22, 23, 57]. For the other works, we estimated the energy efficiency using values and measurement results from the papers in [16, 30, 31, 47], using an optimistic fixed-point FFT kernel workload estimate of $14 \cdot (N/2)\log_2(N)$ arithmetic operations for $N = 256$. Considering the throughput of 461 MOPS, and power dissipation of 3.99 mW for the R-Blocks CGRA-L mappings of the *fft* benchmark, the workload efficiency is 115 MOPS/mW. If we include all executed operations on the R-Blocks fabric, the workload efficiency improves to 332 MOPS/mW, which is more equivalent to MIPS/mW that is often reported [14, 15, 30, 31]. Similarly, the fixed-point *matmul* benchmark achieves a throughput of 678 MOPS, when we consider $3 \cdot N^3$ operations per execution, with a power dissipation of 5.84 mW, resulting in an energy efficiency of 116 MOPS/mW, or 283 MOPS/mW when including all executed operations.

It should be noted that the R-Blocks fabric can also effectively accelerate complete applications. When we consider our multi-kernel benchmarks, i.e. *ffos*, *sampen*, *apen*, *mfcc* and *cnn*, the workload efficiency is 101, 86, 90, 107 and 115 MOPS/mW when only considering essential arithmetic operations, or 426, 298, 391, 345 and 313 MOPS/mW when all operations are included, respectively. These values are very similar to the energy efficiency obtained on the smaller *fft* and *matmul* benchmarks, unlike in [23], where the energy efficiency drops significantly for a complete CNN application, most likely due to the additional reconfiguration and data transfers introduced by splitting the application in multiple contexts. R-Blocks supports standalone execution of large applications on the same virtArch, which reduces the synchronization overheads between the host processor. To accelerate complete applications with different virtArchs we need to reconfigure the fabric occasionally. As long as the execution time between reconfigurations is sufficiently large, i.e. >100 Kcycles, or we can reuse the configuration for multiple consecutive acceleration requests,

the reconfiguration overhead will be negligible (<5%) [8, 15]. For reference, the seizure detection application in [15] has a CGRA reconfiguration overhead of only 1.89% of the total execution time.

Finally, in terms of area efficiency, the R-Blocks reconfigurable interconnect has a significant area penalty. It follows from Table 1 that the area increase of the R-Blocks CGRA-L instance due to the interconnect is approximately 1.9×, which is comparable to [23] (54% of total area), which includes support for control flow in the NoC, but significantly more than the reconfigurable 2D-mesh network as presented in [29] (24% of total area). A possible explanation for this large interconnect area penalty in R-Blocks is the larger number of tracks required to provide sufficient RF connectivity for the OpenASIP compiler. We hypothesize that an improved compiler capable of handling R-Blocks cores with even more restricted datapaths could significantly reduce the interconnect complexity by lowering the number of tracks required for RF connectivity, as was recently demonstrated within the same OpenASIP framework for TTA cores [51].

8 CONCLUSIONS AND FUTURE WORK

This work presented R-Blocks: a CGRA for energy-efficient acceleration of complex applications with accompanying compilation tool-flow. This tool-flow enabled efficient code generation using the well-established VLIW-SIMD execution model for a wide variety of application-optimized cores with heavily restricted PE connectivity. The R-Blocks CGRA is unique in the sense that it allows for flexible SIMD-processing without excessive specialization of dedicated SIMD units. We primarily focused on evaluating the R-Blocks CGRA against highly tuned baselines. To achieve this, we optimized an embedded RISC-V platform, and closely matched the operation set, instruction delivery, and application mappings of the R-Blocks CGRA. This allowed us to evaluate the cost of reconfiguration and benefits of architectural optimizations only: flexible SIMD-processing, software bypassing, and the use of local scratchpads. Even though the R-Blocks CGRA offers greater flexibility compared to many other ULP CGRAs and has potential for further code generation improvements, it achieved state-of-the-art energy efficiency on a popular FFT benchmark, and comparable efficiency on more complex benchmarks. This work provides a motivation to the CGRA research community to design future CGRAs with more flexibility using standardized execution models, which allows the reuse of existing compiler developments.

In terms of future work on the R-Blocks CGRA, ISA extensions that are commonly found in DSPs are certainly promising. Also, software pipelining support for CGRAs with heavily restricted PE connectivity is necessary to further improve inner loop code density and performance. We also aim to extend the R-Blocks tool-flow with fast DSE approaches to automatically find an optimized R-Blocks virtArch and physArch for a set of applications under resource constraints. Finally, HW/SW support for multi-processing and run-time resource management on the R-Blocks CGRA is critical to maximize resource utilization and energy efficiency for supporting more complex applications.

ACKNOWLEDGMENT

This work was part of the research programme Perspectief ZERO (P15-06 Project 5), which is (partly) financed by the Dutch Research Council (NWO). This work was also supported by the Academy of Finland (decision #331344).

REFERENCES

[1] Tomi Äijö, Pekka Jääskeläinen, Tapio Elomaa, Heikki Kultala, and Jarmo Takala. 2015. Integer linear programming-based scheduling for transport triggered architectures. *ACM Transactions on Architecture and Code Optimization* 12, 4 (2015), 1–22. https://doi.org/10.1145/2845082

[2] Ensieh Aliagha and Diana Göhringer. 2022. Energy efficient design of coarse-grained reconfigurable architectures: Insights, trends and challenges. In *International Conference on Field-Programmable Technology (ICFPT)*. 1–11. https://doi.org/10.1109/ICFPT56656.2022.9974339

[3] Thilini Kaushalya Bandara, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. 2022. REVAMP: A systematic framework for heterogeneous CGRA realization. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Lausanne, Switzerland). 918–932. https://doi.org/10.1145/3503222.3507772

[4] S. Benatti, F. Montagna, D. Rossi, and L. Benini. 2016. Scalable EEG seizure detection on an ultra low power multicore architecture. In *IEEE Biomedical Circuits and Systems Conference (BioCAS)*. 86–89. https://doi.org/10.1109/BioCAS.2016.7833731

[5] Vaughn Betz and Jonathan Rose. 1997. VPR: A new packing, placement and routing tool for FPGA research. In *7th International Conference on Field Programmable Logic and Applications (FPL)*. 213–222.

[6] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. 1999. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, USA.

[7] Anthony Brandon, Joost Hoozemans, Jeroen van Straten, and Stephan Wong. 2017. Exploring ILP and TLP on a polymorphic VLIW processor. In *Architecture of Computing Systems (ARCS)*. 177–189.

[8] Alex Carsello, Kathleen Feng, Taeyoung Kong, Kalhan Koul, Qiaoyi Liu, Jackson Melchert, Gedeon Nyengele, Maxwell Strange, Keyi Zhang, Ankita Nayak, Jeff Setter, James Thomas, Kavya Sreedhar, Po-Han Chen, Nikhil Bhagdikar, Zachary Myers, Brandon D'Agostino, Pranil Joshi, Stephen Richardson, Rick Bahr, Christopher Torng, Mark Horowitz, and Priyanka Raina. 2022. Amber: A 367 GOPS, 538 GOPS/W 16nm SoC with a coarse-grained reconfigurable array for flexible acceleration of dense linear algebra. In *IEEE Symposium on VLSI Technology and Circuits*. 70–71. https://doi.org/10.1109/VLSITechnologyandCir46769.2022.9830509

[9] Matheus Cavalcante, Domenic Wüthrich, Matteo Perotti, Samuel Riedel, and Luca Benini. 2022. Spatz: A compact vector processing unit for high-performance and energy-efficient shared-L1 clusters. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design* (San Diego, California,) *(ICCAD'22)*. Association for Computing Machinery, New York, NY, USA, Article 22, 9 pages. https://doi.org/10.1145/3508352.3549367

[10] S. Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. 2017. CGRA-ME: A unified framework for CGRA modelling and exploration. In *IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 184–189. https://doi.org/10.1109/ASAP.2017.7995277

[11] Henk Corporaal. 1997. *Microprocessor Architectures: From VLIW to TTA*. John Wiley & Sons, Inc., USA.

[12] Satyajit Das, Kevin J. M. Martin, Davide Rossi, Philippe Coussy, and Luca Benini. 2019. An energy-efficient integrated programmable array accelerator and compilation flow for near-sensor ultralow power processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 6 (2019), 1095–1108. https://doi.org/10.1109/TCAD.2018.2834397

[13] Satyajit Das, Rohit Prasad, Kevin J. M. Martin, and Philippe Coussy. 2020. Energy efficient acceleration of floating point applications onto CGRA. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 1563–1567. https://doi.org/10.1109/ICASSP40776.2020.9054613

[14] Satyajit Das, Davide Rossi, Kevin J. M. Martin, Philippe Coussy, and Luca Benini. 2017. A 142MOPS/mW integrated programmable array accelerator for smart visual processing. In *IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–4. https://doi.org/10.1109/ISCAS.2017.8050238

[15] Barry de Bruin, Kamlesh Singh, Ying Wang, Jos Huisken, José Pineda de Gyvez, and Henk Corporaal. 2021. Multilevel optimization of an ultra-low power brainwave system for non-convulsive seizure detection. *IEEE Transactions on Biomedical Circuits and Systems* 15, 5 (2021), 1107–1121. https://doi.org/10.1109/TBCAS.2021.3120965

[16] Benoît W. Denkinger, Miguel Peón-Quirós, Mario Konijnenburg, David Atienza, and Francky Catthoor. 2022. VWR2A: A very-wide-register reconfigurable-array architecture for low-power embedded devices. In *59th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 895–900. https://doi.org/10.1145/3489517.3530980

[17] Benoît Walter Denkinger, Miguel Peón-Quirós, Mario Konijnenburg, David Atienza, and Francky Catthoor. 2023. Acceleration of control intensive applications on coarse-grained reconfigurable arrays for embedded systems. *IEEE Trans. Comput.* 72, 9 (2023), 2548–2560. https://doi.org/10.1109/TC.2023.3257504

[18] L. Duch, S. Basu, R. Braojos, G. Ansaloni, L. Pozzi, and D. Atienza. 2017. HEAL-WEAR: An ultra-low power heterogeneous system for bio-signal analysis. *IEEE Transactions on Circuits and Systems I: Regular Papers* 64, 9 (2017), 2448–2461. https://doi.org/10.1109/TCSI.2017.2701499

[19] Loris Duch, Soumya Basu, Miguel Peón-Quirós, Giovanni Ansaloni, Laura Pozzi, and David Atienza. 2019. i-DPs CGRA: An interleaved-datapaths reconfigurable accelerator for embedded bio-signal processing. *IEEE Embedded Systems Letters* 11, 2 (2019), 50–53. https://doi.org/10.1109/LES.2018.2849267

[20] Marco Fariselli, Manuele Rusci, Joel Cambonie, and Eric Flamand. 2021. Integer-only approximated MFCC for ultra-low power audio NN processing on multi-core MCUs. In *IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. 1–4. https://doi.org/10.1109/AICAS51828.2021.9458491

[21] Stephen Friedman, Allan Carroll, Brian Van Essen, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. 2009. SPR: An architecture-adaptive CGRA mapping tool. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)* (Monterey, California, USA). 191–200. https://doi.org/10.1145/1508128.1508158

[22] Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia, and Nathan Beckmann. 2021. Snafu: An ultra-low-power, energy-minimal cgra-generation framework and architecture. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 1027–1040. https://doi.org/10.1109/ISCA52012.2021.00084

[23] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. 2022. RipTide: A programmable, energy-minimal dataflow compiler and architecture. In *55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 546–564. https://doi.org/10.1109/MICRO56248.2022.00046

[24] Graham Gobieski, Amolak Nagi, Nathan Serafin, Mehmet Meric Isgenc, Nathan Beckmann, and Brandon Lucia. 2019. MANIC: A vector-dataflow architecture for ultra-low-power embedded systems. In *52nd IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Columbus, OH, USA). 670–684. https://doi.org/10.1145/3352460.3358277

[25] Yifan He, Zhenyu Ye, Dongrui She, Bart Mesman, and Henk Corporaal. 2011. Feasibility analysis of ultra high frame rate visual servoing on FPGA and SIMD Processor. In *Advanced Concepts for Intelligent Vision Systems*. Springer Berlin , 623–634.

[26] Alex Hirvonen, Topi Leppänen, Kari Hepola, Joonas Multanen, Joost Hoozemans, and Pekka Jääskeläinen. 2023. AEx: Automated high-level synthesis of compiler programmable co-processors. *Journal of Signal Processing Systems* (2023), 1–15. https://doi.org/10.1007/s11265-023-01841-3

[27] Pekka Jääskeläinen, Timo Viitanen, Jarmo Takala, and Heikki Berg. 2017. *HW/SW Co-design Toolset for Customization of Exposed Datapath Processors*. Springer International Publishing, 147–164. https://doi.org/10.1007/978-3-319-49679-5_8

[28] Pekka Jääskeläinen, Aleksi Tervo, Guillermo Payá Vayá, Timo Viitanen, Nicolai Behmann, Jarmo Takala, and Holger Blume. 2018. Transport-triggered soft cores. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 83–90. https://doi.org/10.1109/IPDPSW.2018.00022

[29] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. 2017. HyCUBE: A CGRA with reconfigurable single-cycle multi-hop interconnect. In *54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/3061639.3062262

[30] Changmoo Kim, Mookyoung Chung, Yeongon Cho, Mario Konijnenburg, Soojung Ryu, and Jeongwook Kim. 2014. ULP-SRP: Ultra low-power samsung reconfigurable processor for biomedical applications. *ACM Transactions on Reconfigurable Technology and Systems* 7, 3, Article 22 (9 2014), 15 pages. https://doi.org/10.1145/2629610

[31] Mario Konijnenburg, Yeongojn Cho, Maryam Ashouei, Tobias Gemmeke, Changmoo Kim, Jos Hulzink, Jan Stuyt, Mookyung Jung, Jos Huisken, Soojung Ryu, Jungwook Kim, and Harmke de Groot. 2013. Reliable and energy-efficient 1MHz 0.4V dynamically reconfigurable SoC for ExG applications in 40nm LP CMOS. In *IEEE International Solid- State Circuits Conference (ISSCC)*. 430–431. https://doi.org/10.1109/ISSCC.2013.6487801

[32] Kalhan Koul, Jackson Melchert, Kavya Sreedhar, Leonard Truong, Gedeon Nyengele, Keyi Zhang, Qiaoyi Liu, Jeff Setter, Po-Han Chen, Yuchen Mei, Maxwell Strange, Ross Daly, Caleb Donovick, Alex Carsello, Taeyoung Kong, Kathleen Feng, Dillon Huff, Ankita Nayak, Rajsekhar Setaluri, James Thomas, Nikhil Bhagdikar, David Durst, Zachary Myers, Nestan Tsiskaridze, Stephen Richardson, Rick Bahr, Kayvon Fatahalian, Pat Hanrahan, Clark Barrett, Mark Horowitz, Christopher Torng, Fredrik Kjolstad, anPriyanka Raina. 2023. AHA: An agile approach to the design of coarse-grained reconfigurable accelerators and compilers. *ACM Transactions on Embedded Computer Systems* 22, 2, Article 35 (1 2023), 34 pages. https://doi.org/10.1145/3534933

[33] Zhaoying Li, Dhananjaya Wijerathne, Xianzhang Chen, Anuj Pathania, and Tulika Mitra. 2022. ChordMap: Automated mapping of streaming applications onto CGRA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 2 (2022), 306–319. https://doi.org/10.1109/TCAD.2021.3058313

[34] 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. *Comput. Surveys* 52, 6, Article 118 (10 2019), 39 pages. https://doi.org/10.1145/3357375

[35] João D. Lopes and José T. de Sousa. 2017. Versat, a minimal coarse-grain reconfigurable array. In *High Performance Computing for Computational Science – VECPAR 2016*. Springer International Publishing, Cham, 174–187.

[36] L. McMurchie and C. Ebeling. 1995. PathFinder: A negotiation-based performance-driven router for FPGAs. In *Third International ACM Symposium on Field-Programmable Gate Arrays*. 111–117. https://doi.org/10.1109/FPGA.1995.242049

[37] J. Melchert, K. Zhang, Y. Mei, M. Horowitz, C. Torng, and P. Raina. 2023. Canal: A flexible interconnect generator for coarse-grained reconfigurable arrays. *IEEE Computer Architecture Letters* 22, 01 (Jan. 2023), 45–48. https://doi.org/10.1109/LCA.2023.3268126

[38] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-dataflow acceleration. In *44th Annual International Symposium on Computer Architecture (ISCA)* (Toronto, ON, Canada). 416–429. https://doi.org/10.1145/3079856.3080255

[39] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-vectorization of interleaved data for SIMD. In *27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Ottawa, Ontario, Canada). 132–143. https://doi.org/10.1145/1133981.1133997

[40] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2013. Efficient execution of augmented reality applications on mobile programmable accelerators. In *2013 International Conference on Field-Programmable Technology (FPT)*. 176–183. https://doi.org/10.1109/FPT.2013.6718350

[41] Yongjun Park, Jason Jong Kyu Park, Hyunchul Park, and Scott Mahlke. 2012. Libra: Tailoring SIMD execution using heterogeneous hardware and dynamic configurability. In *45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 84–95. https://doi.org/10.1109/MICRO.2012.17

[42] A. Podobas, K. Sano, and S. Matsuoka. 2020. A survey on coarse-grained reconfigurable architectures from a performance perspective. *IEEE Access* 8 (2020), 146719–146743. https://doi.org/10.1109/ACCESS.2020.3012084

[43] 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 *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 389–402. https://doi.org/10.1145/3079856.3080256

[44] Rohit Prasad, Satyajit Das, Kevin JM Martin, and Philippe Coussy. 2021. Floating point CGRA based ultra-low power DSP accelerator. *Journal of Signal Processing Systems* 93, 10 (2021), 1159–1171. https://doi.org/10.1007/s11265-020-01630-2

[45] Rohit Prasad, Satyajit Das, Kevin J. M. Martin, Giuseppe Tagliavini, Philippe Coussy, Luca Benini, and Davide Rossi. 2020. TRANSPIRE: An energy-efficient TRANSprecision floating-point Programmable archItectuRE. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1067–1072. https://doi.org/10.23919/DATE48585.2020.9116408

[46] Namita Sharma, Preeti Ranjan Panda, Francky Catthoor, Praveen Raghavan, and Tom Vander Aa. 2015. Array interleavingan energy-efficient data layout transformation. *ACM Transactions on Design Automation of Electronic Systems* 20, 3 (2015), 1–26. https://doi.org/10.1145/2747875

[47] Sander Smets, Manil Dev Gomony, Mihaela Jivanescu, Toon Goedemé, and Marian Verhelst. 2020. A 28-nm coarse grain 2D-reconfigurable array with data forwarding. *IEEE Solid-State Circuits Letters* 3 (2020), 226–229. https://doi.org/10.1109/LSSC.2020.3012019

[48] Chilankamol Sunny, Satyajit Das, Kevin J. M. Martin, and Philippe Coussy. 2021. Hardware based loop optimization for CGRA architectures. In *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Springer International Publishing, 65–80. https://doi.org/10.1007/978-3-030-79025-7_5

[49] Cheng Tan, Nicolas Bohm Agostini, Jeff Zhang, Marco Minutoli, Vito Giovanni Castellana, Chenhao Xie, Tong Geng, Ang Li, Kevin Barker, and Antonino Tumeo. 2021. OpenCGRA: Democratizing coarse-grained reconfigurable arrays. In *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 149–155. https://doi.org/10.1109/ASAP52443.2021.00029

[50] Kati Tervo, Samawat Malik, Topi Leppänen, and Pekka Jääskeläinen. 2020. TTA-SIMD soft core processors. In *30th International Conference on Field-Programmable Logic and Applications (FPL)*. 79–84. https://doi.org/10.1109/FPL50879.2020.00023

[51] Kanishkan Vadivel, Barry de Bruin, Roel Jordans, Henk Corporaal, and Pekka Jääskeläinen. 2022. Prebypass: Software register file bypassing for reduced interconnection architectures. In *25th Euromicro Conference on Digital System Design (DSD)*. 157–164. https://doi.org/10.1109/DSD57027.2022.00030

[52] Kanishkan Vadivel, Roel Jordans, Sander Stujik, Henk Corporaal, Pekka Jääskeläinen, and Heikki Kultala. 2019. Towards efficient code generation for exposed datapath architectures. In *Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems*. 86–89. https://doi.org/10.1145/3323439.3323990

[53] Kanishkan Vadivel, Mark Wijtvliet, Roel Jordans, and Henk Corporaal. 2017. Loop overhead reduction techniques for coarse grained reconfigurable architectures. In *2017 Euromicro Conference on Digital System Design (DSD)*. 14–21. https://doi.org/10.1109/DSD.2017.83

[54] Artem Vasilyev, Nikhil Bhagdikar, Ardavan Pedram, Stephen Richardson, Shahar Kvatinsky, and Mark Horowitz. 2016. Evaluating programmable architectures for imaging and vision applications. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783755

[55] Luc Waeijen, Dongrui She, Henk Corporaal, and Yifan He. 2014. Reduction operator for wide-SIMDs reconsidered. In *51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2593069.2593198

[56] Luc Waeijen, Dongrui She, Henk Corporaal, and Yifan He. 2015. A low-energy wide SIMD architecture with explicit datapath. *Journal of Signal Processing Systems* 80, 1 (2015), 65–86. https://doi.org/10.1007/s11265-014-0950-8

[57] 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 *IEEE Asian Solid-State Circuits Conference (A-SSCC)*. 133–136. https://doi.org/10.1109/A-SSCC47793.2019.9056954

[58] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. 2020. DSAGEN: Synthesizing programmable spatial accelerators. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 268–281. https://doi.org/10.1109/ISCA45697.2020.00032

[59] Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunarathne, Anuj Pathania, and Tulika Mitra. 2019. CASCADE: High throughput data streaming via decoupled access-execute CGRA. *ACM Transactions on Embedded Computing Systems* 18, 5s, Article 50 (2019), 26 pages. https://doi.org/10.1145/3358177

[60] Dhananjaya Wijerathne, Zhaoying Li, Anuj Pathania, Tulika Mitra, and Lothar Thiele. 2022. HiMap: Fast and scalable high-quality mapping on CGRA via hierarchical abstraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 10 (2022), 3290–3303. https://doi.org/10.1109/TCAD.2021.3132551

[61] Mark Wijtvliet, Henk Corporaal, and Akash Kumar. 2021. CGRA-EAMrapid energy and area estimation for coarse-grained reconfigurable architectures. *ACM Transactions on Reconfigurable Technology and Systems* 14, 4, Article 19 (2021), 28 pages. https://doi.org/10.1145/3468874

[62] M. Wijtvliet, J. Huisken, L. Waeijen, and H. Corporaal. 2019. Blocks: Redesigning coarse grained reconfigurable architectures for energy efficiency. In *29th International Conference on Field Programmable Logic and Applications (FPL)*. 17–23. https://doi.org/10.1109/FPL.2019.00013

[63] M. Wijtvliet, A. Kumar, and H. Corporaal. 2022. Blocks: Challenging SIMDs and VLIWs with a reconfigurable architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 9 (2022), 2915–2928. https://doi.org/10.1109/TCAD.2021.3120541

[64] Mark Wijtvliet, Luc Waeijen, and Henk Corporaal. 2016. Coarse grained reconfigurable architectures in the past 25 years: Overview and classification. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. 235–244. https://doi.org/10.1109/SAMOS.2016.7818353

[65] Chen Yin, Naifeng Jing, Jianfei Jiang, Qin Wang, and Zhigang Mao. 2023. A reschedulable dataflow-SIMD execution for increased utilization in CGRA cross-domain acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 3 (2023), 874–886. https://doi.org/10.1109/TCAD.2022.3185544

[66] Matthew Ouellette, Mouli C. Venkata, Brian Philofsky, Harpinder Matharu, Faisal Dada, Ashwin Thiagarajan, Nick Ni, Ryan Koehn, and Frederic Rivoallon. 2023. *"System-Level Benefits of the Versal Platform"*. Xilinx inc. Available at https://xilinx.eetrend.com/files/2021-08/wen_zhang_/100553221-218344-wp539-versal-system-level-benefits.pdf (visited on Aug. 2, 2023).

Received 14 June 2023; revised 29 February 2024; accepted 3 April 2024