

MC-CGRA: A Memory-Computation Coordinated CGRA Framework for Stream Processing

Chen Shi, Chunhua Xiao*, Han Diao, Weijie Yuan and Junling Wang

College of Computer Science, Chongqing University, Chongqing, China

*Email: xiaochunhua@cqu.edu.cn

Abstract—Coarse-Grained Reconfigurable Arrays (CGRAs) have emerged as a promising platform for domain-specific accelerators. However, traditional designs face significant limitations in large-scale stream processing. Current decoupled software-hardware partitioning approaches often result in underutilized hardware parallelism and suboptimal memory organization, which severely constrains the scalability of kernel implementations. Consequently, large-scale kernels fail to fully exploit their inherent data locality, resulting in frequent off-chip memory accesses and overhead from repeated invocations, thereby substantially degrading overall performance. To address these challenges, this paper introduces MC-CGRA, a memory-computation coordinated CGRA framework. By leveraging its novel Chain-of-Computation (CoC) model, which uniformly represents operations within kernels as stream nodes, MC-CGRA achieves seamless coordination between memory access and pipelined computation through a software-defined approach. The framework incorporates a stream-centric CGRA microarchitecture to minimize frequent data exchanges between large-scale stream computing kernels and off-chip memory. An MC-CGRA prototype with an 8×10 PE array has been implemented on the AMD/Xilinx VCU118 platform. Experimental results show that the prototype combines fast compilation with sustained high throughput for stream processing kernels of varying scales, underscoring its efficiency in real-time scenarios. The prototype attains an average performance of 29.73 GOPS, outperforming state-of-the-art solutions by $1.55 \times$ and $1.62 \times$ in FFT and FIR workloads, respectively.

Index Terms—CGRA, Domain-Specific Accelerator, Stream Processing

I. INTRODUCTION

The demand for efficient large-scale stream processing continues to grow across diverse domains such as wireless communications, multimedia, and scientific computing. Consequently, the design of accelerators tailored to the specific requirements of these applications has become a prominent research focus in recent years. Coarse-grained reconfigurable arrays (CGRAs), as emerging computing platforms, comprise 2-D arrays of instruction-level Processing Elements (PEs) interconnected via programmable on-chip networks [1]. By leveraging spatial computation, CGRAs achieve high performance and energy efficiency while supporting post-fabrication programmability, making them widely adopted for constructing domain-specific accelerators [2].

Traditional CGRA software stacks typically employ dataflow graph (DFG)-based mapping strategies, which unrolls hot-spot computation kernels into single-level loops and then maps them onto PEs [3]. However, this approach has significant limitations in large-scale stream processing, where the workload is more accurately modeled as a set of functions repeatedly executed

over infinite input data streams [4]. Based on this assumption, the scheduling requirements consisting of computation, memory access, and control flow are deterministic. Therefore, the software stack must generate periodic admissible schedules for CGRAs. Moreover, to ensure continuous data availability during computation, the schedule must efficiently manage frequent memory accesses and complicated control flows with varying patterns, which are not sufficiently captured by the conventional DFG-based CGRAs [5].

To address these challenges, numerous studies have focused on mapping strategies and architectural designs. Several CGRAs optimize the mapping of stream processing kernels by employing elastic scheduling techniques [6] [7]. The Stream-Dataflow model [8] decouples memory access from computation, supporting various direct or indirect memory access patterns. Canalis [5] provides an intuitive programming interface for describing data streams and uses a direct execution model to improve data reuse across the PE array, thereby eliminating global memory accesses. However, separate software and hardware partitioning limits opportunities for hardware parallelism and leads to suboptimal memory organization due to insufficient exploitation of data locality. Our motivation experiment demonstrates that for end-to-end acceleration of large-scale stream processing, performance overheads associated with memory access and loop control remain significant and tend to increase as the workload size increases.

We argue that coordinating streaming-centric computation nodes with decoupled memory access patterns during scheduling is essential to maximize hardware parallelism and achieve efficient memory layouts. To take advantage of this opportunity, we present MC-CGRA, a memory-computation coordinated CGRA framework for end-to-end large-scale stream processing. The main contributions of this work are as follows.

- We propose the Chain-of-Computation (CoC) model, which unifies memory accesses and computation operations as stream nodes within a Synchronous Data Flow Graph (SDFG), thereby facilitating the generation of periodic admissible schedules for CGRAs.
- We introduce the MC-CGRA framework, comprising a software stack and a stream-centric microarchitecture, both tailored to the requirements of the CoC model, thereby achieving seamless memory-computation coordination.
- We implement an MC-CGRA prototype on the AMD/Xilinx VCU118 platform, which achieves an average compilation time of 0.16s and delivers an average

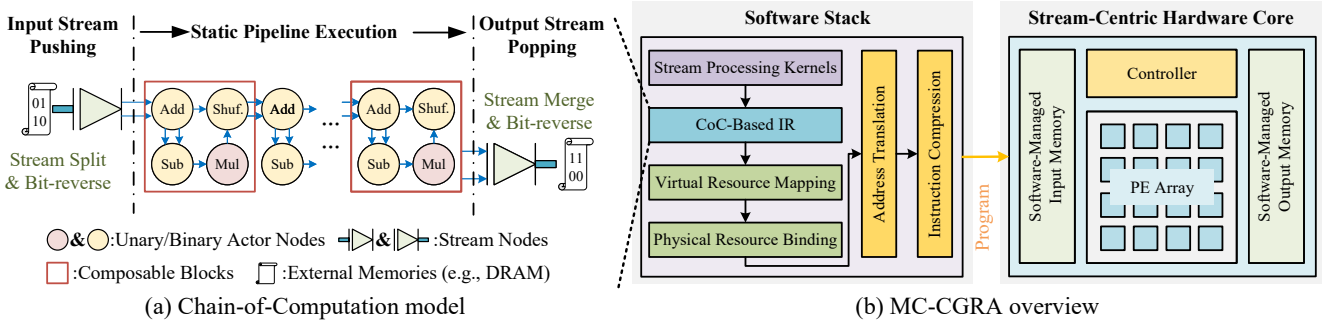


Fig. 3: MC-CGRA framework overview

access operations at arbitrary granularities. Building upon this model, we propose the MC-CGRA framework, which achieves higher parallelism and higher data reuse by coordinating memory and computation.

IV. CHAIN-OF-COMPUTATION MODEL

The CoC model formally defines stream processing kernels in a datatype-agnostic and hardware-agnostic manner, as illustrated in Fig. 3 (a).

A. Static Pipeline Execution

The CoC model encapsulates computations into discrete actors, utilizing lifetime metadata derived from access pattern descriptors (e.g., iteration bounds) to parameterize hardware counters. By leveraging the stream semantics of SDFGs, where node execution is purely triggered by data availability, the model eliminates the need for explicit control-flow mechanisms for inter-node synchronization. To optimize interface efficiency, the model unrolls and embeds constant arrays as internal immediates to bypass external ports. This enables the restriction of external operands to a maximum of two, minimizing runtime I/O overhead and consequently classifying actor nodes into two categories: Unary-Op and Binary-Op.

Subsequently, these nodes are interconnected to form an SDFG that enforces a strict 1:1 ratio between token generation and consumption rates on each arc. This inherent rate balance facilitates global synchronization within a uniform clock domain, thereby eliminating the necessity for supplemental cache FIFOs. By pre-planning delays for each PE input port, deterministic static scheduling is achieved without reliance on elastic flow-control techniques. Uniquely, the CoC model leverages this determinism to statically interleave floating-point operations of varying latencies, ensuring pipeline saturation and avoiding the reconfiguration overheads (i.e., pipeline flushes) inherent to dynamic operation switching.

B. Input Stream Pushing & Output Stream Popping

While the conventional SDFG representation presumes instantaneous data availability across all input ports, sustaining such throughput for fine-grained parallelism requires efficient caching of external data through optimized on-chip data layout configurations. For instance, the FFT computation shown in Fig. 3 (a) splits input data into dual substreams to achieve a throughput of two points per cycle, while necessitating data

permutations such as bit-reversal. This necessitates specialized streaming operations to transform continuous data streams originating from external memory or I/O sources into the required parallel format. To address this, the CoC model explicitly defines Input Stream Pushing and Output Stream Popping as stream nodes. These nodes encapsulate access parameters used to synthesize the intrinsic functions of the software-managed memory subsystem, enabling on-the-fly data layout transformations. Although the model abstracts away the specific hardware implementation of these nodes, it provides a precise architectural specification for the design of the underlying memory subsystems.

C. The Unique Characteristics of the CoC Model

First, the CoC model supports the aggregation of multiple atomic operations into composable blocks (CBs), such as the radix-2 FFT butterfly unit illustrated in Fig. 3(a). Incorporating stream access nodes into these CBs allows for higher parallelism via straightforward block instantiation. This hierarchical approach not only exploits the inherent synchronization of SDFGs but also provides robust encapsulation for advanced programming interfaces. Furthermore, by abstracting configuration details, the model supports rapid remapping for parameter updates or block alterations without requiring a full-graph recompilation. Second, the model facilitates explicit unrolling and node definition, enabling the folding of external constant arrays into internal immediate values of unary nodes. It also supports fused operations, such as constant multiply-and-add in FIR (shown in Fig. 1), aggregating them into efficient binary operators. Finally, the deterministic nature of the CoC model enables support for pipelining with branches. By utilizing lifetime metadata to pre-calculate timing synchronization, the model resolves complex control flows into a static, deterministic pipeline structure, thereby eliminating the need for dynamic flow control. These characteristics are essential for achieving efficient memory layouts and higher parallelism.

V. MC-CGRA DESIGN

A. Software Stack

As illustrated in Fig. 3 (b), the MC-CGRA software stack is designed to establish a unified development framework bridging the gap between high-level programming and hardware specific operations through software-defined approaches.

```

1 dispatcher→dispatch(m, n, k, STRIDE);
2 for (row = 0; row < tile_m; ++row) {
3   loader[row]→load(k, DIRECT);
4   for (col=0; col<tile_n; ++col) {
5     loader[tile_m + col]→load(k, REPEAT);
6     // CU directly forward input data to next PEs
7     pe[row][col]→forward(WEST, EAST);
8     pe[row][col]→forward(NORTH, SOUTH);
9     // FU execute the MAC operation
10    pe[row][col]→execute(MAC, k, OP_SRC_WEST, OP_SRC_NORTH);
11    // CU set the output datapath
12    pe[row][col]→outputRes(RESULT_CHANNEL);
13    // Align the input data arrival times and configure delay FIFOs
14    pe[row][col]→alignArrivalTimes();
15  }
16  storer[row]→store(tile_n, DIRECT);
17 }
18 integrator→integrate(tile_m, tile_n, TRANSPOSE);

```

Fig. 4: Pseudocode of a systolic array matrix multiplication

1) *Compiler Design*: The MC-CGRA compiler is built upon the LLVM framework, supporting stream processing kernels written in C/C++. To unify the compilation process, we define a customized Intermediate Representation (IR) derived from the CoC model. The backend workflow initiates by parsing the CoC IR to extract computational dependencies and stream access semantics, which are then mapped onto an idealized infinite virtual resource pool for initial scheduling and allocation. Subsequently, intrinsic functions are employed to bind these virtual mappings to physical hardware resources. Finally, the runtime environment generates the executable machine code through address translation and instruction compression.

2) *Static Branch Delay Matching*: The deterministic nature of the CoC model allows for aggressive, purely software-defined latency control. During the mapping phase, the compiler constructs a routing table for each PE, and uses a depth-first traversal strategy to calculate the cumulative path latency (i.e., data arrival time) for every PE input port. Subsequently, programmable delay FIFOs are configured to align the input data arrival times across different computational paths. This mechanism effectively resolves synchronization mismatches, seamlessly accommodating the non-uniform latencies characteristic of complex floating-point operations as well as diverse data path lengths.

3) *Hardware-Aware Runtime Environment*: As a system-level solution, MC-CGRA provides a hardware-aware runtime environment to support the long-term execution and dynamic reconfiguration of the hardware core. It maintains a software-level metadata repository tracking memory resource utilization and performs address translation from virtual instruction space addresses to physical hardware locations during kernel execution. Moreover, to avoid the transmission of duplicate instructions, the runtime environment tracks the Idle, Remain, and Update state of each module to identify and compress redundant instructions.

4) *Software-Hardware Programming Interface*: The intrinsic functions of MC-CGRA serve as the software-hardware programming interface for hardware modules, facilitating efficient lowering into low-level configuration instructions. As illustrated by the systolic array implementation in Fig. 4, these functions define module behaviors in a declarative, stream-centric paradigm, which is fundamental to constructing a data-driven hardware execution model. Acting as high-level abstractions,

these intrinsics not only retain fine-grained control over critical parameters but also automate cross-component coordination through software-level context propagation. Furthermore, they abstract away low-level hardware intricacies via automatic parameter inference, enabling the compiler to prioritize the mapping strategy rather than low-level resource management.

B. Stream-Centric Hardware Microarchitecture

1) *Global Controller*: As is illustrated in Fig. 5 (a), the global controller orchestrates the operational workflows defined by the CoC model. An instruction memory buffers the binary instruction packages generated by the software stack. The instruction decoder unit fetches and processes these instructions through a dedicated pipeline, parsing them based on designated instruction type fields. Subsequently, the decoded parameters are dispatched by the instruction issuer, which directs them to their respective target modules according to the identifier field extracted from each instruction.

2) *Software-Managed Memory Subsystem*: The memory subsystem establishes a consistent, programmable abstraction layer to standardize memory access patterns driven by CoC stream nodes. It comprises Stream Dispatcher/Integrator units equipped with private DMA engines, dedicated memory banks, Stream Load/Store units, and a specialized Stream Write-Back Channel. The Stream Dispatcher and Integrator orchestrate the movement of contiguous data to and from bank regions, adhering to the stream production and consumption requirements. These memory banks are optimized for streaming access, organized as circular buffers managed in a FIFO arrangement. The Stream Load/Store units are tasked with fetching inputs and committing outputs according to specific access patterns, while simultaneously handling on-the-fly pre-processing and post-processing. Furthermore, the Write-Back Channel is designed to recirculate intermediate results back into the PE array for iterative processing. This mechanism is pivotal for realizing the binding of virtual resource mappings to physical hardware, analogous to the approach in [14]. This decoupling of computational requirements from hardware constraints facilitates the scalable assembly of extended computational sequences.

Collectively, these components constitute a highly efficient and flexible memory subsystem capable of handling diverse affine and nonaffine data access patterns via the direct execution model introduced in [5]. Leveraging deterministic computation, this software-defined memory subsystem effectively eliminates the latency overhead associated with external memory interactions during large-scale data stream processing.

3) *PE Microarchitecture*: The PE architecture of MC-CGRA is tailored for high-efficiency stream processing, characterized by sustained continuity and high-precision complex computations. As illustrated in Fig. 5 (b), each PE comprises two core modules: the Function Unit (FU) and the Connection Unit (CU). Distinct from conventional designs, the FU supports dual-in-dual-out operations, enabling continuous stream transformations that significantly enhance parallelism and processing efficiency. The CU manages bidirectional communication across four orthogonal directions to ensure coordination between adjacent PEs. To accommodate Binary-Ops with dual operands

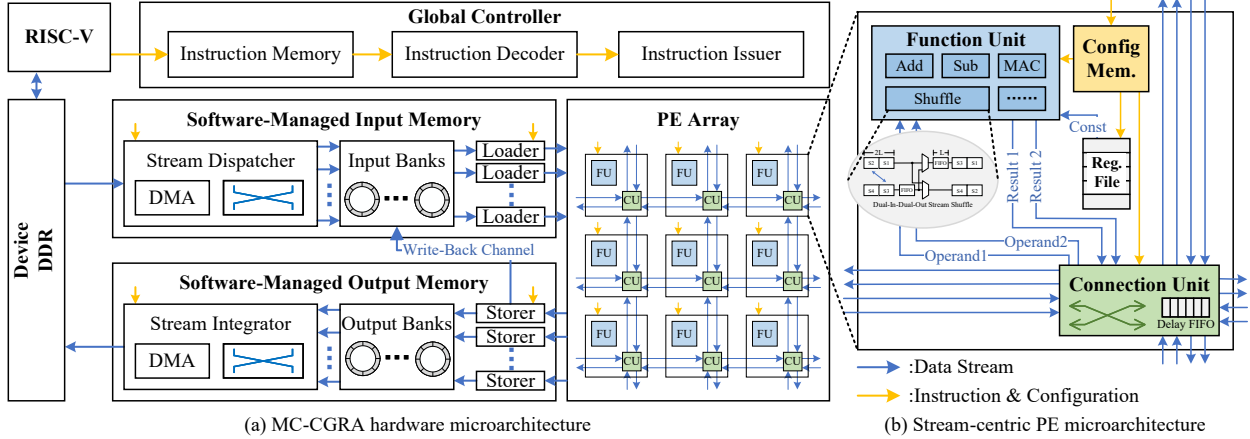


Fig. 5: Hardware overview of the MC-CGRA

while permitting simultaneous data forwarding, each direction is equipped with two 64-bit physical channels. Each channel is configurable to transmit either a single 64-bit complex FP32 token or two scalar FP32 tokens. These CUs constitute a lightweight bufferless Network-on-Chip (NoC) that relies entirely on software-defined static routing. This design strategy substantially reduces the complexity of algorithm mapping within the MC-CGRA framework.

VI. EVALUATION

A. FPGA Prototype Implementation

TABLE I: MC-CGRA FPGA prototype resource utilization

Resource	Used	Available	Utilization (%)
URAM	784	960	81.67
BRAM	1748.5	2160	80.95
DSP slice	2466	6840	36.05
FF	1397203	2364480	59.09
LUT	577589	1182240	48.86

The prototype system implementation of MC-CGRA employed the AMD/Xilinx VCU118 evaluation platform, which incorporates a Virtex UltraScale+ XCVU9P-L2FLGA2104E FPGA and dual 2GB DDR4 memory modules. It integrates an 8×10 PE array, with on-chip memory resources partitioned into 18 input banks and 8 output banks. Additionally, it leverages Xilinx Floating-Point Operator IP cores to enable both FP32 real and complex number computations. TABLE I summarizes the FPGA resource utilization.

B. Compilation Time Evaluation

We select typical stream processing kernels including FFT, FIR filtering, CFAR detection, and Matrix Multiplication (MM) to evaluate the real-time compilation time of MC-CGRA in large-scale scenarios. As summarized in TABLE II, the average compilation time measured on the embedded RISC-V processor is 0.16s. The compilation time increases proportionally with the number of PEs and the scale of the kernel. Notably, when the FIR tap count surpasses 80, the average compilation time exhibits a sharp increase. This increase is attributed to the

TABLE II: Compilation times on the embedded RISC-V

Kernel	Scale	PE Num.	Avg. Times
FFT	32 \rightarrow 512K Points	20 \rightarrow 80	0.15s
FIR	(<80 Taps)	= 1+Taps	0.14s
	(\geq 80 Taps)		0.24s
CFAR	4 \rightarrow 256 Reference Cells	8	0.07s
MM	64 \rightarrow 1024*	80	0.20s
Average	-	-	0.16s

* The k and n dimensions are fixed at 48 and 10 respectively.

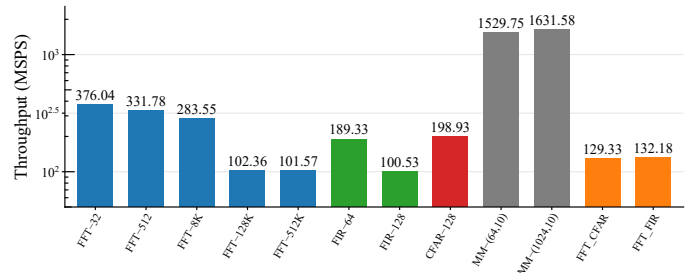


Fig. 6: Throughputs of MC-CGRA prototype system

saturation of physical computational resources, which necessitates iterative scheduling via the virtual mapping mechanism described in Section V-A1.

C. Throughput Evaluation

As shown in Fig. 6, we evaluate the throughput of the prototype system. The prototype dynamically adjusts FFT size from 32 point at 376.04 MSPS (Mega Samples Per Second) to 512K point at 101.57 MSPS. As the FFT point increase, throughput gradually decreases due to the need for more PE resources to support larger-scale FFT computations, resulting in increased pipeline initialization time. For FFT computations with very large scales, an increased demand for twiddle factors reduces the overall throughput efficiency. When the FIR tap exceeds the limitation, additional iterative calculations are required, resulting in a decrease in throughput (from 189.33 MSPS to 100.53 MSPS). For the CFAR detection benchmark configured with 128 reference cells, the system achieves a throughput of 198.93 MSPS. The throughput of MM increases

TABLE III: MC-CGRA performance comparison

	UE-CGRA [6]	HyCube [15]	RipTide [16]	Dual-DISO [17]	STRELA [18]	Riken* [7]	MC-CGRA* (this work)
Platform	ASIC	ASIC	ASIC	FPGA	ASIC	ASIC	FPGA
Array Size	8×8	4×4	6×6	8×8	4×4	8×8	8×10
Frequency	750MHz	50MHz	250MHz	300MHz	250MHz	288MHz	200MHz
Perf. (GOPS)	FFT	0.625	5.38	0.062	9	1.224	21.89
	FIR	—	—	—	—	30.24	48.85
	MM	—	—	0.164	4.5	0.438	25.47
	CFAR	—	—	—	—	—	10.74

* Performing floating-point operation.

with matrix size, driven by higher parallel resource utilization and the amortization of pipeline initialization overhead by the increased data volume.

We also evaluate the throughput of MC-CGRA during runtime kernel switching (FFT_CFAR and FFT_FIR). The results indicate that the overhead associated with switching leads to a decrease in throughput. However, the overall system performance remains within an acceptable operational range.

D. Performance Comparison

As summarized in TABLE III, we evaluated the performance of the MC-CGRA prototype system across various stream processing tasks, measured in GOPS, and compared it with existing CGRA architectures. The results demonstrate that MC-CGRA achieves significant performance improvements, even when performing FP32 complex number computations. MC-CGRA attains an average performance of 29.73 GOPS, achieving $1.55\times$ and $1.62\times$ higher GOPS compared to state-of-the-art solutions for FFT and FIR workloads, respectively. In MM tasks, MC-CGRA exploits the CoC model to efficiently unroll computational nodes and streams, achieving a throughput of up to 25.47 GOPS, substantially outperforming competing architectures.

VII. CONCLUSION

This paper presents MC-CGRA, a memory-computation coordinated CGRA framework designed for end-to-end large-scale stream processing. We introduce the Chain-of-Computation (CoC) model, which unifies the semantics of computation and memory access within SDFG-style stream nodes, thereby enabling the description of periodic admissible schedules for CGRAs. To materialize this model, we developed a stream-centric PE microarchitecture and a software-managed memory subsystem, supported by a hardware-aware compiler and runtime stack. Experimental results on an AMD/Xilinx VCU118 FPGA prototyping system demonstrate that MC-CGRA achieves millisecond-level compilation agility while delivering superior computational performance.

REFERENCES

- [1] K. Koul, J. Melchert, K. Sreedhar *et al.*, “Aha: An agile approach to the design of coarse-grained reconfigurable accelerators and compilers,” *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 2, pp. 1–34, 2023.
- [2] S. Liu, J. Weng, D. Kupsh *et al.*, “Overgen: Improving fpga usability through domain-specific overlay generation,” in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 35–56.
- [3] X. Mo, Y. Li, and D. Liu, “Optimizing imperfectly-nested loop mapping on cgras via polyhedral-guided flattening,” in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024, pp. 1–6.
- [4] E. Lee and D. Messerschmitt, “Synchronous data flow,” *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.
- [5] K.-Y. Chen, T. Mason Nelson, A. Khadem *et al.*, “Canalis: A throughput-optimized framework for real-time stream processing of wireless communication,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 17, no. 4, Nov. 2024.
- [6] C. Torng, P. Pan, Y. Ou *et al.*, “Ultra-elastic cgras for irregular loop specialization,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 412–425.
- [7] E. Del Sozzo, X. Wang, B. Adhi *et al.*, “Exploration of trade-offs between general-purpose and specialized processing elements in hpc-oriented cgra,” in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2024, pp. 668–680.
- [8] T. Nowatzki, V. Gangadhar, N. Ardalani *et al.*, “Stream-dataflow acceleration,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. New York, NY, USA: Association for Computing Machinery, 2017, p. 416–429.
- [9] E. A. Lee and D. G. Messerschmitt, “Static scheduling of synchronous data flow programs for digital signal processing,” *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24–35, 1987.
- [10] Y. Huang, P. Jenne, O. Temam *et al.*, “Elastic cgras,” in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 171–180.
- [11] J. Melchert, K. Feng, C. Donovan *et al.*, “Apex: A framework for automated processing element design space exploration using frequent subgraph analysis,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 33–45.
- [12] C. Tan, N. B. Agostini, T. Geng *et al.*, “Drips: Dynamic rebalancing of pipelined streaming applications on cgras,” in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 304–316.
- [13] J. Lou, X. Gao, Y. Mao *et al.*, “An agile deploying approach for large-scale workloads on cgra-cpu architecture,” in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024, pp. 1–6.
- [14] Y. Li, J. Zhu, Y. Fu *et al.*, “Circular reconfigurable parallel processor for edge computing : Industrial product,” in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 863–875.
- [15] M. Karunaratne, A. K. Mohite, T. Mitra *et al.*, “Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect,” in *Proceedings of the 54th Annual Design Automation Conference 2017*. New York, NY, USA: Association for Computing Machinery, 2017.
- [16] G. Gobieski, S. Ghosh, M. Heule *et al.*, “Riptide: A programmable, energy-minimal dataflow compiler and architecture,” in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 546–564.
- [17] A. K. Jain, D. L. Maskell, and S. A. Fahmy, “Coarse grained fpga overlay for rapid just-in-time accelerator compilation,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 6, pp. 1478–1490, 2022.
- [18] D. Vazquez, J. Miranda, A. Rodriguez *et al.*, “Strela: Streaming elastic cgra accelerator for embedded systems,” 2024. [Online]. Available: <https://arxiv.org/abs/2404.12503>