FLAME: A Framework Exploring Execution Strategies for Multi-Cycle Operations in CGRA

Jiajun Qin[1], Cheng Tan[2,3], Ruihong Yin[4], Tianhua Xia[5], Sai Qian Zhang[5], Bei Yu[1]
[1]The Chinese University of Hong Kong, [2]Google, [3]Arizona State University,
[4]University of Minnesota, [5]New York University

***Abstract*—Effective mapping of dataflow graphs onto Coarse-Grained Reconfigurable Arrays necessitates compiler-architecture co-design, yet existing approaches frequently assume single-cycle operations despite real-world applications often involving multi-cycle operations that constrain achievable clock frequencies. To address this, we propose *FLAME*, a novel framework supporting three execution strategies (exclusive, distributed, inclusive) specifically designed for multi-cycle operations, with co-designed compiler and hardware support. Our evaluations demonstrate that FLAME not only surpasses prior methods in performance and but also enables flexible exploration of these operations. The framework achieves average speedups of 2.21× over baseline CGRA and 1.49× over prior state-of-the-art framework while highlighting the distinct characteristics of each strategy.**

I. Introduction

Coarse-grained reconfigurable architectures (CGRAs) have been successfully deployed across multiple high-impact application domains [1]–[5]. However, efficient application deployment on CGRAs remains challenging. A major bottleneck lies in the compiler's ability to effectively map dataflow graphs (DFGs) onto the hardware fabric, which involves optimally assigning computational nodes to available processing tiles while adhering to resource, timing, and communication constraints.

In the majority of existing CGRA mapping methods [6]–[12], DFG nodes are typically modeled as single-cycle operations mapped to a single tile. Although this assumption simplifies the mapping process, it does not accurately capture real-world scenarios, where many operations incur substantially higher latencies. These delays most often arise in two cases: (1) complex arithmetic operations, such as division, and (2) fused operations, such as multiply-accumulate (MAC). Modeling them as single-cycle operations would introduce an excessively long critical path, severely constraining the attainable clock frequency. Hence, multi-cycle modeling and efficient mapping of such operations are essential to achieve practical timing.

To highlight the importance of multi-cycle operations, we examine representative kernels across embedded systems (`fir`, `latnrm`, `fft`), ML (`spmv`, `conv`, `relu`), and HPC (`histogram`, `mvt`, `gemm`) domains. By synthesizing Functional Units (FUs) tailored to these kernels, we classify each operation as either single-cycle or multi-cycle depending on its ability to satisfy timing constraints. As illustrated in Figure 1, forcing all operations into single-cycle execution prevents nearly half of the kernels from reaching 500 MHz. Achieving higher target frequencies, such as 1 GHz or 1.5 GHz, further demands efficient handling of multi-cycle operations according to their distribution within each kernel.

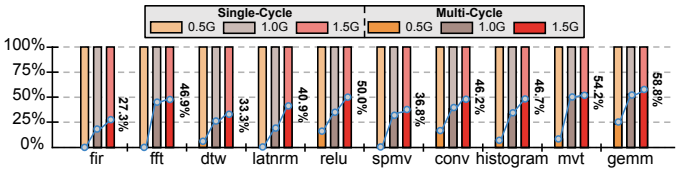

Fig. 1: Proportion of multi-cycle and single-cycle operations at different frequencies. Numbers indicate frequency in Hz.

Existing solutions to this challenge remain narrow in scope and limited in performance. For example, [13], [14] address only floating point operations by serializing multi cycle computations. Other methods [15]–[17] introduce pipeline registers to meet timing constraints. However, these approaches are either restricted to specific operation classes or experience performance losses due to inefficient resource usage, particularly when computations are confined to a single tile, which results in substantial underutilization of the CGRA fabric.

Effectively managing multi-cycle operations while optimizing resource utilization requires specialized execution strategies and a unified framework for mapping space exploration. This demands tight compiler-architecture co-design, significantly complicating the mapping process. To address this challenge, we propose *FLAME*: A Framework expLoring strAtegies for Multi-cyclE operations in CGRAs. FLAME introduces three optimized execution strategies, namely exclusive, distributed, and inclusive, each tailored for different application scenarios. Our main contributions are summarized as follows:

- We summarize three general purpose execution strategies for managing diverse multi cycle operations, with each strategy optimized for particular application scenarios.
- We propose a novel framework, *FLAME*, which combines a compiler toolchain mapping DFGs on a reconfigurable architecture for efficient execution based on the proposed strategies. It can additionally determine the optimal execution strategy within a given application.
- We demonstrate through comprehensive evaluation that FLAME delivers significant performance gains, achieving substantial speedups over existing approaches while enabling flexible exploration of multi cycle operation execution strategies.

This work is partially supported by The Research Grants Council of Hong Kong SAR (No. CUHK14211824 and No. CUHK14210723).

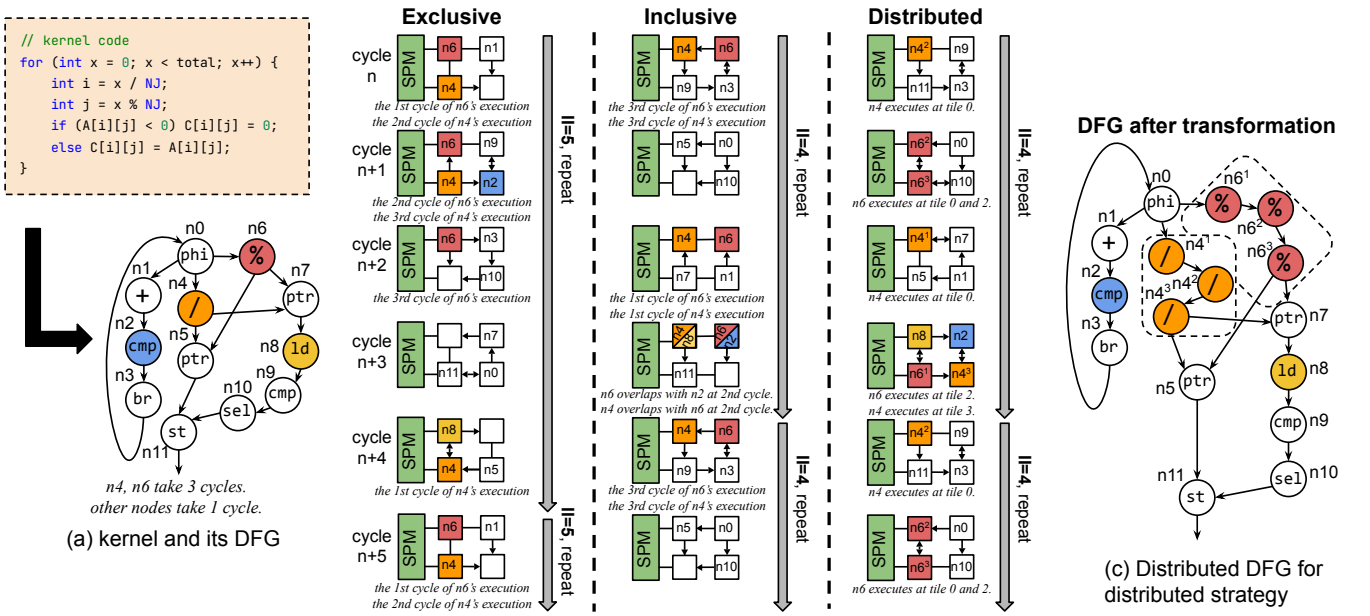

Fig. 2: An example of mapping a kernel onto a 2×2 CGRA, where division (nodes 4 and 6) takes 3 cycles and others are single-cycle. Exclusive and inclusive execution use the original DFG in (a), while distributed execution uses the modified DFG in (c). For clarity, nodes like address calculations before load operations are omitted. SPM denotes scratchpad memory.

II. Background and Motivation

A. Conventional CGRA Architecture and Mapping

CGRAs are programmable accelerators consisting of an array, integrated with host CPUs in either a loosely coupled [13], [17]–[20] or a tightly coupled [21]–[24] manner. In our design, we focus on the loosely coupled approach, where the CGRA operates as an independent accelerator and the CPU is responsible for data movement and instruction dispatch.

A CGRA is typically composed of multiple tiles arranged in a grid, with each tile containing FUs for performing operations, register files for temporal data storage, configuration memory, and on-chip interconnects for communication with other tiles or memory. The compiler produces the configuration signals that direct the tiles each cycle to read their configurations and execute corresponding operations, while also controlling inter-tile and memory routing. Notably, although each tile includes multiple FUs, conventional CGRA designs allow only one FU's output to be used as the tile's result per cycle, discarding the outputs of all other FUs regardless of their computation.

Given an application kernel, the compiler generates its DFG and maps the DFG nodes to the CGRA tiles for computation and data routing. The initial interval (II), calculated by the compiler, represents the number of cycles between the initiation of sequential loop iterations. For loops with a large number of iterations, the II heavily influences the overall execution latency.

B. Different Multi-Cycle Execution Strategies

Given the prevalence of multi-cycle operations in applications shown in Figure 1, a key challenge lies in how to execute these operations efficiently. Figure 2 presents an example of mapping a kernel onto a 2×2 CGRA, where Figure 2 (a) depicts the kernel's source code alongside its compiled DFG, while Figure 2 (b) illustrates the corresponding DFG placement generated by the compiler. We next summarize three strategies for supporting multi-cycle execution. These strategies are illustrated here with the example to provide a comparative overview.

Exclusive - The simplest approach to handling multi-cycle operations is to allocate a dedicated tile throughout its execution. However, this method keeps the tile occupied until completion, leaving other FUs idle and preventing the processing of subsequent control signals. While this mapping strategy is simple to realize, its performance is fundamentally constrained by the latency of multi-cycle operations. As illustrated in Figure 2, the resulting II is five cycles, indicating that a new iteration can only begin after five cycles. During cycles n through $n+2$, the remainder operation (n6) occupies Tile 0 exclusively, preventing additional node assignments and leaving the other FUs within the tile underutilized.

Distributed - Another strategy is to decompose multi-cycle operations into a sequence of single-cycle sub-operations within the DFG, enabling their distribution across multiple tiles. For example, a division that requires three cycles can be represented as three consecutive single-cycle nodes, as shown in Figure 2(c). In this case, the approach achieves an II of four, an improvement over exclusive execution, while the execution trace in Figure 2(b) illustrates that all tiles remain active in every cycle. Nonetheless, when applied to already large DFGs, this expansion can lead to suboptimal mapping results due to excessive growth in graph complexity which we will discuss further in Section V-B.

Inclusive - The idle resources in exclusive execution offer optimization potential. Observing that operations only require input ports during initialization and output ports upon comple-

tion, an inclusive execution strategy that utilizes intermediate cycles for parallel computation is possible. When a tile starts a multi-cycle operation, it sends inputs to the FU and marks the operation as in-progress, allowing the tile to process subsequent control signals in the next cycle. Throughout execution, additional operations can be performed since input and output ports remain available. Upon original operation completion, the tile outputs the result. This approach significantly improves tile utilization without altering the DFG. As shown in Figure 2(b), during cycle $n+3$ while nodes n4 and n6 are ongoing, nodes n8 and n2 are mapped to Tile 0 and Tile 1 respectively, achieving an II of 4 without exhausting all resources.

Each of the three strategies provides unique advantages for specific application scenarios; however, unifying them within a single framework introduces substantial implementation challenges. Our proposed FLAME framework overcomes these difficulties by offering integrated and comprehensive support for all three execution strategies.

C. Related Works

Conventional compilers also address multi-cycle operations during code generation, primarily through instruction scheduling. However, these techniques are typically applied only at the compiler level, without explicit consideration of hardware constraints. In the context of CGRA mapping, systematic methods for handling multi-cycle operations remain largely underexplored. Several prior works employ graph transformation techniques to serialize multi-cycle operations [13], [14], [25], thereby mitigating some of the associated challenges in a way that resembles FLAME's distributed execution strategy. However, these approaches are limited to floating-point operations. Alternative strategies address latency, for example, REVAMP [16] hides memory delays through pipelined data streaming, while FLEX [26] reduces the cost of address calculations. CGRA-ME2 [17] leverages pipeline optimizations for floating-point units, and APEX [15] improves frequency via register insertion in fused operations. However, pipelining is less effective for spatial–temporal CGRAs, where tile reconfiguration and intermittent inputs cause pipeline underutilization.

In contrast to prior work, FLAME extends both distributed and exclusive strategies to a broader range of applications and further introduces a novel inclusive strategy that significantly enhances performance and resource utilization. Moreover, the framework enables flexible exploration of execution strategies, allowing the compiler–hardware co-design to automatically select the most suitable approach.

III. FLAME Compiler

We develop a compiler toolchain built on LLVM [27] to support comprehensive mapping of multi-cycle operations. As illustrated in Figure 3, the compiler processes kernel code by applying loop transformations (Section III-A) and DFG manipulations (Section III-B). In addition, it integrates a specialized optimization phase for multi-cycle operations (Section III-C), which evaluates alternative execution strategies and automatically selects the most effective approach in accordance with user-defined performance objectives.

A. Loop Transformation

The loop transformation stage applies two principal techniques, *loop flattening* and *loop unrolling*, that restructure kernels to improve instruction-level efficiency and expose greater parallelism during execution.

Loop Flattening - For nested loops, flattening converts multiple loop levels into a single-level loop structure, eliminating repeated inner loop initialization overhead. This transformation typically inserting division and remainder operations to decompose the nested loop hierarchy.

Loop Unrolling - Loop unrolling replicates the loop body to expand the kernel, increasing the size of DFG and enabling more effective utilization of resources. By exposing additional concurrent operations, this technique can significantly improve performance on reconfigurable fabrics.

B. DFG Manipulation

After loop transformations, FLAME compiler proceeds to generate and refine DFGs during the DFG manipulation phase.

DFG Generation - Given kernel code, we generate its corresponding DFG where each DFG node represents one instruction in LLVM IR, with control-flow instructions converted to data-flow through partial prediction [28].

DFG Tuning - In this phase, frequent computation patterns are fused into single composite nodes, either based on user-defined specifications or through the compiler's automatic detection of common patterns, such as MAC operations.

C. Multi-Cycle Strategy Optimization

FLAME allows users to either explore or specify their preferred execution strategies. For those targeting a specific objective such as performance, the framework automatically generates and evaluates all three strategies and selects the optimal mapping based on their outcomes. If a specific strategy is chosen, the compiler applies only that strategy during mapping. As shown in Figure 3, our compiler processes the tuned DFG through cycle-aware optimization, which initializes node properties and generates a distributed DFG, followed by strategy generation that maps DFGs onto the CGRA, and finally strategy selection via hardware simulation and synthesis to evaluate statistics and determine the optimal approach.

Cycle-Aware Optimization - Following DFG modification, FLAME configures node properties by assigning execution latencies to multi-cycle nodes and specifying their pipelining capabilities, as described in Section IV. These parameters can be manually provided by users or obtained from post-synthesis timing analysis. The DFG is then transformed into a distributed representation, where multi-cycle nodes are broken down into multiple sub-nodes to support our distributed execution strategy.

Strategy Generation - Given DFGs optimized for multi-cycle operations, our compiler then maps them onto CGRAs under various strategies to generate control signals that guide CGRA execution. Our framework utilizes a heuristic optimization algorithm that maps the DFG onto the CGRA's Modulo Routing Resource Graph [29]. The algorithm begins with the theoretical lower bound of II, calculated as the maximum value between the resource constrained minimum II and recurrence constrained minimum II. It then iteratively increases the II value

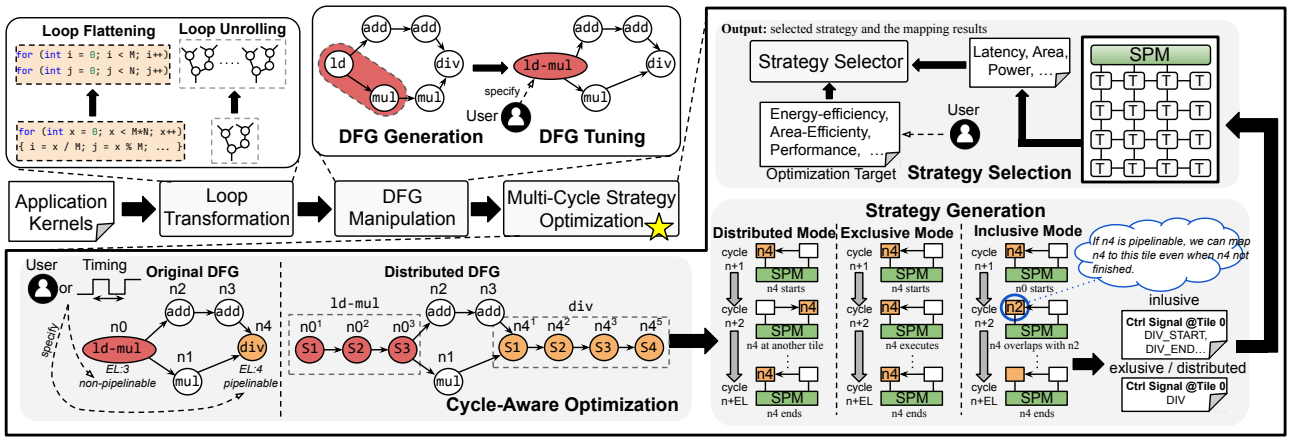

Fig. 3: Overall compiler toolchain of FLAME. EL denotes execution latency.

until a valid hardware mapping is found, while consistently working to minimize the final achieved II.

To support different strategies, the FLAME mapping algorithm evaluates whether a DFG node opt can be allocated to a tile beginning at a specified cycle under a given II. Tile occupancy is categorized into four states: `SO` for single-cycle operations, and `SPO`, `IPO`, and `EPO` denoting the start, intermediate, and end stages of multi-cycle operations, respectively.

In the exclusive execution strategy, the II is configured to exceed the maximum node latency, with tiles assigned to multi-cycle operations marked as occupied throughout their duration to prevent resource conflicts. The distributed strategy follows a similar logic but involves pre-decomposition of multi-cycle operations during scheduling. Under the inclusive strategy, placement restrictions are applied based on operation type: single-cycle operations cannot be placed on tiles marked with `SO`, `SPO`, or `EPO` due to port contention, while multi-cycle operations avoid `SO` and `SPO` during their start stages and `SO` and `EPO` during end stages, as they don't concurrently occupy both input and output ports. Furthermore, non-pipelined multi-cycle operations sharing FUs cannot be scheduled on the same tile during overlapping cycles.

Upon the mapping process, the framework produces the final II along with tile-specific control signals to coordinate operation execution. In inclusive execution strategy, these control signals incorporate additional `OPT_START` and `OPT_END` markers (where `OPT` identifies particular multi-cycle operations) to explicitly indicate the initiation and completion points of multi-cycle operations, as will be elaborated in Section IV.

Strategy Selection - With the control signals for each tile obtained, we simulate CGRA execution to determine latency and synthesize the tiles with FUs under different strategies to evaluate power consumption, area usage, and other metrics. As shown in Figure 3, these results are fed into our strategy selector, which automatically selects the optimal configuration based on the user-specified optimization objectives, then outputs both the chosen strategy and corresponding mapping results. Alternatively, if a specific strategy is explicitly chosen instead of automated exploration, the framework directly produces the corresponding mapping output.

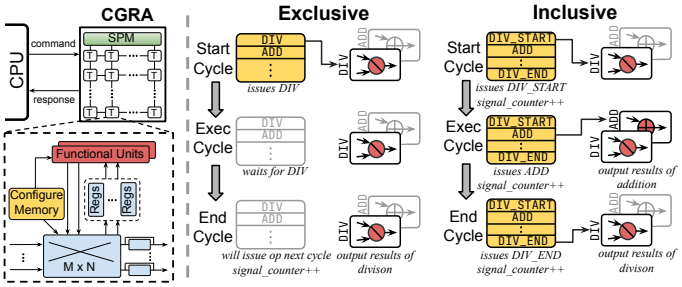

Fig. 4: FLAME architecture and its execution process.

IV. FLAME Architecture

FLAME's architecture, shown in Figure 4, is based on a general N×M CGRA prototype loosely coupled to the CPU via an accelerator interface. Each tile integrates FUs, crossbars, and configuration memory. The topmost tiles include dedicated load/store units to access SPM. Tile count, port count per tile, and crossbar sizes are all fully parameterizable.

To enable exclusive execution, we incorporate logic into tiles to ensure they maintain their current state during multi-cycle computations and only fetch subsequent control signals after finishing previously mapped operations. As shown in Figure 4, multi-cycle operations produce valid data only upon completion, prior to which the tile is solely engaged in that operation. The system automatically increments the signal counter once finished, allowing tiles to proceed to the next control signal in the subsequent cycle.

For distributed execution, hardware execution is the same as the conventional CGRA structure, as all modifications are managed at the compiler level. This strategy provides the dual benefit of enabling resource-efficient implementations, as each split node executes only a portion of the original multi-cycle computation, allowing tiles to incorporate simplified hardware components rather than complete functional units. For example, distributing a 4-cycle operation across four tiles means each tile only requires the hardware specific to its assigned stage, significantly reducing overall resource requirements.

For inclusive execution, our architecture also adds control logic to handle the `OPT_START` and `OPT_END` signals. As illustrated in Figure 4, operations are treated as single-cycle

Application	Kernel Description
ResNet	Convolution (k1); BatchNorm (k2, k3); ReLU (k4)
Attention	Linear projection and matmul (k1); Softmax (k2, k3); Mask and normalization (k4)
Harris Corner	Gradient computation (k1); Gaussian smoothing (k3) Gradient squared and product Computation (k2); Harris response calculation and thresholding (k4);

TABLE I: Applications used for our evaluation. k1 denotes the first kernel of the application. Some operations like BatchNorm requires multiple kernels for their computations.

from the tile's perspective, which means the tile assumes each operation completes in one cycle, allowing it to proceed without stalling ongoing multi-cycle computations and thereby facilitating overlap between operations. Upon receiving `OPT_START`, the tile initiates execution by supplying inputs to the FU, while `OPT_END` triggers the final result selection. Our framework maintains full compatibility across all execution strategies, differing only in the compiler-generated control signals: exclusive execution uses signals that cause tiles to wait for FU completion, whereas inclusive execution utilizes `OPT_START` and `OPT_END` to support overlapped execution.

In inclusive execution, pipelines constitute a special case. While inclusive execution typically overlaps operations of different types requiring different FUs, pipelined operations of the same type can still overlap within our framework. This requires adding pipeline registers to the FU to buffer intermediate results each cycle, allowing it to accept new inputs before the current computation finishes. Without these registers, the FU can only handle one operation at a time. This behavior is user-specified, with further analysis provided in Section V-D.

V. Experimental Evaluation

A. Experimental Setup

To assess the proposed framework, we employ benchmarks spanning multiple domains, including embedded systems, machine learning, and HPC. For embedded systems, we evaluate digital signal processing kernels, including finite impulse response filters (`fir`), normalized lattice filters (`latnrm`), fast Fourier transforms (`fft`), and dynamic time warping (`dtw`). For machine learning workloads, we use sparse matrix–vector multiplication (`spmv`), convolution (`conv`), and rectified linear unit (`relu`). HPC benchmarks include histogram computation (`histogram`), matrix–vector product with transpose (`mvt`), and generalized matrix multiplication (`gemm`). In addition, we assess full applications such as ResNet, attention mechanisms, and Harris corner detection (see Table I for details). To improve CGRA utilization, all benchmarks are transformed through loop flattening and unrolling to generate large DFGs.

We implement the 4×4 FLAME CGRA in RTL using synthesizable Verilog generated from Pymtl3 [30] and synthesize it with Synopsys Design Compiler [31] using a 45 nm TSMC library for area and power estimation. The FLAME compiler framework is built on top of the LLVM [27] infrastructures.

B. Effect of Different Strategies

Figure 5 compares three strategies across various kernels. Through our synthesis of FUs at 1GHz, we identify memory accesses, multiplications, and divisions as multi-cycle operations. The results reveal distinct trade-offs for each approach:

The exclusive strategy uses the simplest mapping logic but suffers significant performance degradation at low unrolling factors, where multi-cycle operations create critical path bottlenecks and cause severe resource underutilization. With small DFGs, division operations take at least 9 cycles, blocking subsequent operations and leading to an II of at least 9. Increasing the unrolling factor enhances performance as larger DFGs offer more operations to better utilize available resources.

The distributed strategy demonstrates strong performance on smaller DFGs by distributing multi-cycle operations across tiles, as shown in kernels such as `conv` and `hist` in Figure 5. It achieves comparable results to the inclusive strategy at low unrolling factors with manageable DFG sizes. However, its efficiency decreases for larger workloads like `mvt` and `gemm`, where DFGs exceeding 180 nodes introduce increased mapping complexity and compromise performance gains. It should be noted that even state-of-the-art mapping algorithms for CGRAs face greater challenges in optimizing larger DFGs, and the distributed strategy will exacerbate this issue.

Inclusive strategy consistently achieves near-optimal performance across all benchmarks, with an average speedup of 1.76× and 1.28× over exclusive and distributed strategy. It maintains high resource utilization without incurring the scalability limitations of the distributed approach, making it well-suited for most practical scenarios.

Our evaluation also covers kernels like `fir` and `latnrm`, which contain relatively few multi-cycle operations (under 20% of the total) at 1GHz, as shown in Figure 1. In these cases, the other strategies provide limited improvements over the exclusive approach, so we excluded these kernels from results.

In summary, each strategy targets different workload needs: the exclusive strategy offers simplicity and delivers competitive performance with limited multi-cycle operations; the distributed strategy suits medium-sized DFGs; the inclusive strategy serves as a versatile default for complex workloads. The optimal choice depends on DFG size and operation characteristics, with the inclusive strategy providing the most favorable for typical scenarios. Our FLAME framework supports exploring the optimal strategy based on application requirements.

C. Effect of Frequency

Figure 6a shows the speedup achieved at various operating frequencies in a setup where division is the only multi-cycle operation. The results indicate that performance improvements over exclusive execution scale with frequency, as higher clock rates raise the cycle count of multi-cycle operations, which leaves FLAME more space for optimization. Conversely, at lower frequencies such as 200MHz where divisions complete in only two cycles, performance gains are constrained and the inclusive strategy converges toward the exclusive approach with identical speedups. Notably, even the exclusive strategy running at higher frequencies can outperform all strategies operating at lower frequencies, underscoring the importance of targeting relatively high clock rates. Elevated frequencies lead to increased cycle counts of multi-stage operations, while

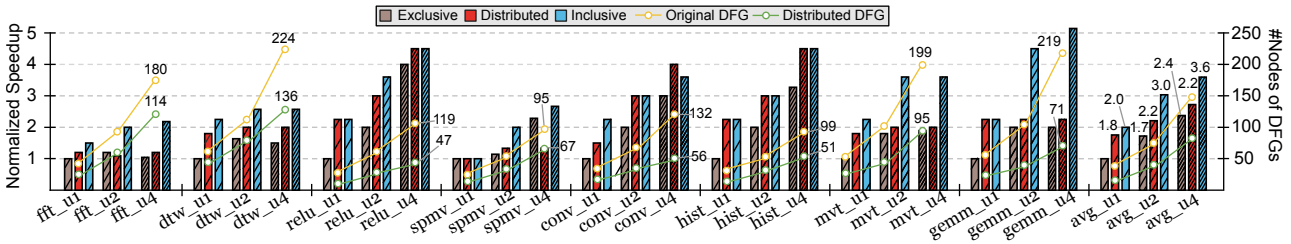

Fig. 5: Normalized speedup over exclusive execution (without loop unrolling) and total DFG node counts, with u2 denoting an unrolling factor of 2 and similarly for others, and hist denoting histogram. The inclusive strategy incorporates pipelining.

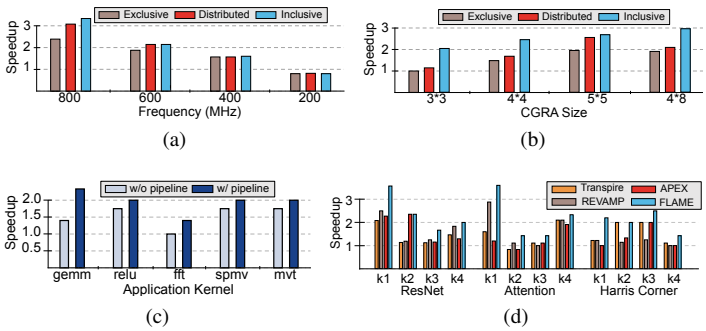

Fig. 6: Normalized speedup of FLAME. The kernels evaluated in (a), (b), and (c) are the same as those in Figure 5.

FU	Area (μm^2)	Power (μW)
Exclusive FU	811	108
Distributed FU	382	59
Inclusive FU	1889	1860
Tile	**Area (μm^2)**	**Power (μW)**
Tile with Exclusive FU	57218	22800
Tile with Distributed FU	56808	22700
Tile with Inclusive FU	58225	24500

TABLE II: Power and area consumption of a division FU under different strategies, along with the tile configured with corresponding FUs, where the evaluated tile supports all basic operations like addition and comparison.

FLAME strategically leverages this characteristic to achieve optimized performance, demonstrating its capability to support higher clock frequencies while maintaining strong performance.

D. Effect of Pipelining

Figure 6c demonstrates the performance improvement achieved through the pipelining technique described in Section IV, under two conditions: the DFG must be sufficiently large to expose resource bottlenecks that justify pipelining, and numerous operations must share the same FU. These criteria are met in kernels with high unrolling factors, where our method yields an average speedup of 1.30×. While previous works [15], [17] often employ pipeline registers to manage multi-cycle operations, our evaluation shows that such registers are necessary only when the aforementioned conditions are satisfied, and only limited kernels can benefit from them.

E. Scalability

Our scalability evaluation of FLAME (Figure 6b) reveals nuanced performance characteristics as CGRA size expands. Speedup improvements do not scale linearly with array size, as the 4×8 configuration achieves only a modest 1.4× (<2×) speedup over the 4×4 design, primarily constrained by current compiler mapping limitations. This finding suggests that partitioning may be more effective than monolithic scaling up for larger CGRAs.

F. Effect of Fusion

To validate the practical effectiveness of FLAME, we evaluate three multi-kernel applications that contain numerous frequent operation patterns suitable for fusion. We compare FLAME with three baselines employing distinct approaches for multi-cycle operations—APEX [15], REVAMP [16], and Transpire [25]. We reproduce these works and use the same set of patterns for fair comparisons.

As shown in Figure 6d, FLAME achieves speedups of 1.53×, 1.49×, and 1.60× over Transpire, REVAMP, and APEX, respectively, across the three applications. This improvement occurs because fusing multiple operations often produces large patterns with high latency, which would otherwise impose strict II constraints. FLAME's multi-cycle strategies enable aggressive operation fusion by fully utilizing hardware resources even under long-latency operations. These findings also highlight that fusion optimization should consider not only DFG patterns but also hardware compatibility, where our FLAME framework offers a flexible and comprehensive solution to explore this.

G. Hardware Evaluation

We evaluate the area and power consumption of our multi-cycle FUs at 1GHz. Table II presents the hardware costs for distributed, exclusive, and inclusive FUs along with their corresponding tiles, demonstrating that the distributed approach significantly reduces hardware costs.

VI. Conclusion

This work proposes FLAME, which enables efficient mapping of multi-cycle operations on CGRAs via multiple execution strategies. It leverages compiler-architecture co-design to unify hardware flexibility and strategy optimization. Evaluation shows that FLAME systematically explores strategy trade-offs, facilitating efficient deployment across diverse applications.

REFERENCES

[1] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "Dsagen: Synthesizing programmable spatial accelerators," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 268–281.

[2] J. Qin, T. Xia, C. Tan, J. Zhang, and S. Q. Zhang, "Picachu: Plug-in cgra handling upcoming nonlinear operations in llms," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025, pp. 845–861.

[3] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, "Xilinx adaptive compute acceleration platform: Versaltm architecture," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 84–93.

[4] Y. Luo, C. Tan, N. B. Agostini, A. Li, A. Tumeo, N. Dave, and T. Geng, "Ml-cgra: an integrated compilation framework to enable efficient machine learning acceleration on cgras," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[5] C. Tan, N. B. Agostini, T. Geng, C. Xie, J. Li, A. Li, K. J. Barker, and A. Tumeo, "Drips: Dynamic rebalancing of pipelined streaming applications on cgras," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 304–316.

[6] C. Tan, D. Patil, A. Tumeo, G. Weisz, S. Reinhardt, and J. Zhang, "Vecpac: A vectorizable and precision-aware cgra," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023, pp. 1–9.

[7] Z. Li, P. Dangi, C. Yin, T. K. Bandara, R. Juneja, C. Tan, Z. Bai, and T. Mitra, "Enhancing cgra efficiency through aligned compute and communication provisioning," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025, pp. 410–425.

[8] D. Wijerathne, Z. Li, A. Pathania, T. Mitra, and L. Thiele, "Himap: Fast and scalable high-quality mapping on cgra via hierarchical abstraction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 10, pp. 3290–3303, 2021.

[9] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "Cgra-me: A unified framework for cgra modelling and exploration," in *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)*. IEEE, 2017, pp. 184–189.

[10] C. Tan, C. Xie, A. Li, K. J. Barker, and A. Tumeo, "Opencgra: An open-source unified framework for modeling, testing, and evaluating cgras," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 2020, pp. 381–388.

[11] ——, "Aurora: Automated refinement of coarse-grained reconfigurable accelerators," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1388–1393.

[12] C. Tan, M. Jiang, D. Patil, Y. Ou, Z. Li, L. Ju, T. Mitra, H. Park, A. Tumeo, and J. Zhang, "Iced: An integrated cgra framework enabling dvfs-aware acceleration," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2024, pp. 1338–1352.

[13] R. Prasad, S. Das, K. J. Martin, and P. Coussy, "Floating point cgra based ultra-low power dsp accelerator," *Journal of Signal Processing Systems*, vol. 93, no. 10, pp. 1159–1171, 2021.

[14] S. Das, R. Prasad, K. J. Martin, and P. Coussy, "Energy efficient acceleration of floating point applications onto cgra," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2020, pp. 1563–1567.

[15] J. Melchert, K. Feng, C. Donovick, R. Daly, R. Sharma, C. Barrett, M. A. Horowitz, P. Hanrahan, and P. Raina, "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*, 2023, pp. 33–45.

[16] T. K. Bandara, D. Wijerathne, T. Mitra, and L.-S. Peh, "Revamp: A systematic framework for heterogeneous cgra realization," in *Proceedings of the 27th ACM international conference on architectural support for programming languages and operating systems*, 2022, pp. 918–932.

[17] O. Ragheb, S. Wicklund, M. Walker, R. Beidas, A. Ragab, T. Yu, and J. Anderson, "Cgra-me 2.0: A research framework for next-generation cgra architectures and cad," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2024, pp. 642–649.

[18] C. Sunny, S. Das, K. J. Martin, and P. Coussy, "Hardware based loop optimization for cgra architectures," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2021, pp. 65–80.

[19] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE transactions on computers*, vol. 49, no. 5, pp. 465–481, 2000.

[20] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "Piperench: A reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, 2000.

[21] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings, "A reconfigurable arithmetic array for multimedia applications," in *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, 1999, pp. 135–143.

[22] Mirsky and DeHon, "Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *1996 Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*. IEEE, 1996, pp. 157–166.

[23] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, 2012.

[24] J. D. Souza, L. Carro, M. B. Rutzig, and A. C. S. Beck, "A reconfigurable heterogeneous multicore with a homogeneous isa," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 1598–1603.

[25] R. Prasad, S. Das, K. J. Martin, G. Tagliavini, P. Coussy, L. Benini, and D. Rossi, "Transpire: An energy-efficient transprecision floating-point programmable architecture," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 1067–1072.

[26] T. K. Bandara, D. Wu, R. Juneja, D. Wijerathne, T. Mitra, and L.-S. Peh, "Flex: Introducing flexible execution on cgra with spatio-temporal vector dataflow," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.

[27] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[28] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Branch-aware loop mapping on cgras," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6.

[29] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," in *2003 Design, Automation and Test in Europe Conference and Exhibition*, 2003, pp. 296–301.

[30] S. Jiang, P. Pan, Y. Ou, and C. Batten, "Pymtl3: A python framework for open-source hardware modeling, generation, simulation, and verification," *IEEE Micro*, vol. 40, no. 4, pp. 58–66, 2020.

[31] P. Kurup and T. Abbasi, *Logic synthesis using Synopsys®*. Springer Science & Business Media, 1997.