STEP: Spatially Threaded Execution Pipeline

Ang Da Lu
angl7@uw.edu
University of Washington
Seattle, Washington, USA

Jiayi Wang
jwang710@uw.edu
University of Washington
Seattle, Washington, USA

Ang Li
angliz@uw.edu
University of Washington
Seattle, Washington, USA

Abstract

Coarse-Grained Reconfigurable Arrays (CGRAs) accelerate parallel workloads by executing programs across spatially distributed processing elements. Existing compilation approaches rely on modulo scheduling over modulo resource routing graphs (MRRGs), enforcing fixed initiation intervals and deterministic memory latency. This restricts execution to small, regular kernels and forces manual program partitioning, limiting scalability and flexibility.

We propose Spatially Threaded Execution Pipeline (STEP), an execution paradigm that enables arbitrarily long vectorizable programs on CGRAs while tolerating variable-latency memory. STEP decouples control flow from cyclic reconfiguration, organizing instruction-level operations into predicate-dependent configurations executed on single-context processing elements. Execution is driven by a program counter, not modulo cycles, allowing threads to span configuration boundaries without violating dependencies.

STEP's compiler operates in three stages: global partitioning of control and dataflow graphs, architecture-aware spatial mapping under heterogeneous constraints, and target-specific routing. By eliminating cyclic scheduling, STEP supports longer, irregular programs and improves programmability without sacrificing performance, pointing toward more flexible, general-purpose CGRA execution.

1 Introduction

Coarse-Grained Reconfigurable Arrays (CGRAs) execute programs on a spatial array of processing elements (PEs), often arranged in a grid, Figure 3(a). A pipeline chain of PEs and switches are statically configured by a configuration bitstream. Conventional CGRA architectures [1, 2, 4] adopt a modulo resource routing graph (MRRG) [3] scheduling approach, by software pipelining the innermost loops. Figure 3(b-d) illustrates a mapped dataflow graph (DFG) on a flattened CGRA where PE operations advance in a round robin fashion across space and time.

To maintain high performance without stalling downstream pipelines, modulo based CGRAs rely on single-cycle scratchpad memories, limiting them to small, regular kernels with deterministic memory latency, Figure 3(f). As a result, developers decompose source programs into fixed-sized kernels that fit the CGRA prior to compilation, frequently introducing suboptimal cut points. During compilation, dataflow edges are placed and routed across time seeking to minimize inter-node distances. Explicit partitioning is unnecessary, as the mapping already spans time and execution proceeds via cycle-by-cycle reconfiguration.

In contrast, Spatially Threaded Execution Pipeline (STEP) uses single-context PEs and partitions DFGs into predicate-dependent configurations. This approach enables program-counter–style execution at the granularity of full CGRA configurations rather than individual instructions. STEP further introduces the notion of a logical thread, a vectorizable program element within a loop iteration that carries its own execution state, including registers and predicates.

Under this formulation, STEP enables program-counter-style execution at the granularity of full CGRA configurations rather than individual instructions. STEP (a) generates minimal configuration partitions with optimized intra-configuration operation assignment, and (b) pipelines complete logical threads along configuration boundaries, deferring memory-dependent operations into subsequent configurations to tolerate variable memory latency while preserving control- and data-dependent edges. To realize this, STEP decomposes CGRA compilation into three stages:

Partitioning: Source-level Control-Dataflow Graphs (CDFGs) are analyzed holistically before kernel or function decomposition. Instruction-level operations are assigned across CGRA configuration boundaries using a unified intermediate representation supporting multiple target architectures.

Mapping: Traditional computer-aided design techniques, such as simulated annealing, spatially place operations within each configuration to minimize long-distance nets and maximize spatial pipelining. Target-specific constraints including memory port organization, heterogeneous PE capabilities, and path-delay balancing, guide placement.

Routing: CGRA network topologies impose varying routing constraints. Simple fabrics may support neighbor-to-neighbor connections with handshaking while switch-based topologies may require additionally branch-delay matching. Across all cases, routing aims to minimize edge delays and reduce critical paths within each configuration.

2 Partition, Mapping, and Routing

Vectorizable programs compile into an intermediate representation (IR) capturing producer–consumer dependencies and spatial resource requirements. Existing static analyses, such as loop peeling and loop-bound analysis, guide the partitioning of program-order constraints. STEP's partitioning optimizer then applies the following configuration breakpoint rules.

1. **Control-flow constraint (Figure 1(a)):** Control divergence requires a configuration boundary. Branches and jumps are not permitted within a configuration.
2. **Memory-load constraint (Figure 1(b)):** Load-to-use dependencies require dependent nodes to be placed in separate configurations. No memory load–use edges are permitted within a single configuration.
3. **Resource constraint (Figure 1(c)):** CGRA resources capacity must not be exceeded within a configuration.

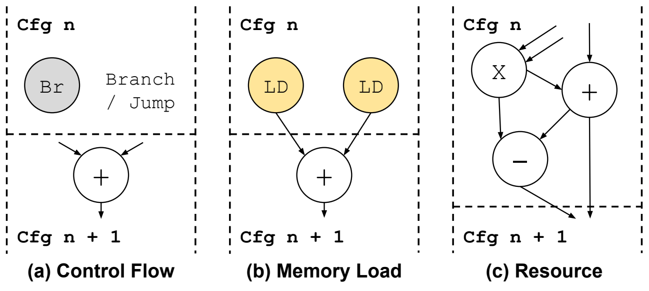

Figure 1. Partitioning Constraints

Beyond these constraints, instructions can be placed in any configuration as long as dependencies and control-flow are preserved. This allows initially nearby instructions to be scheduled at opposite ends of a basic block without violating program order. Control flow is decoupled from the CGRA fabric: rather than embedding jumps in the dataflow, execution from the external program counter controls the predication-based reconfiguration. Figure 2 shows basic blocks partitioned into configurations, with predication guiding execution to Cfg1 or Cfg3.

3 STEP Execution

Figure 3(f) shows steady-state MRRG execution, where logical threads launch every two cycles, achieving overlapped execution at the target initiation interval (II). This multi-context model requires each PE to perform different operations across time, with the II constraining scheduling, resource allocation, and throughput. To preserve program order, all PEs advance in lockstep, relying on single-cycle scratchpad memories to avoid stalls. However, when accessing system memory, unpredictable latency can stall subsequent iterations and disrupt the steady-state pipeline.

In contrast, STEP spatially pipelines the logical threads in program order, eliminating cyclic execution. Figure 3(g) shows the same DFG on STEP: threads stream through the fabric while memory accesses overlap with subsequent threads. Cross-configuration dependencies are buffered in local registers, shifting optimization from II to configuration count. With sufficient logical thread counts, double buffered configurations launch immediately after fabric completion, effectively hiding memory latency.

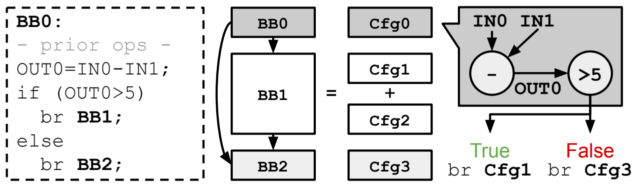

Figure 2. Control-Dataflow Graph Example (BB: Basic Block, Cfg: Configuration)

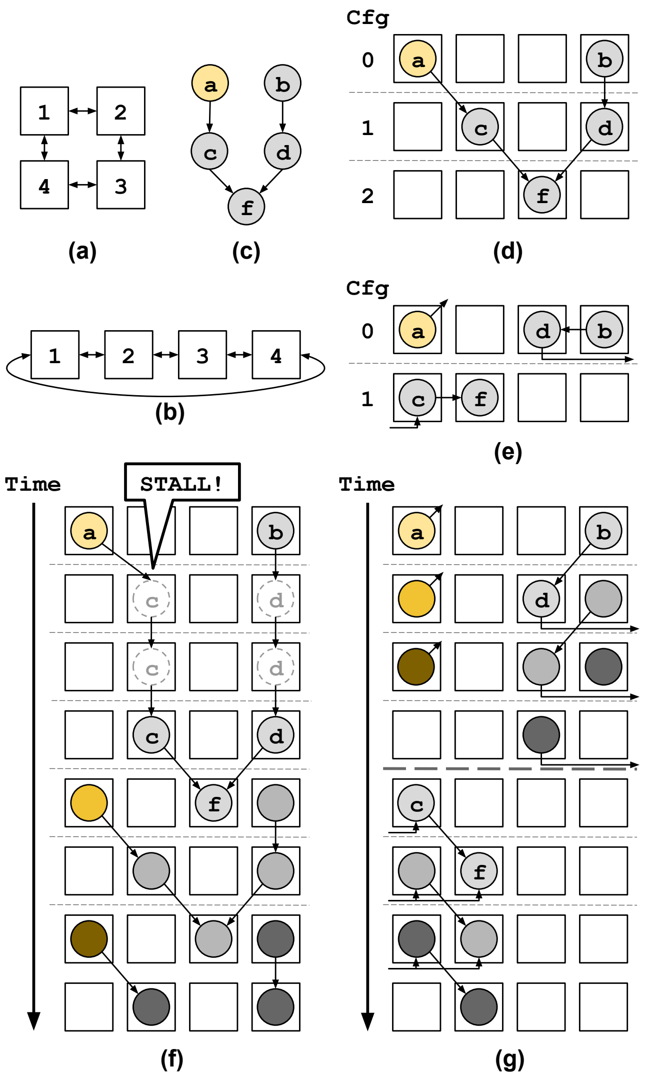

Figure 3. (a) 2×2 CGRA (b) Flattened 2D 2×2 CGRA (c) Input DFG (d) DFG mapped to MRRG based CGRA (II=2) (e) DFG mapped to STEP CGRA (f) MRRG execution of 3 iterations (g) STEP execution with pipelined iterations

References

[1] 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 *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, Seattle, WA, USA, 184–189. doi:10.1109/ASAP.2017.7995277

[2] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. 2023. RipTide: A Programmable, Energy-Minimal Dataflow Compiler and Architecture. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture* (Chicago, Illinois, USA) *(MICRO '22)*. IEEE Press, 546–564. doi:10.1109/MICRO56248.2022.00046

[3] Bingfeng Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. 2003. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *2003 Design, Automation and Test in Europe Conference and Exhibition*. 296–301. doi:10.1109/DATE.2003.1253623

[4] Cheng Tan, Deepak Patil, Antonino Tumeo, Gabriel Weisz, Steve Reinhardt, and Jeff Zhang. 2023. VecPAC: A Vectorizable and Precision-Aware CGRA. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. 1–9. doi:10.1109/ICCAD57390.2023.10323910