SYSTOLIC-SIM: A FRAMEWORK FOR THE RAPID PROTOTYPING OF CUSTOM SYSTOLIC ARCHITECTURES FOR RECURRENCE-BASED COMPUTATION

Akhilesh Balasingam

Stanford University
May 2025

An honors thesis submitted to the department of
Electrical Engineering
in partial fulfillment of the requirements for the undergraduate
honors program

Advisor: Fredrik Kjolstad

Fredrik B. Kjolstad _____ Date: 05/14/2025

Fredrik Kjolstad (Thesis Advisor)
Assistant Professor
Computer Science

_____ Date: _____

John Pauly (Thesis Advisor)
Professor
Electrical Engineering

Date:

Fredrik Kjolstad (Thesis Advisor)
Assistant Professor
Computer Science

Date: May 14, 2025

John Pauly (Thesis Advisor)
Professor
Electrical Engineering

Abstract

Recurrence-based computations—where each iteration depends on values produced in earlier steps—are ubiquitous in domains ranging from numerical linear algebra to machine learning and signal processing. While systolic arrays excel at exploiting fine-grained pipelining and local data movement, mapping general affine recurrences onto efficient systolic arrays remains a major challenge. Systolic-Sim closes that gap by providing an end-to-end framework that automates the translation from high-level recurrence descriptions to cycle-accurate performance evaluation of custom systolic fabrics.

Specifically, our contributions falls into three categories:

1. **A domain-specific intermediate representation (IR):** We design a Protocol-Buffer–based IR that cleanly captures both the structural topology of a systolic array (including mesh, torus, and hierarchical meta-cell layouts) and the per-PE microarchitecture (instruction streams, register files, channel latencies and bandwidths).

2. **A cycle-accurate simulation framework:** Building on the Dataflow Abstract Machine (DAM), we implement a simulator that interprets the IR to model computation and communication at the granularity of individual cycles, faithfully reproducing pipelining, backpressure, and synchronization stalls.

3. **A domain-space exploration engine:** We develop an automated engine that sweeps architectural parameters—array dimensions, communication bandwidths, register capacities—and collects detailed metrics (throughput, utilization, stall rates, per-cycle activity) to guide design trade-offs.

Unlike prior efforts that focus narrowly on dense GEMM or fixed topologies, Systolic-Sim generalizes to a wide spectrum of recurrence-driven workloads and architectural configurations. We validate our framework on canonical kernels—including matrix multiplication (SUMMA and Cannon), triangular solve (TRSM), Cholesky decomposition, FFT with meta-cell hierarchies—and communication-strategy studies. Our results demonstrate how targeted asymmetries (e.g., heterogeneous link provisioning and register allocations) and topology choices can substantially improve pipeline balance

and overall utilization. Systolic-Sim offers a novel foundation for exploring the design space of domain-specialized accelerators; by decoupling high-level recurrence mappings from low-level hardware models, this work accelerates hardware–software co-design and paves the way for rapid iteration from algorithm specification to synthesizable accelerator architectures.

Acknowledgements

I am grateful to my advisor, Professor Fredrik Kjolstad, for his support and guidance on this project over the past eighteen months. I first joined his group working on SAM, a compiler for Onyx—a CGRA-based hardware–software system for dense image processing and sparse tensor algebra—where I learned to bridge compiler abstractions with hardware implementations. That foundation led me to my current work on recurrence-based systolic array prototyping, applying similar dataflow principles to express and evaluate loop-carried dependencies in spatial architectures. Throughout both phases, Professor Kjolstad has provided insightful feedback of the research process. His mentorship has been instrumental in fostering my technical growth and my confidence as a researcher.

I am also deeply grateful to Shiv Sundram for his close collaboration on this project. From the initial problem formulation through every subsequent design decision, Shiv's deep expertise in compilers and dataflow abstractions was instrumental in shaping the IR and DAM integration. He mentored me as I implemented key features of the simulator, troubleshot performance issues, and iterated on the various kernels. Shiv's talent for distilling complex concepts into clear, actionable insights—coupled with his creativity, technical skill, and steadfast support—made this research both productive and enjoyable.

I would also like to thank Nathan Zhang for patiently answering my many questions about the Dataflow Abstract Machine. Nathan's clear explanations of CSPT semantics, channel modeling, and synchronization mechanisms enabled me to implement this simulation framework DAM with confidence.

I also wish to thank Professor John Pauly for serving as the secondary reader on this thesis and for the guidance he has offered throughout my undergraduate studies. Professor Pauly's thoughtful, broader career direction has been a constant source of inspiration.

I would also like to thank Professor Priyanka Raina for giving me the opportunity to do research in electrical engineering as early as my freshman year, and Professor Doug James for giving me research guidance as I explored computer graphics and physics-based animation in his lab. These research experiences helped me hone my technical skills and further refine my research interests.

I would like to thank my brother Arjun and my sister Ramya for being constant sources of

support, love, and entertainment over the years. Even from across the country, their encouragement and good humor have meant the world to me, and I've always looked forward to our calls.

Finally, but most importantly, I am forever grateful to my parents, Pratheep and Esha Balasingam, for their unwavering faith in me and for inspiring me to work hard and stay true to my goals. They have sacrificed so much for my education, and without their love, patience, and encouragement, none of this would have been possible. I dedicate this thesis to them.

Contents

Abstract **iii**

Acknowledgements **v**

1 Introduction **1**

1.1 Recurrences: Relevance in Modern Computation 2

1.2 Contributions 5

2 Background **7**

2.1 Prior Work on Custom Systolic Array Generation 7

2.1.1 Foundational Contributions 7

2.1.2 Advancements in Automated Synthesis and Application-Specific Designs . . . 8

2.1.3 Gaps in Existing Research 10

2.2 Novel Contributions of This Work 11

2.2.1 General-Purpose Simulator for Recurrence-Based Computations 11

2.2.2 Extensible Architecture Modeling and Performance Metrics 12

2.2.3 Topology-Aware Execution Semantics 13

2.2.4 Enabling Rapid Prototyping and Hardware-Software Co-Design 13

2.3 The Dataflow Abstract Machine (DAM) 13

2.3.1 Interface and Execution Model 14

2.4 Recurrence to Per-PE Programs 15

2.4.1 Matrix Multiplication (GEMM) 16

2.4.2 Triangular Solve (TRSM) 17

2.4.3 Systolic Constraints and Mapping Properties 18

2.5 Hardware Considerations for Systolic Arrays 18

2.5.1 Processing Element (PE) Microarchitecture 18

2.5.2 Interconnect and Communication Patterns 19

2.5.3 Buffering and Storage 19

2.5.4 Resource Utilization and Scalability 19

3 Overview of Systolic-Sim **21**

4 Intermediate Representation (IR) **23**

4.1 Blocks: Representing Structured Data . . . 23
4.2 Operations: Defining Computation and Communication . . . 24
4.3 Cells: Modeling Processing Elements . . . 25
4.4 Topology and Simulation Structure . . . 27
4.5 Choice of Representation: Protocol Buffers . . . 27
4.6 The IR in our End-to-End Framework . . . 28

5 Implementation: Lowering IR to DAM **29**

5.1 Configuration Loading and Preprocessing . . . 30
5.2 Channel Topology Construction . . . 30
5.3 Runtime Context Construction . . . 32
5.4 Execution Model and Cell Behavior . . . 33
5.5 Architectural Features Captured in the Simulator . . . 33
5.5.1 Complex Operations and Computational Modeling . . . 34
5.5.2 Worked Example: Execution of a GEMM Cell . . . 35
5.6 Architectural Fidelity and Pipeline Modeling . . . 36
5.7 Support for Complex Topologies . . . 36
5.7.1 Modeling Sentinel Cells . . . 37
5.7.2 Broadcast Simulation and Aggregated Communication . . . 38
5.7.3 Advanced Communication Behaviors . . . 39
5.7.4 Fine-Grained Performance Metrics . . . 40
5.8 Summary and Outlook . . . 41

6 Evaluation **42**

6.1 Evaluation of Cannon's GEMM . . . 42
6.2 Evaluation of Triangular Solve (TRSM) . . . 43
6.3 Evaluation of Fast Fourier Transform (FFT) . . . 46
6.3.1 Problem Formulation . . . 46
6.3.2 Experimental Setup . . . 47
6.3.3 Utilization Results and Analysis . . . 48
6.4 Evaluation of Cholesky Decomposition . . . 49
6.4.1 Problem Formulation and Experimental Setup . . . 49
6.4.2 Utilization and Throughput Results and Analysis . . . 50
6.5 Evaluation of Communication Strategies, GEMM . . . 51
6.6 Summary of Evaluation . . . 52

7 Conclusions **53**

Bibliography **55**

Chapter 1

Introduction

Over the past few decades, the increasing demand for high-performance computing from domains ranging from scientific simulations to machine learning has resulted in a resurgence of interest in special-purpose hardware. Systolic arrays, first proposed by Kung and Leiserson in the late 1970s, are a prominent example of such architectures. These spatially distributed, rhythmically pipelined processing elements have been used to great effect in dense linear algebra operations, particularly matrix multiplication. Their regular, highly parallel structure and ability to exploit data locality make them an attractive target for domain-specific accelerators. Google's TPU, for instance, employs a large systolic array as the backbone for high-throughput matrix operations in deep learning workloads.

However, the utility of systolic arrays extends far beyond matrix multiplication. Triangular solve with multiple right-hand sides (TRSM), Cholesky decomposition, and even certain stencil and recurrence-based computations can all be expressed as affine loop nests and thus lend themselves to systolic execution. Mapping these more irregular or dependent computations to systolic arrays poses significant challenges. Systolic-Sim is part of a larger project that explores this frontier: how to map recurrence relations—computations with loop-carried dependencies—onto custom systolic arrays. Specifically, this thesis will focus on automatically generating simulations of systolic arrays of various dimensions and topologies given some high-level description of their executing programs. We leverage the dataflow abstract machine (DAM), a cycle-accurate hardware simulator that models the behavior of systolic arrays, to perform rapid prototyping of various scheduling and communication

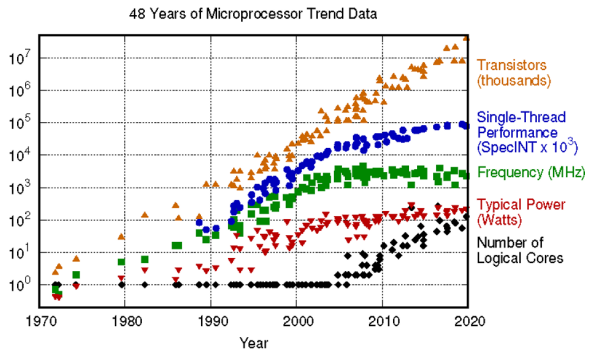


Figure 1.1: Decline of Moore's Law and Slowing of Dennard Scaling

strategies and to conduct a wide domain-space exploration of custom hardware.

1.1 Recurrences: Relevance in Modern Computation

Recurrences appear pervasively across scientific computing, numerical linear algebra, and machine learning. Algorithms like forward and backward substitution, Kalman filtering, dynamic programming, and iterative solvers often include loop-carried dependencies that limit parallelism. Naively parallelizing such algorithms can result in either inefficient synchronization or incorrect results. However, with careful scheduling and communication, it is possible to restructure these algorithms into a systolic form, where dependencies are spatially shifted across a mesh of processors, allowing pipelined execution.

The importance of this work stems from the intersection of several converging trends in hardware design, compiler technology, and compute demand. The motivation for mapping recurrence relations to systolic arrays in hardware—particularly using cycle-accurate simulators that reflect realistic system constraints—responds to pressing technological and algorithmic needs. There are three key considerations that make this research timely and impactful.

- **Evolving Workloads:** Contemporary compute workloads are increasingly dominated by structured, data-parallel operations that exhibit regular patterns of dependence. In domains such as machine learning, scientific simulation, computer graphics, and signal processing, many of the underlying algorithms are naturally expressed using recurrence relations—mathematical

formulations where each output depends on a fixed, local set of previous computations. These patterns lend themselves to spatial execution models, where compute elements are laid out in a physical array and communicate directly with neighbors. Traditional CPUs and GPUs, while powerful, are not optimized for this style of computation; their performance is often hindered by instruction dispatch overhead, cache management complexity, and inefficient control logic. As the scale and dimensionality of these workloads grow, particularly in large neural network inference and multi-dimensional stencil computations, the ability to directly map recurrence structures onto hardware becomes a critical advantage. Systolic arrays provide an ideal substrate for this mapping, enabling fine-grained pipelining and deterministic data movement, which together improve both throughput and energy efficiency.

- **Architectural Constraints:** As shown in Figure 1.1, the decline of Moore's Law and the breakdown of Dennard scaling have placed hard limits on the performance gains available from traditional process scaling. As a result, general-purpose processor architectures face increasing challenges in delivering energy-efficient performance improvements year over year. CPUs struggle with parallelism due to control-heavy pipelines and speculative execution, while GPUs, despite their massive throughput, incur energy and latency penalties from global memory accesses and irregular communication patterns. In contrast, spatial architectures such as systolic arrays avoid many of these pitfalls by employing static dataflow execution: data moves between neighboring compute units in a regular pattern, and operations are orchestrated in a time-synchronized, pipeline-friendly manner. These advantages are visualized in Figure 1.2. This not only minimizes energy per operation but also enables predictable performance, which is crucial in real-time and embedded applications. Moreover, the fixed-function structure of systolic arrays—where control logic is largely amortized—allows for higher compute density per watt, a key metric in the era of energy-constrained computing. Additionally, for recurrences—which are the domain of this research project—systolic arrays offer a natural rhythmic consistency that naturally parallels the regularity in recurrence-based computation.

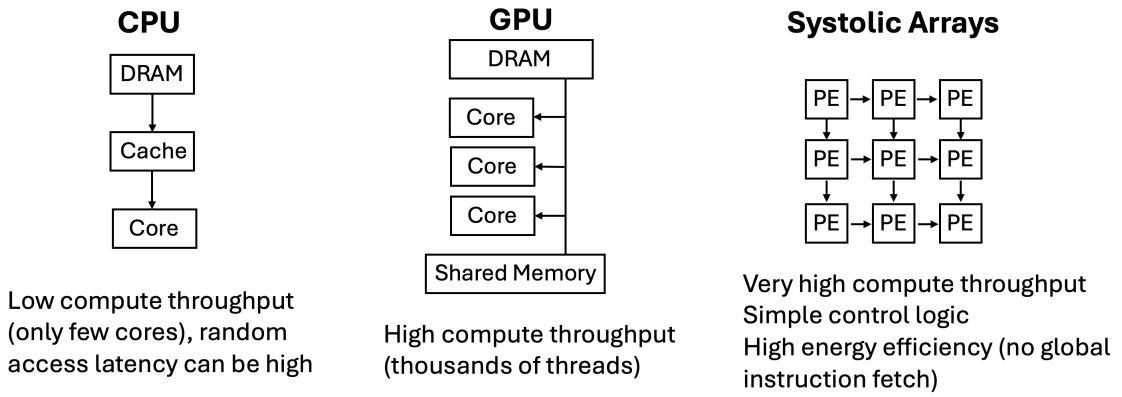


Figure 1.2: Spatial architectures have some advantages over von Neumann architectures

- **Toolchain Maturity:** Historically, the design of custom hardware accelerators required significant manual effort, often involving low-level RTL development and labor-intensive validation processes. However, recent advances in compiler infrastructure, hardware modeling languages, and high-level synthesis (HLS) tools have dramatically reduced this barrier. Frameworks like LLVM, CIRCT, and MLIR, along with domain-specific tools such as AutoSA and Halide, allow hardware designers and software engineers to collaboratively explore architectural ideas at a high level of abstraction. These developments enable rapid iteration across the hardware-software boundary, making it possible to co-design algorithmic representations and their corresponding architectural mappings. Additionally, cycle-accurate simulators, like the one described in this work, allow researchers to reason quantitatively about performance tradeoffs, communication bottlenecks, and utilization patterns without first fabricating silicon. This newfound agility in prototyping and evaluation is particularly important as we move toward more heterogeneous and application-specific compute landscapes, where tailoring hardware to the algorithm is no longer optional but necessary.

The central goal of this project was to develop a framework for the rapid prototyping of custom systolic architectures. As I discuss in Section 2, a compiler generated as part of the larger project outputs a program of operations for each processing element in the iteration space. Systolic-Sim takes this high-level systolic array description as input and generates a hardware prototype for the

systolic array. Specifically, I extend DAM to support regular, torus-like, and meta-cell topologies. I also extend DAM to capture realistic constraints such as communication latency, synchronization, and buffer limitations. This enables a rigorous study of the design tradeoffs inherent in implementing recurrence algorithms as spatially distributed hardware processes.

1.2 Contributions

This thesis contributes to the growing body of work on hardware-software codesign for spatial architectures by focusing on the systolic realization of recurrence computations. Specifically, it introduces tools and abstractions for modeling and evaluating such mappings on hardware using a dataflow abstract machine (DAM). The main contributions of this work are:

- **A domain-specific intermediate representation (IR) for systolic array architectures.** I introduce a new IR designed to express both the structural organization of systolic arrays and the instruction set architectures (ISAs) of individual processing elements (PEs). This IR captures spatial connectivity, temporal scheduling behavior, and low-level compute operations, serving as a bridge between high-level compiler output and cycle-accurate simulation.

- **A simulation framework for lowering this IR into a cycle-accurate simulation.** I develop a runtime infrastructure that combines architectural specifications with recurrence-derived programs emitted by a compiler (created separately, but in tandem with the project discussed in this thesis; to be detailed in future work). This framework supports execution on a dataflow abstract machine (DAM), enabling per-cycle simulation of systolic execution, including communication, buffering, and PE compute semantics.

- **A domain-space exploration engine with cycle-accurate performance analysis.** Building on the DAM simulation backend, I implement a domain-space exploration engine that evaluates how design choices—such as tile sizes, communication patterns, and PE microarchitectures—affect performance for given recurrence mappings. The engine supports automated

sweeping and evaluation across design points, using cycle-accurate metrics including throughput, stall rates, and utilization. Unlike state-of-the-art systolic array generators and simulators, which focus narrowly on regular grid-like topologies for matrix multiplication (GEMM), this tool can adapt to a wide variety of compute workloads and architectural topologies. I will highlight the superiority of our simulation framework through case studies, including of matrix multiplication (GEMM)—SUMMA and Cannon's variants—triangular solve (TRSM), Cholesky decomposition, and divide-and-conquer algorithms like fast Fourier transforms (FFT).

In the next chapter, I will provide background on relevant research and the dataflow abstract machine (DAM). Then, I will discuss the IR I devised to represent systolic array architectures, as well as the simulation framework for integrating architectural descriptions with the compiler-generated recurrence mappings. I will then show a domain-space exploration engine that uses this framework to investigate the impact of various design decisions on some example recurrences.

Chapter 2

Background

The evolution of systolic arrays has been marked by significant milestones, from their conceptual inception to sophisticated automated design frameworks. In this chapter, I will discuss prior work in this area, the Dataflow Abstract Machine (DAM), a simulation language in which I've implemented Systolic-Sim, and the relevant hardware considerations for systolic arrays. I will also outline the novel contributions of Systolic-Sim in this chapter.

2.1 Prior Work on Custom Systolic Array Generation

In the following sections, I review foundational work that established the theoretical underpinnings of systolic array architectures and their use in regular computations. I then examine more recent research on the automated synthesis of systolic arrays, highlighting both the progress made and the limitations that motivate the development of Systolic-Sim.

2.1.1 Foundational Contributions

Kung and Leiserson (1979)

H. T. Kung and Charles E. Leiserson introduced the concept of systolic arrays in their seminal work on VLSI processor arrays (Kung & Leiserson, 1979). They proposed a network of simple processing elements (PEs) that rhythmically compute and pass data through the system, akin to the rhythmic

contraction of the heart, hence the term "systolic." This architecture emphasized local interconnects and regular data flow, enabling high-throughput and efficient hardware implementations for regular computations such as matrix multiplication and convolution.

While groundbreaking, their work primarily focused on the theoretical underpinnings and hardware design aspects. It lacked a systematic methodology for mapping high-level algorithmic descriptions, particularly those involving complex recurrence relations, onto systolic architectures.

2.1.2 Advancements in Automated Synthesis and Application-Specific Designs

The evolution of systolic array design has been influenced significantly by the development of automated synthesis tools and application-specific frameworks. These advancements have aimed to streamline the design process, improve performance, and cater to specific computational workloads. Notable among these are PolySA, AutoSA, and Odyssey, each contributing uniquely to the field.

PolySA: Polyhedral-Based Systolic Array Auto-Compilation

PolySA, introduced by Cong and Wang in 2018 (Cong & Wang, 2018), represents a pioneering effort in automating the synthesis of systolic arrays using the polyhedral model. This framework leverages the mathematical rigor of the polyhedral approach to transform high-level loop nests into efficient systolic array implementations on FPGAs. By systematically applying space-time transformations, loop tiling, and other optimizations, PolySA can generate designs for applications like matrix multiplication and convolutional neural networks.

One of PolySA's significant contributions is its ability to reduce the development cycle from several months to within an hour, producing designs with performance comparable to manually optimized counterparts. However, PolySA primarily focuses on applications with regular computation patterns and may not generalize well to workloads with irregular dependencies or control flows. Additionally, while it automates the compilation process, it offers limited support for exploring a wide range of architectural configurations or performing detailed performance trade-off analyses.

AutoSA: End-to-End Compilation Framework for Systolic Arrays

Building upon the foundations laid by PolySA (Cong & Wang, 2018), AutoSA (Wang, Guo, & Cong, 2021) extends the capabilities of automated systolic array synthesis. AutoSA provides an end-to-end compilation framework that not only automates the transformation of high-level code into systolic array architectures but also incorporates comprehensive design space exploration. Utilizing the polyhedral model, AutoSA can handle applications with complex dependence structures, enabling the generation of high-performance designs for a broader class of computations.

AutoSA introduces several enhancements, including support for various dataflows (e.g., output-stationary, weight-stationary), latency hiding techniques, and SIMD vectorization. These features allow for more efficient utilization of hardware resources and improved performance. Nevertheless, AutoSA's design space exploration is primarily guided by heuristics and may not exhaustively cover all possible configurations. Moreover, while it provides performance estimates, it lacks integration with cycle-accurate simulation tools, limiting its ability to model dynamic behaviors such as pipeline stalls or communication bottlenecks accurately.

GEMMINI: Parametrizable Systolic Array Generator for Matrix Multiplication

GEMMINI (Genc et al., 2019) introduces a flexible and open-source framework for generating systolic arrays tailored to dense matrix multiplication. Developed as part of the RISC-V ecosystem, GEMMINI offers a parameterizable hardware generator built on the Chisel hardware construction language, allowing users to configure array dimensions, dataflow strategies, and memory hierarchies. One of GEMMINI's key contributions is its integration of a tiled matrix multiplication kernel with an optional on-chip accumulation buffer, enabling high throughput for common DNN workloads. The framework also includes a software stack that supports kernel offloading, memory management, and microkernel scheduling, making it a practical platform for real-world deployment on embedded accelerators.

However, GEMMINI's scope is intentionally focused on dense matrix-matrix multiplication (GEMM) and lacks native support for broader classes of recurrence-based computations, such as triangular

solves or Cholesky decomposition. Furthermore, it does not include built-in mechanisms for evaluating alternative topologies or simulating execution with fine-grained performance metrics. As a result, while GEMMINI is highly effective for targeted DNN acceleration, it is less suitable for general-purpose systolic architecture exploration or hardware-software co-design workflows involving irregular computation patterns.

Odyssey: Comprehensive Design Space Exploration for Systolic Arrays

Odyssey (Wang & Cong, 2021) addresses the need for a more thorough exploration of the systolic array design space. Recognizing the limitations of heuristic-based approaches, Odyssey employs a hybrid optimization strategy combining mathematical programming and evolutionary algorithms to search for optimal configurations. This methodology enables the identification of high-performance designs that might be overlooked by traditional techniques.

A key strength of Odyssey is its comprehensive performance modeling, which accounts for factors like prologue and epilogue latencies, resource utilization, and communication overheads. By integrating these considerations, Odyssey can provide more accurate predictions of design performance. However, similar to its predecessors, Odyssey does not incorporate cycle-accurate simulation, which is crucial for capturing the temporal dynamics of systolic array execution. Additionally, while it excels in design space exploration, it may require significant computational resources for large-scale problems, potentially limiting its practicality in certain scenarios.

2.1.3 Gaps in Existing Research

While PolySA, AutoSA, GEMMINI, and Odyssey have significantly advanced the field of automated systolic array design, several gaps remain. Notably, the absence of cycle-accurate simulation in these frameworks limits their ability to model and analyze temporal behaviors, such as pipeline utilization and communication delays, which are critical for optimizing performance. Furthermore, their focus on specific application domains and reliance on heuristic or computationally intensive exploration methods may restrict their adaptability and scalability. The limitations of prior research in this area are highlighted in Table 2.1.

Table 2.1: Summary of Related Work on Systolic Array Generation

Paper	**Contribution**	**Limitation**
Kung & Leiserson (1979)	Introduced systolic arrays for VLSI implementations of regular computations	Focused on theoretical formulation
Cong & Wang (2018): PolySA	Automatic systolic array generation for GEMM/CONV	Narrow workload focus, FPGA-based
Genc et al. (2019): GEMMINI	Parametrizable systolic array generator for GEMM	Full RTL generation; only focused on GEMM
Cong et al. (2020): AutoSA	Loop-transformation driven end-to-end compilation	Narrow workload focus
Wang & Cong (2021): Odyssey	Design-space exploration for optimal configurations	No execution modeling or runtime behavior feedback

These observations highlight the need for a unified framework that combines automated synthesis with cycle-accurate simulation and flexible design space exploration. Such a framework would enable more accurate performance modeling, support a wider range of applications, and facilitate rapid prototyping and evaluation of custom systolic architectures. Addressing these needs forms the basis of our proposed approach, which aims to bridge the gap between high-level algorithm specifications and low-level hardware implementations through an extensible and efficient simulation infrastructure.

2.2 Novel Contributions of This Work

This work presents a general-purpose simulation framework for systolic arrays grounded in the formalism of recurrence equations. It is designed to support a broad class of dataflow computations, provide realistic cycle-accurate performance modeling, and enable architecture exploration at fine granularity. The key contributions of this work are outlined below.

2.2.1 General-Purpose Simulator for Recurrence-Based Computations

At the heart of this work is a simulator that generalizes the execution of systolic arrays beyond fixed-purpose designs. Unlike prior frameworks that are tightly coupled to specific algorithms (e.g., matrix multiplication) or fixed microarchitectures, our simulator accepts high-level recurrence relations and

lowers them to a dataflow abstract machine (DAM). This abstraction enables broad applicability across domains such as signal processing, scientific computing, and deep learning, where many algorithms can be expressed as recurrence relations.

This general-purpose design empowers researchers and hardware architects to model novel compute patterns without reengineering the backend infrastructure. Recurrences define not only the computation but also the data dependencies, which are translated into spatial execution schedules and communication topologies automatically.

The simulator supports a rich and extensible set of operations that captures both elementary computations (e.g., `ADD`, `MUL`, `SEND`, `RECV`) and more complex ones (e.g., `GEMM`, `FFT`, `SYRK`, `CHOL`, `TRISOLVE`). This reflects a realistic approximation of operations supported by contemporary accelerators.

Each instruction is modeled with cycle-level granularity, taking into account factors like operand size, pipeline latency, and data readiness. This allows for realistic performance estimation and behavioral analysis of systolic array designs under varied workloads.

By decoupling computation from physical hardware assumptions, the ISA serves as a versatile layer where high-level compiler outputs and low-level hardware models can meet. Moreover, the modularity of the instruction definitions allows easy integration of new operations, making the framework future-proof and adaptable.

2.2.2 Extensible Architecture Modeling and Performance Metrics

The simulation framework enables fine-grained specification of hardware characteristics via a Protobuf-based intermediate representation (IR). Users can configure topologies (e.g., mesh, torus, broadcast), interconnect constraints (e.g., channel bandwidth), and cell-level behavior (e.g., pipelining, buffer sizes). Sentinel cells and generator contexts allow modeling of hierarchical and source-driven behaviors, respectively.

The simulator collects detailed per-cycle performance metrics at the level of individual processing elements (PEs). This instrumentation supports precise profiling and diagnostic visualization, such as heatmaps of utilization and bottleneck analysis, which are critical for co-design and optimization workflows.

2.2.3 Topology-Aware Execution Semantics

One of the most distinctive features of this simulator is its ability to capture and simulate the effect of different spatial communication topologies. Using declarative configuration files, users can instantiate complex interconnects—including wrap-around (toroidal) links, broadcast buses, and multi-hop routing schemes—without modifying core simulation logic.

This level of flexibility allows the study of how data movement strategies impact performance. For example, users can explore the latency trade-offs between nearest-neighbor communication versus broadcast patterns, or compare pipeline behavior under row-major versus column-major tile ordering. The dynamic channel remapping logic ensures that even algorithms requiring skewed communication (such as Cannon's matrix multiplication algorithm) can be easily represented.

2.2.4 Enabling Rapid Prototyping and Hardware-Software Co-Design

The simulator is architected to serve as a rapid prototyping platform for hardware-software co-design. Compiler developers can generate instruction streams based on recurrence transformations, while hardware architects can test how those streams perform under varying architectural assumptions.

This workflow encourages an iterative design methodology: high-level programs are compiled into operation sequences; those sequences are mapped to architectural models; simulations are run to evaluate performance and extract metrics; insights guide both software optimizations and hardware revisions.

This end-to-end flow directly supports research into domain-specific hardware acceleration, compiler-driven mapping strategies, and architectural parameter sweeps (e.g., varying pipeline depths or register file sizes).

2.3 The Dataflow Abstract Machine (DAM)

As spatial and streaming architectures continue to rise in relevance, the need for high-performance and flexible simulation tools has become increasingly important. The Dataflow Abstract Machine (DAM) is a parallel, event-queue-free simulator designed specifically to support the development

and analysis of dataflow systems at scale. In this project, DAM plays a central role in enabling the cycle-accurate simulation of systolic array hardware as a target for recurrence-based computations.

This project focuses on mapping affine recurrence equations to systolic arrays composed of general-purpose processing elements. These mappings are generated by an external compiler and integrated into a simulation environment via a custom architectural IR. DAM provides the underlying runtime and interface for this environment, allowing hardware-software co-design techniques to be applied to real workloads without committing to RTL or physical hardware.

DAM's core abstraction is built around Communicating Sequential Processes with Time (CSPT). This model exposes a highly expressive yet lightweight interface for defining the behavior of components in a dataflow system. Each unit, or context, in DAM executes independently, with its own local notion of simulated time. This asynchrony, enabled by distributed timekeeping and slack-aware scheduling, forms the foundation for DAM's scalability.

2.3.1 Interface and Execution Model

DAM provides three main constructs to users: contexts, time, and channels. Contexts encapsulate computational elements, channels represent point-to-point communication between contexts, and local time enables cycle-accurate simulation. Communication over channels follows an explicitly timestamped protocol, capturing not only the values exchanged but also the timing of their production and consumption.

The CSPT model enables intuitive descriptions of system behavior while preserving simulation fidelity. Time advancement is explicit and local: contexts can advance their own clocks as needed, but cannot regress. This design contrasts sharply with traditional event-driven simulators, which require global event queues and synchronization barriers.

To support efficient parallel simulation, DAM implements two key synchronization mechanisms:

- **Synchronization via Atomics (SVA)** uses acquire/release memory semantics to ensure consistency across contexts without expensive global coordination. This mechanism allows contexts to read each other's state safely, provided that simulated times align.

- **Synchronization via Parking (SVP)** uses futex-based thread parking to block contexts

only when strictly necessary—i.e., when a context needs to wait for another to progress in simulated time. SVP provides a fallback for cases where SVA is insufficient, such as in strongly time-dependent coordination.

A critical feature of DAM is its support for **time-bridging channels**. These channels decouple the simulated time of the sender and receiver, allowing for realistic modeling of communication delays, backpressure, and buffer overflows. Channels are bounded or unbounded, and each transmission carries a timestamp that determines its visibility on the receiving end.

When the receiver dequeues a message, it acknowledges the action by sending a response (with a simulated timestamp) back to the sender. This response mechanism lets the sender update its view of channel occupancy and time, enabling accurate modeling of flow control. This is essential for systems like systolic arrays, where tight coordination and pipeline balance are necessary for peak performance.

By using DAM, I am able to simulate realistic architectural constraints such as bounded channel depth, multi-cycle compute latencies, and synchronization-induced stalls. This allows for domain-space exploration: sweeping across design parameters like tile sizes and buffer capacities, and analyzing their impact on throughput, latency, and hardware utilization. The system enables the study of not only functional correctness but also performance bottlenecks, such as pipeline bubbles and communication mismatches, which require longer synthesis-level simulations in typical hardware simulators.

2.4 Recurrence to Per-PE Programs

A central goal of this work is to generate systolic arrays from affine recurrence equations that respects data dependencies and exposes pipeline parallelism. In this section, we walk through two canonical examples—matrix multiplication (GEMM) and triangular solve (TRSM)—to illustrate how space-time mappings can be derived from loop nests and dependence relations, and how they inform systolic execution.

2.4.1 Matrix Multiplication (GEMM)

Consider the standard three-nested-loop implementation of matrix multiplication:

$$C[i,j] \;=\; \sum_{k=0}^{K-1} A[i,k] \;\cdot\; B[k,j].$$

The canonical three-loop nest is given by:

Original Loop Nest

```
for i = 0 to M-1:
  for j = 0 to N-1:
    C[i][j] = 0
    for k = 0 to K-1:
      C[i][j] += A[i][k] * B[k][j]
```

This is then mapped to a per-PE program, given by:

Per-PE Program

```
acc = 0
for k = 0 to K-1:
  wait for A[i][k] and B[k][j]
  acc += A[i][k] * B[k][j]
end
C[i][j] = acc
```

This PE program corresponds directly to the recurrence above, with each element of A and B streamed in at time step k and accumulated into a local register `acc`. We will focus in this thesis on the cycle-accurate execution of such PE programs under various architectural configurations.

2.4.2 Triangular Solve (TRSM)

Consider solving $LX = B$ where L is a lower triangular matrix. The recurrence for TRSM is:

$$X[i,j] = \left(B[i,j] - \sum_{k=0}^{i-1} L[i,k] \cdot X[k,j] \right) / L[i,i] \tag{2.1}$$

This recurrence depends on $X[k,j]$ for $k < i$, implying a triangular dependency pattern.

As in the GEMM example, we begin with the original loop nest:

```
for i = 0 to M-1:
  for j = 0 to N-1:
    X[i][j] = B[i][j]
    for k = 0 to i-1:
      X[i][j] -= L[i][k] * X[k][j]
    X[i][j] /= L[i][i]
```

Now, we can generate the per-PE programs, as follows:

```
for k = 0 to i-1
  wait for L[i][k] and X[k][j]
  acc -= L[i][k] * X[k][j]


acc += B[i][j]
X[i][j] = acc / L[i][i]
```

Unlike GEMM, where the recurrence is purely feed-forward and spatially parallel across all (i,j) pairs, TRSM introduces true data dependencies along the i dimension due to the sequential nature of triangular solve. Each output $X[i][j]$ depends on previously computed values $X[k][j]$ for $k < i$, which constrains the mapping to a wavefront schedule. The automatic mapping of recurrences to per-PE programs will be discussed in a forthcoming publication.

2.4.3 Systolic Constraints and Mapping Properties

Both examples illustrate properties common to valid systolic schedules:

- **Legality:** All dependencies are satisfied under the space-time mapping; producers execute before consumers.
- **Local communication:** Each PE only communicates with its neighbors, avoiding centralized coordination.

These patterns generalize to more complex recurrence relations and form the foundation for the compiler and Systolic-Sim. In particular, the space-time mappings are generated automatically by a compiler designed to work in tandem with the hardware-focused simulation infrastructure presented here. While this thesis focuses on the architectural modeling and evaluation aspects, the compiler itself will be described in greater detail in a forthcoming publication.

2.5 Hardware Considerations for Systolic Arrays

While mapping from recurrence to PE program provides a software-level abstraction for scheduling recurrence computations, effective realization on hardware demands careful attention to physical constraints. In this section, we examine key considerations that impact the performance, area, and feasibility of implementing systolic arrays in hardware.

2.5.1 Processing Element (PE) Microarchitecture

Each processing element (PE) in a systolic array is a simple compute unit with a fixed datapath and local register storage, managed by a finite state machine for control. The microarchitecture must support the specific class of operations required by the computation, such as multiply-accumulate or fused multiply-add, and include pipeline registers to maintain high throughput across clock cycles. To ensure correct and efficient communication, the PE must buffer both incoming and outgoing data, aligning its internal execution with the availability of operands and the readiness of downstream consumers. Most importantly, each PE implements per-cycle update rules that directly

encode the semantics of the recurrence relation it is assigned to evaluate, ensuring the overall computation unfolds correctly across space and time. All these elements are captured in our simulator environment, as I will discuss in the following chapters.

2.5.2 Interconnect and Communication Patterns

Systolic arrays achieve performance through localized, regular communication. The array's interconnect topology is fixed (e.g., mesh, ring, torus), and:

- Should match the communication pattern implied by the per-PE program.
- Has a particular latency and bandwidth associated with it that impacts message frequency.

For example, matrix multiplication requires broadcast of operands along rows and columns; TRSM requires diagonal propagation of partial results (implemented through sequential down-and-right data movement).

2.5.3 Buffering and Storage

PE-local storage is crucial for: (1) holding partial results or state across time steps, (2) decoupling producers and consumers to avoid stalling, (3) accommodating variable delays due to input arrival skew. Choosing appropriate buffer depths involves a trade-off between area overhead and tolerance to communication imbalance.

2.5.4 Resource Utilization and Scalability

The array must scale with workload size while fitting within the available chip area and power envelope. Key design considerations include the number of registers available within each PE, which constrains the amount of state or intermediate data that can be stored locally without resorting to slower global memory accesses. Additionally, the number of interconnects that can be practically routed to each PE affects the array's communication topology and limits how many concurrent data streams can be supported. To address these physical and architectural limits, designers often group PEs into larger meta-cells—regular subarrays with shared control or communication fabric—that can

be replicated and tiled across the chip. This meta-cell organization enables better locality, simplifies routing, and supports modular scaling while maintaining efficient data reuse and throughput.

Chapter 3

Overview of Systolic-Sim

A high-level overview of our project is provided in Figure 3.1. Systolic-Sim focuses on bridging the gap between compiler-generated recurrence mappings and hardware-aware simulation of systolic arrays. The compiler that generates recurrence mappings will be discussed in a forthcoming publication.

The system takes as input two components:

- A **high-level program of operations** per processing element (PE) in the systolic array. This will be generated by our compiler.
- An **architectural description** of the systolic array, specified via a custom intermediate representation (IR) based on protocol buffers. This IR describes the topology of the array, the capabilities of each PE, and the communication fabric.

Given these inputs, the system automatically generates a full **DAM (Dataflow Abstract Machine)** program that models the execution of the computation on the specified hardware configuration. Specifically, the framework does the following:

- Generates per-PE programs, analogous to instruction streams or microcode, that implement the scheduled recurrence updates based on the space-time mapping.
- Instantiates a simulated systolic array in DAM, connecting PEs according to the provided architectural description.

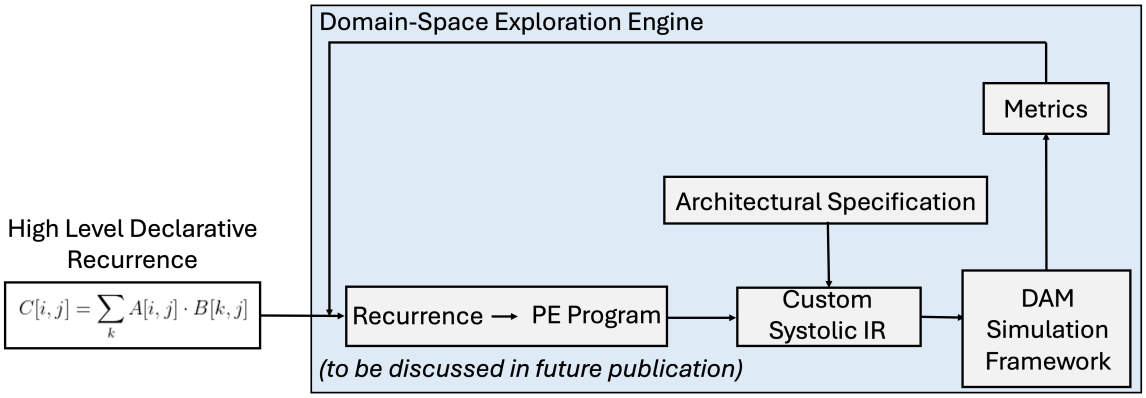


Figure 3.1: Systolic-Sim

- Orchestrates the simulation to reflect realistic timing behavior, including communication delays, backpressure, and compute latencies.
- Collects cycle-accurate performance metrics, enabling analysis of throughput, utilization, and bottlenecks.

This approach enables rapid evaluation of different mappings, architectural choices, and design trade-offs, without requiring RTL-level implementation. The system thus provides a flexible and scalable platform for hardware-software co-design of systolic arrays. The following chapters will discuss the details of the IR, the process of lowering this IR to DAM, and a comprehensive evaluation of Systolic-Sim on various compute kernels, topologies, and architectural configurations.

Chapter 4

Intermediate Representation (IR)

To bridge the gap between compiler-generated space-time mappings and realistic simulation of systolic arrays, this project defines a hardware-focused intermediate representation (IR) using protocol buffers (protobuf). This IR captures the computational behavior, communication structure, and architectural layout of systolic arrays in a machine-readable form. By providing an explicit, flexible description of array architectures, the IR enables the automatic generation of Dataflow Abstract Machine (DAM) programs without requiring manual modeling for each new configuration.

In this section, we describe the design of the IR, organized around its key components: data storage, operations, cell behavior, array topology, and simulation-level structure. Selected excerpts from the protobuf schema are embedded for clarity.

4.1 Blocks: Representing Structured Data

At the foundation of the IR is the `Block` message, which describes structured two-dimensional data. Each block stores a flattened array of double-precision values in row-major order, along with metadata recording the number of rows, columns, and an optional name.

```
message Block {
    repeated double values; // Flat array of values
    uint32 rows;
    uint32 cols;
    string name;
}
```

Figure 4.1: Protocol buffer definition for a `Block`, representing 2D data arrays.

Blocks are used to represent operands for computations, initial local states of processing elements (PEs), and intermediate values passed between cells. Explicit storage of dimensions enables the simulator to reason about matrix and vector sizes during runtime.

4.2 Operations: Defining Computation and Communication

Each PE executes a sequence of `Operation` messages that define its behavior. The `Operation` schema supports a range of computational and communication primitives.

```
message Operation {
   OpType op_type;
   uint32 trip_count;
   string channel_id;
   string register_id;
   string a; string b; string c; string d;
   float value;
   Block data;
   repeated Operation nested_operations;
   float cycle_count;
}
```

Figure 4.2: Protocol buffer definition for `Operation`, encoding computational and communication primitives.

Operations specify either basic computations (such as addition, multiplication, triangular solve) or communication actions (such as sending or receiving over a channel). Loops are handled hierarchically by allowing nested sequences of operations, providing a compact way to describe complex repeated behavior.

4.3 Cells: Modeling Processing Elements

Each processing element is described by a `CellContext`, specifying its type, location, connections, memory state, and local computation program.

```
message CellContext {
   string type;
   uint32 xloc;
   uint32 yloc;
   map<string, Block> initial_state;
   repeated Operation operations;
   repeated string channel_names;
   map<string, float> channel_latencies;
   map<string, float> channel_bandwidths;
   uint32 num_registers;
   map<Operation, uint32> resource_constraint;
}
```

Figure 4.3: Protocol buffer definition for `CellContext`, modeling the behavior of a processing element.

Each cell can be a generator (e.g., an input feeder) or a regular processing element. The cell's set of incoming and outgoing communication channels, its initial data, and its computation program are all explicitly defined. A high-level diagram highlighting these features is shown in Figure 4.4.

Each entry in the `channel_names` array corresponds to a named communication port on the PE, and must match one of the channel identifiers specified in the global interconnect description. By using descriptive labels (e.g. `"[-1,0]"` for a left reception, `"[0,1]"` for a downward send, `"[-i,0]"`

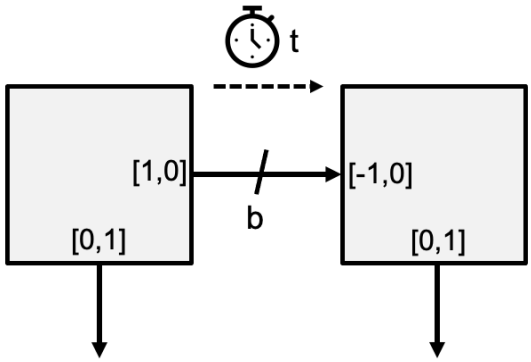


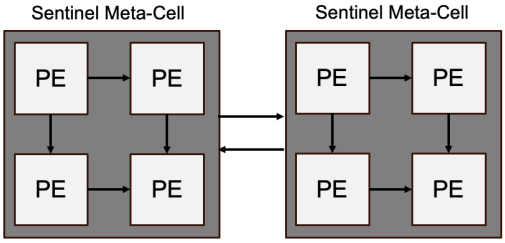
Figure 4.4: Processing Element Representation with Architectural Descriptors

for broadcast reference, or custom multicast group names), we can flexibly wire each CellContext into arbitrary topologies—from simple meshes to hierarchical or toroidal networks.

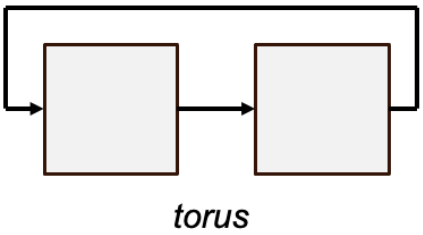
The `channel_latencies` and `channel_bandwidths` maps then attach quantitative timing parameters to each port. A nonzero latency entry models the one-time setup cost before the first word can traverse the link, while the bandwidth parameter (in tokens or blocks per cycle) governs sustained throughput and backpressure behavior. During DAM simulation, these values are used to enforce realistic enqueue/dequeue timing: if a sender attempts to push more data than the bandwidth allows, or if downstream buffers are full, the PE will stall until the channel becomes available. This fine-grained modeling makes it possible to study the impact of heterogeneous link speeds (for example, faster vertical "feed" channels and slower horizontal "shift" channels) on overall pipeline utilization and stall patterns.

Because each CellContext carries its own local copy of these channel properties, different PEs can have different communication characteristics, enabling heterogeneous designs where, say, boundary PEs or sentinel meta-cells enjoy higher-bandwidth links or lower latencies. Together with the per-PE `num_registers` and `resource_constraint` settings, this level of detail allows us to explore architectural trade-offs—trading off area (number of registers), power (fewer high-speed links), and performance (latency/bandwidth)—all within the same Protobuf-defined IR and without modifying simulator internals.

4.4 Topology and Simulation Structure



(a) Meta-cell hierarchies, called `Sentinels` in our IR



(b) Toroidal Connections

Figure 4.5: Example of Possible Topologies in our IR.

The IR describes the entire systolic array using a `SimulationConfig`. This includes the array's width and height, dimensionality (1D, 2D), topology (regular grid or torus), and the complete set of `CellContext` descriptions. Some simple examples of these are visualized in Figure 4.5.

Communication between cells is defined explicitly by listing logical channel connections rather than inferring them from grid adjacency. This generalizes the IR to handle non-standard communication patterns, including bypassing, folding, or multi-hop designs.

Additional modularity is provided through `SentinelConfig`, which describes sub-arrays used for input injection, boundary handling, or output monitoring. A top-level `Config` message combines the main array description and any sentinels.

4.5 Choice of Representation: Protocol Buffers

Several alternatives were considered for representing the architectural IR, including MLIR (Multi-Level Intermediate Representation) and custom file formats. Protocol buffers were ultimately selected due to their simplicity, portability, and efficiency.

MLIR provides powerful infrastructure for compiler-level transformations, but introduces significant complexity and runtime cost. In contrast, protobuf offers:

- Efficient binary serialization for large architectures
- Simple integration into existing Python and Rust toolchains

Furthermore, protobuf allows different parts of the project—the compiler backend, the DAM program generator, and the runtime simulator—to remain cleanly decoupled while communicating through a shared, extensible format.

4.6 The IR in our End-to-End Framework

In the complete project workflow, the IR acts as the bridge between compilation and simulation. The compiler emits a per-PE program, and specifies some architectural details that are exposed via the IR's Python front-end.

The DAM program generator then processes the IR, emitting a complete executable program for the DAM simulator. During simulation, cycle-accurate metrics such as execution latency, communication utilization, and memory usage are collected, providing key feedback on the suitability of a systolic array design for a given class of workloads. In the next chapter, we will discuss how the DAM program executes the description provided in the IR.

Chapter 5

Implementation: Lowering IR to DAM

Systolic-Sim provides a high-fidelity modeling environment for evaluating systolic arrays configured through the Dataflow Abstract Machine (DAM) abstraction. It is designed to faithfully capture key hardware features such as pipelining, backpressure, bandwidth-limited communication, and fine-grained performance metrics, while remaining modular and extensible enough to support a wide variety of architectural configurations and workloads. Inputs to the simulator consist of two main artifacts: a low-level program describing the per-cell computation schedule, and an architectural description specifying the communication topology and initial memory state, both serialized using Protocol Buffers.

Unlike traditional monolithic simulators that hardcode assumptions about array shape or operation sequences, the DAM-based simulator treats each processing element (PE) as an independent coroutine, communicating asynchronously over explicitly modeled channels. This allows for flexible simulation of regular systolic arrays, broadcast-enhanced topologies, and hierarchical organizations involving sentinel cells. By extracting rich metrics at both the per-cycle and per-cell levels, the simulator provides insights into utilization, latency, throughput, and pipeline behavior that guide the design of future specialized hardware.

5.1 Configuration Loading and Preprocessing

Simulation begins by loading a serialized `Config` object, which may contain either a single `SimulationConfig` describing the full array or multiple `SentinelConfig` objects to support hierarchical configurations. The deserialization pipeline, implemented through the `load_simulation_configs` function, parses these structures into internal Rust data types, populating the array dimensions, cell contexts, and operation sequences.

During this stage, special attention is paid to the differences between regular and sentinel configurations. Regular cells load their instruction streams directly from the protobuf, while sentinels may carry additional metadata and require remapping of channels to reflect their global aggregation role. Mapper functions translate serialized operations from the IR into variants of the simulator's ISA, performing type validation and ensuring semantic consistency across all instruction streams.

In broadcast-enabled topologies, preprocessing also includes a critical remapping step. Communication channels are transformed from static declarations such as `[0, -1]` into dynamic indices like `[0, -j]`, permitting the use of broadcast relays in lieu of dedicated point-to-point links (unless strict translation is specified). This is handled by the `remap_broadcast_ops` function, which traverses each instruction stream and rewrites channel identifiers where necessary to reflect the underlying broadcast behavior of the topology. We discuss the construction of broadcast-based topologies in a future section.

5.2 Channel Topology Construction

Communication is a cornerstone of systolic array performance. To model it accurately, the simulator builds a detailed communication graph by constructing sender-receiver pairs between cells. This is performed by the `fill_channel_map` algorithm, whose logic is summarized in Algorithm 1.

Algorithm 1 `fill_channel_map()`: Channel Map Construction

```
Initialize an empty channel map keyed by cell coordinates.
for each cell do
    Insert empty sender and receiver maps.
end for
for each declared channel in each cell do
    Parse the channel string to determine relative offset.
    Compute source and destination coordinates.
    if broadcast detected then
        Create intermediate broadcast relay node.
        Connect sender to relay and relay to multiple consumers.
    else
        Connect sender directly to receiver.
    end if
end for
Return the completed channel map.
```

This approach provides the flexibility that is fundamental to our framework. Point-to-point communication, row-wise broadcasts, and column-wise broadcasts are all supported uniformly by extending the basic channel map construction process. When a broadcast is detected, an additional relay context is created, ensuring that data sent once from a generator can be fanned out efficiently to multiple consumers. This mirrors the behavior of specialized hardware buses in real-world accelerator designs.

The `fill_channel_map` algorithm also supports topologies beyond simple 2D meshes. By varying the channel offsets and the locations of broadcast relays, users can model toroidal arrays, folded arrays, and even hierarchical communication patterns found in high-performance scientific computing hardware.

5.3 Runtime Context Construction

Once the channel topology has been established, the simulator constructs runtime contexts for each cell. This process is handled by the `process_simulation_config` function, which iterates over the set of loaded cells and initializes a `RuntimeCellContext` for each.

Each context is initialized with:

- A map of input receivers, keyed by channel identifiers.
- A map of output senders, keyed by channel identifiers.
- A local register file initialized from the cell's declared initial state.
- A sequence of instructions derived from the program emitted by the compiler.
- Local coordinate metadata (x, y location within the array).

The process is straightforward for regular cells. For sentinel cells, however, additional steps are performed. First, auxiliary channels are created to connect the sentinel to each cell it aggregates. Then, after the array is instantiated, the `join_sentinels` algorithm aggregates multiple sentinel contexts into a single logical entity, creating additional inter-sentinel communication links where necessary. This flexible modeling of sentinel behavior supports the simulation of large, hierarchical arrays composed of many smaller systolic blocks.

The full construction flow is summarized in Algorithm 2.

Algorithm 2 `process_simulation_config()`: Runtime Context Construction

```
for each simulation config do
    for each cell in the config do
        Create sender and receiver maps.
        Instantiate RuntimeCellContext with instructions and initial state.
        Connect cell to its neighbors via channels.
    end for
    if sentinel mode then
        Aggregate sentinel communication channels.
        Join multiple sentinels if necessary.
    end if
end for
```

The resulting system models an entire array of interconnected cells, each independently executing its own instruction stream and communicating over realistic links.

5.4 Execution Model and Cell Behavior

Each `RuntimeCellContext` corresponds to a processing element (PE) that executes a sequence of instructions defined by its local program. The core execution loop is implemented in the `run` method of each context, and is based on a simple, yet powerful interpreter model. The cell continuously dequeues instructions from its local buffer and attempts to execute them immediately. If an instruction depends on the availability of data on an input channel—such as a `RECV`—the context suspends execution until the required data becomes available, accurately modeling pipeline stalls and backpressure.

This instruction-driven simulation captures both computational and communication delays. Computational instructions, such as `GEMM` or `FFT`, are assigned multi-cycle latencies reflecting their real-world operation complexity. Communication instructions, on the other hand, block based on channel availability, modeling network contention and bandwidth constraints. This dual handling of computation and communication ensures that performance bottlenecks arising from either source are faithfully reproduced.

The execution model is asynchronous across all contexts. Each cell acts as a coroutine scheduled cooperatively within the DAM runtime, enabling large arrays to be simulated efficiently without thread contention. Logical time advances according to the progress of instructions and communication events, rather than being fixed globally, allowing natural modeling of fine-grained dynamic load imbalance.

5.5 Architectural Features Captured in the Simulator

The simulator provides detailed modeling of several critical architectural features relevant to accelerator design.

First, pipelining within each cell is explicitly modeled. Multi-cycle instructions such as `GEMM`

or `CHOL` execute over multiple logical cycles, and overlapping of communication and computation is possible. A cell can continue executing local instructions even while waiting for input or output channels, provided that data dependencies allow. This models realistic data-driven execution patterns, as seen in out-of-order architectures or deep pipelines with bypassing.

Second, the simulator captures communication backpressure. If a destination cell is not ready to receive data—for example, because it is stalled on a previous instruction—the corresponding sending operation blocks. This behavior is particularly important in systolic arrays where synchronization between producers and consumers is tight, and where bottlenecks can cascade through the array if not carefully managed.

Third, different network topologies are supported seamlessly. By simply changing the Protobuf configuration, users can simulate regular 2D meshes, toroidal grids, hierarchical arrays with sentinel cells, or broadcast-enhanced networks. The ability to study how topology affects latency, throughput, and utilization is critical for co-design of computation and communication.

Fourth, hierarchical memory access is naturally modeled. Generator cells simulate data feeders that populate the array, while regular cells perform local computation. This allows studying memory bandwidth constraints, staging behavior, and the impact of memory hierarchy on end-to-end performance.

Fifth, the simulator collects highly detailed metrics. Each cell logs per-cycle activity, active versus idle cycles, achieved throughput, and wall-clock elapsed time. These metrics allow post-processing to produce heatmaps of utilization, identify communication bottlenecks, and optimize both architectural parameters and mapping strategies.

Finally, extensibility is a core design principle. New operations can be easily added by extending the ISA and implementing corresponding execution paths in the CellContext. This ensures that the simulator remains adaptable to emerging workloads and evolving accelerator architectures.

5.5.1 Complex Operations and Computational Modeling

The simulator supports a rich instruction set that captures a wide range of computation types. Some examples are listed below:

- `GEMM`: Dense matrix-matrix multiply, modeled with cycle counts proportional to $M \times N \times K$.
- `FFT`: Fast Fourier Transform, modeled with $O(N \log N)$ complexity.
- `CHOL`: Cholesky factorization, modeled with $O(N^3/3)$ floating-point operations.
- `SYRK`: Symmetric rank-k updates, modeled with $O(N^3/2)$ operations.
- `TRISOLVE`: Triangular solves, modeled with $O(N^2)$ complexity.
- Scalar operations (`ADD`, `SUB`, `MUL`, `DIV`), modeled with single-cycle latencies.

Each of these operations is carefully instrumented within the `execute` method of the CellContext. For example, the execution of a `GEMM` instruction proceeds by retrieving the input tensors from the register file, invoking a highly optimized general matrix-matrix multiplication kernel from the `ndarray` library, and accounting for the corresponding number of cycles and throughput. Partial results are accumulated into output registers, modeling both compute latency and data movement costs.

By assigning realistic cycle counts to complex operations, the simulator captures the trade-offs between compute-bound and communication-bound regimes. For instance, in large systolic arrays performing GEMM, cells at the center of the array may be dominated by computation, while edge cells may experience communication stalls as they await input data. The simulator naturally exposes these dynamics without requiring manual modeling by the user.

5.5.2 Worked Example: Execution of a GEMM Cell

Consider the execution of a single `GEMM` instruction within a PE. The high-level pseudocode for this operation is shown in Algorithm 3.

Algorithm 3 Execution of GEMM in CellContext

1: Fetch matrices A, B, and C from register file.
2: Perform matrix multiply: $C \leftarrow A \times B + C$.
3: Update cycle count by $M \times N \times K + 1$.
4: Update throughput counter.
5: Write result back to register file.

This execution models the full data movement and computation sequence within a PE, accounting for both active computational cycles and register file updates. If a subsequent instruction depends on the result of this computation, the correct cycle accounting ensures that data dependencies are respected and that downstream consumers observe the correct latency.

Furthermore, the per-cycle active versus idle tracking ensures that partially utilized arrays—for example, during startup and tail phases of a computation—are accurately reflected in performance metrics.

5.6 Architectural Fidelity and Pipeline Modeling

A distinctive strength of this simulator is its fine-grained modeling of pipeline behavior. Each instruction within a PE has a defined latency and throughput.

Moreover, communication and computation can overlap naturally. For example, a PE may simultaneously perform a matrix multiply while prefetching the next set of tiles from its upstream neighbors. If upstream communication stalls occur—due to backpressure or bandwidth saturation—the PE will experience a stall at the next communication point, accurately modeling real-world performance degradations.

Pipelining efficiency, measured as the fraction of active cycles relative to total cycles, is recorded at the granularity of individual cells and can be aggregated across the array to identify pipeline bottlenecks and synchronization barriers.

This high level of fidelity makes the simulator an valuable tool for studying how design parameters such as pipeline depth, register file size, and channel bandwidth affect overall system performance.

5.7 Support for Complex Topologies

One of the key strengths of Systolic-Sim is its ability to model not only simple 2D meshes, but also a rich diversity of array topologies with simple modifications to the front-end. Instead, all topology variations are specified declaratively in the Protobuf configuration and interpreted at runtime.

The simulator naturally handles regular 2D meshes, where each PE is connected to its immediate

neighbors along cardinal directions. More interestingly, by adjusting the channel specification, users can instantiate toroidal arrays (where edges wrap around), arrays with asymmetric communication patterns, or arrays with dynamically inserted broadcast buses.

For instance, Cannon's algorithm for matrix multiplication requires a careful skewing and rotation of tiles across rows and columns. Using the simulator's dynamic channel mapping system, Cannon's shifting communication pattern can be modeled by specifying appropriate offsets for each communication channel at initialization.

Algorithm 4 summarizes the dynamic mapping of logical communication patterns to physical channel creation:

Algorithm 4 Dynamic Channel Mapping

```
for each cell do
    for each logical communication neighbor do
        Parse relative offset.
        if offset crosses boundary and wrap-around is desired then
            Adjust source and destination coordinates accordingly.
        end if
        Create sender-receiver pair or broadcast links as needed.
    end for
end for
```

This flexibility allows the simulator to support a wide variety of computation mappings, including blocked matrix multiplications, stencil computations, tensor contractions, and graph analytics, by adjusting only the input configuration.

5.7.1 Modeling Sentinel Cells

Another unique architectural feature supported is the modeling of sentinel cells. Sentinel cells act as coordinating nodes for a group of PEs, enabling hierarchical systolic array configurations. These configurations are essential when scaling to very large arrays, as they help manage the growing complexity of communication and synchronization.

Sentinel cells are instantiated as special `RuntimeCellContext` instances that aggregate communication endpoints from a group of regular PEs. Each sentinel maintains its own instruction stream and register file, just like regular cells, but typically focuses on higher-level coordination tasks such as synchronization, data aggregation, or partial result accumulation.

The process of constructing sentinel cells involves several phases:

First, for each group of cells assigned to a sentinel, additional channels are created linking the PEs to the sentinel context. These channels are inserted during the `process_simulation_config` phase. Each PE has dedicated send and receive channels for communication with its sentinel, enabling bi-directional data transfer.

Second, sentinel instruction streams are generated dynamically by parsing the nested `Operation` lists attached to the `SentinelConfig` structures in the Protobuf file. These operations can include scalar aggregations, barrier synchronizations, global reductions, or any user-defined instruction sequences.

Third, when multiple processing elements send partial results—such as partial sums—to a sentinel cell, the simulator dynamically constructs additional communication channels that allow the sentinel to receive and aggregate data from multiple sources. This models collective communication behavior commonly found in parallel reductions, where a single coordinating node gathers values from several upstream producers for further computation.

This modeling is essential for studying hierarchical array designs where local computation is performed at the PE level, and global coordination is offloaded to higher-level sentinels. It also allows experimentation with fault-tolerant designs, redundancy strategies, and large-scale synchronization protocols.

5.7.2 Broadcast Simulation and Aggregated Communication

Systolic-Sim supports efficient modeling of broadcast communication through the dynamic construction of `BroadcastContext` nodes. These contexts act as relay stations: they receive data from a source (usually a generator or feeder cell) and forward it to multiple downstream cells with minimal additional overhead.

The construction of broadcast contexts follows a simple yet powerful principle: when multiple

destination cells require data from a common source along the same row or column, instead of establishing separate point-to-point connections, a broadcast node is inserted. This reduces channel creation overhead and mirrors real-world hardware optimizations where broadcast buses replace redundant unicast links.

Algorithm 5 illustrates the creation of broadcast relays:

Algorithm 5 Broadcast Context Creation

```
Identify common source cell for multiple destination cells.
Instantiate a new BroadcastContext.
Connect source cell output to broadcast input.
for each destination cell do
    Connect broadcast output to destination cell input.
end for
```

This optimization enables highly efficient simulation of algorithms that depend heavily on broadcast, such as matrix-matrix multiplication (where entire rows or columns of tiles must be sent simultaneously) and certain stencil codes.

Moreover, because broadcasts are explicitly modeled, the simulator captures contention effects: if too many receivers attempt to pull from a shared broadcast channel simultaneously, communication latency can increase, reflecting real-world bus saturation behavior.

5.7.3 Advanced Communication Behaviors

Beyond basic nearest-neighbor and broadcast communication, the simulator models several advanced behaviors:

First, dynamic channel remapping is supported. During deserialization, the simulator can replace placeholder channel identifiers with concrete offsets that reflect the specific mapping of computation dimensions to spatial coordinates. This enables flexible reuse of the same high-level programs across different physical topologies.

Second, asymmetric communication patterns are allowed. Not all cells need to have symmetric communication partners. For example, in triangular solve computations, only cells above or below

the main diagonal may participate in communication. The simulator handles these cases naturally by simply omitting certain channel connections in the configuration phase.

Third, multi-hop communication patterns can be modeled by chaining `SEND` and `RECV` instructions across intermediate cells. This allows study of scenarios where data must traverse multiple hops to reach its final destination, introducing realistic network latency effects.

5.7.4 Fine-Grained Performance Metrics

A distinguishing feature of the simulator is its extremely fine-grained metrics collection. Every `CellContext` records:

- Total number of cycles executed.
- Number of active computation cycles.
- Number of idle cycles spent waiting for communication.
- Total floating-point operations performed (throughput).
- Elapsed wall-clock time (real execution time).
- Per-cycle activity vector (binary vector marking active vs. idle cycles).

These metrics are serialized to JSON files at the end of each simulation. They enable deep post-processing analyses, such as:

- Heatmaps of array utilization over time.
- Identification of communication bottlenecks and hotspots.
- Cycle-by-cycle visualization of pipelining efficiency.
- Throughput-versus-latency tradeoff curves for different topologies.

By providing detailed, per-cell introspection, the simulator allows users to debug performance issues at a level of detail rarely available even in commercial hardware simulators.

5.8 Summary and Outlook

The simulator described in this work offers a unique combination of flexibility, architectural fidelity, and extensibility. It models not only the computational workloads assigned to each PE but also the rich communication and synchronization behaviors that dominate real-world systolic array performance.

Its support for dynamic channel mapping, broadcast communication, hierarchical sentinel cells, complex instruction streams, and fine-grained performance metrics makes it a powerful tool for hardware-software co-design. Researchers can use it to experiment with new architectural ideas, test novel mapping strategies, or evaluate the impact of network topology on algorithm scalability.

Importantly, because the simulator is based on a modular design, future extensions are straightforward. Support for bounded channels, network contention models, speculative execution, fault tolerance, or heterogeneous compute resources (e.g., mixing general-purpose and specialized cores) could be added by extending the existing framework.

In the next chapter, we leverage this simulator to conduct a series of experiments across different workloads, analyzing how architectural parameters impact performance, utilization, and energy efficiency across a variety of systolic array configurations.

Chapter 6

Evaluation

In this chapter, we will highlight some of the use cases of our simulator, spanning applications and architectural specifications.

6.1 Evaluation of Cannon's GEMM

Experimental Setup. We evaluate the classic Cannon's algorithm for matrix multiplication,

$$C_{i,j} \;=\; \sum_{k=0}^{p-1} A_{i,k}\, B_{k,j},$$

on a $p \times p$ systolic array with $p = 3$. The high-level recurrence is lowered to per-PE programs via our custom IR and executed in the DAM simulation framework. To focus on the core dataflow behavior, we configure the interconnect bandwidth to be effectively infinite and allocate each PE a sufficiently large register file—eliminating stalls due to communication contention or register spills.

Utilization Results. Figure 6.1 shows the utilization heatmap for each PE. All PEs achieve high utilization (above 85%), confirming that Cannon's data-shifting schedule keeps the pipeline nearly full. We observe a diagonal "wavefront" pattern: PEs on the leading edge (e.g. $(0, 0)$) begin computation earliest and maintain peak activity, while PEs further along the schedule (toward $(2, 2)$) exhibit slightly lower utilization as they wait for operands to propagate through the network. This

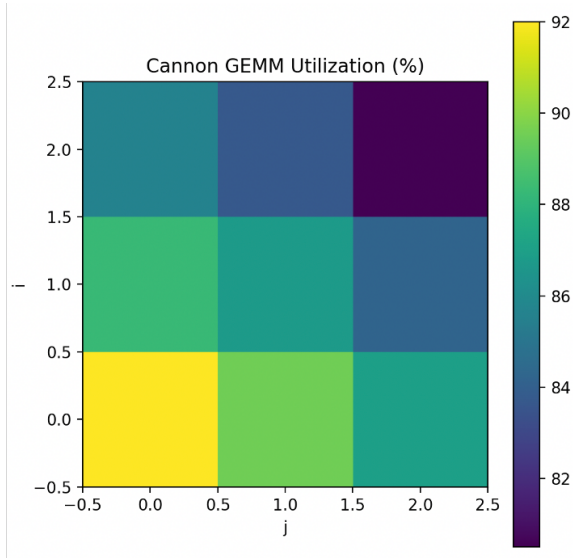


Figure 6.1: Cannon's GEMM on 3x3 Systolic Array

gradient precisely matches the pipeline fill and drain behavior inherent in the spatial mapping of the recurrence, and validates our simulator's accuracy in modeling fine-grained execution dynamics.

6.2 Evaluation of Triangular Solve (TRSM)

Problem Formulation. We consider the forward substitution problem

$$L\,x = b, \quad L \in R^{n\times n} \text{ lower–triangular}, \quad b \in R^n,$$

whose solution is given element-wise by

$$x_i \;=\; \frac{1}{L_{i,i}}\Big(b_i - \sum_{k=1}^{i-1} L_{i,k}\,x_k\Big), \quad i = 1,\ldots,n.$$

Unlike dense matrix multiplication, TRSM exhibits a strict dependency chain along the diagonal: each x_i cannot be computed until all earlier x_k $(k < i)$ are known.

Experimental Setup. Triangular solve is given by the following loop nest:

```
for i = 0 to M-1:
  for j = 0 to N-1:
    X[i][j] = B[i][j]
    for k = 0 to i-1:
      X[i][j] -= L[i][k] * X[k][j]
    X[i][j] /= L[i][i]
```

We map this recurrence onto a 4×4 triangular systolic array. PEs implement both:

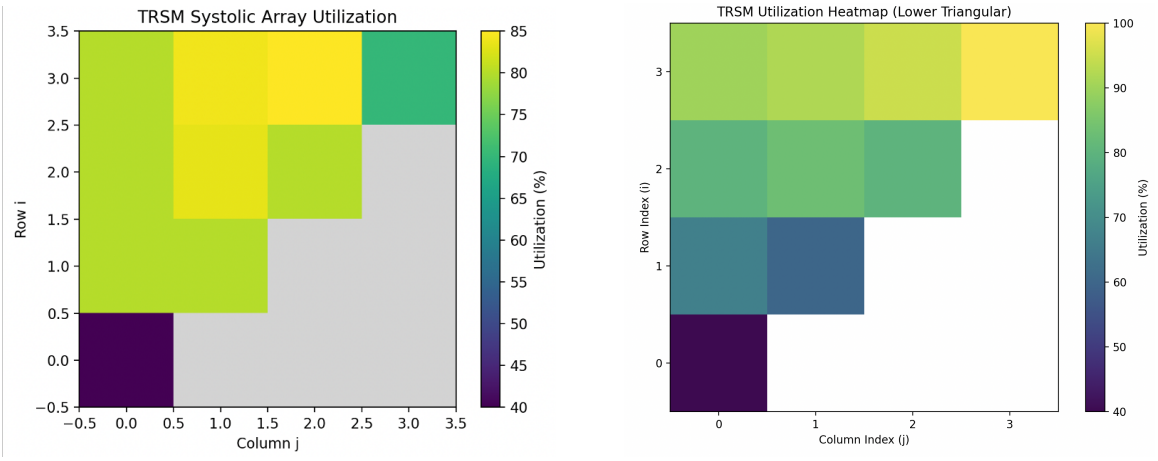
- The scalar divide $x_i = (b_i - \sum_{k=1}^{i-1} L_{i,k} x_k)/L_{i,i}$ once per pivot and then broadcast x_i to the next row and column.

- rank–1 update $b_j \leftarrow b_j - L_{j,i}\, x_i$ (a GEMM-style step) repeatedly for all pivots, before receiving the final diagonal value to compute their own x_j.

We lower both programs from our custom IR into the DAM simulator. Leveraging our domain-space exploration engine, we consider 2 design points: one for symmetric systolic arrays and another with asymmetry.

For our *symmetric systolic array*: we make the interconnect bandwidth and local resources high enough such that there is no contention.

For our *asymmetric systolic array*: we make the bandwidth higher on vertical interconnects (i.e., no stalling required), and the bandwidth fixed at 1 block per cycle on horizontal interconnects (i.e., stalling required). The resources are also asymmetric. The number of registers available to interior cells is 4 while the number of registers available to diagonal cells is 8. This varied asymmetry, across resource constraints and interconnect bandwidths, reflect optimizations that can be easily tweaked in our highly extensible simulator.



(a) TRSM Utilization, Asymmetric Systolic Array (b) TRSM Utilization, Symmetric Systolic Array

Figure 6.2: Comparison of utilization between asymmetric and symmetric systolic arrays for TRSM.

Utilization Results and Analysis. Figure 6.2a reports the per-PE utilization heatmap under our asymmetric configuration: vertical interconnects have effectively infinite bandwidth (no stalls), horizontal links are limited to one block per cycle (introducing backpressure), interior cells are allotted 4 registers each, and diagonal cells have 8 registers each. Despite these constraints, interior PEs sustain high utilization (often above 80%) because they repeatedly execute rank–1 update (GEMM) operations at each pivot. Their smaller register file does force occasional stalls—either to wait for horizontal `RECV` capacity or to free up buffer slots—but the steady stream of updates keeps them busy for the vast majority of cycles. By contrast, diagonal PEs average around 60–65% utilization: they perform exactly one scalar TRSM per pivot, then idle until the next chain of dependencies arrives. Their larger register file and unthrottled vertical sends mean they never stall on communication, but they still sit idle for many cycles.

Furthermore, pipeline fill and drain effects accentuate these differences. The first diagonal PE (at $(1, 1)$) shows the lowest utilization, as it incurs both the initial pipeline fill and the final drain without any intervening work. Similarly, the bottom-right diagonal PE completes its final pivot early and remains idle for the remainder of the simulation. These imbalances highlight how our extensible DAM framework can expose the combined impact of interconnect bandwidth asymmetries and per-PE resource constraints—insights that would be difficult to obtain without cycle-accurate modeling.

When we compare these results against the symmetric setup—where both horizontal and vertical links have identical unthrottled bandwidth and all PEs share the same register capacity—the utilization heatmap becomes dominated by a pronounced wavefront pattern. Early-starting PEs achieve nearly 100% utilization, while later PEs stay at low activity until their data dependencies arrive. In contrast, the asymmetric case smooths out these peaks and valleys: although diagonal cells still idle between pivots, their extra registers and the prioritized vertical bandwidth help them overlap communication and computation more effectively. As a result, the overall utilization distribution is far more uniform, demonstrating that targeted asymmetries in resource and interconnect provisioning can significantly improve load balance in dependency-driven kernels.

6.3 Evaluation of Fast Fourier Transform (FFT)

FFT is a canonical divide-and-conquer algorithm and is therefore well-suited for hierarchical composition via meta-cells. The recurrence is formulated in the next subsection.

6.3.1 Problem Formulation

Let N be the input length and define the twiddle factor

$$\omega_N \;=\; e^{-2\pi i/N}.$$

Split the input sequence $\{x_n\}_{n=0}^{N-1}$ into evens and odds,

$$E_j = x_{2j}, \quad O_j = x_{2j+1}, \quad j = 0, 1, \ldots, \frac{N}{2} - 1.$$

The base case is

$$\mathrm{FFT}_1\big([x_0]\big) \;=\; [\,x_0\,].$$

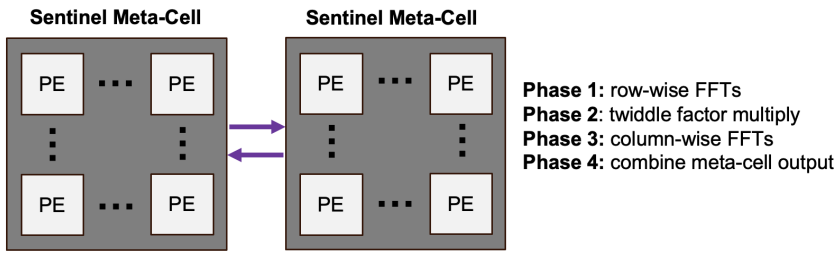


Figure 6.3: Phases of FFT Computation

For $0 \leq k < \frac{N}{2}$, the recurrence is

$$\begin{aligned} X[k] &= \mathrm{FFT}_{N/2}(E)[k] + \omega_N^k \, \mathrm{FFT}_{N/2}(O)[k], \\ X\left[k + \tfrac{N}{2}\right] &= \mathrm{FFT}_{N/2}(E)[k] - \omega_N^k \, \mathrm{FFT}_{N/2}(O)[k]. \end{aligned}$$

6.3.2 Experimental Setup

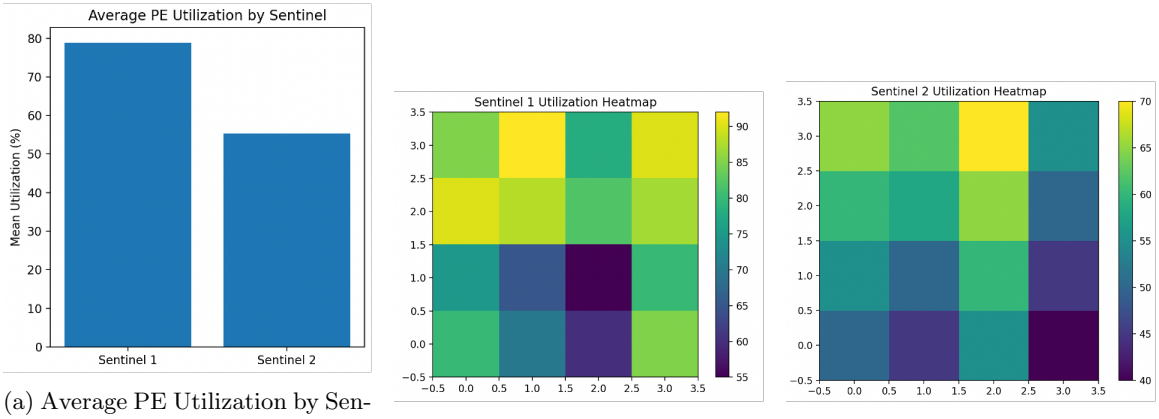
Each N-point FFT is executed in four pipelined phases within each sentinel meta-cell, as visualized in Figure 6.3:

1. **Row-wise FFTs:** Perform independent length-$N/2$ FFTs across each row of the meta-cell, exchanging partial results horizontally between PEs.

2. **Twiddle-factor Multiply:** Multiply each row output by the corresponding power of ω_N^k, using locally stored twiddle tables to avoid global broadcasts.

3. **Column-wise FFTs:** Execute length-$N/2$ FFTs down each column, with PEs communicating vertically to implement the butterfly network.

4. **Meta-Cell Combine:** Exchange the even and odd sub-outputs between adjacent meta-cells and apply the final butterfly merge, interleaving results to form the full N-point transform.



(a) Average PE Utilization by Sentinel

(b) Sentinel 1 Utilization Heatmap

(c) Sentinel 2 Utilization Heatmap

Figure 6.4: Evaluation of the N-point FFT on two sentinel meta-cells: (a) mean PE utilization per meta-cell, (b) detailed PE utilization within the even-index (Sentinel 1) subgrid, and (c) detailed PE utilization within the odd-index (Sentinel 2) subgrid.

6.3.3 Utilization Results and Analysis

Figure 6.4a shows that the "even" meta-cell (Sentinel 1) achieves nearly 80% mean utilization, while the "odd" meta-cell (Sentinel 2) lags at around 55%. This gap arises for two main reasons. First, Sentinel 1 preloads its twiddle-factor tables and begins the row-wise FFT immediately, whereas Sentinel 2 must wait for the shifted odd elements and for the cross-sentinel boundary exchange before it can start its butterfly stages. Second, the final combine phase—the only point where data is exchanged between meta-cells—imposes extra SEND/RECV latency on the initiating side, further delaying Sentinel 2's work.

The heatmaps in Figures 6.4b and 6.4c reveal how these effects manifest spatially. In Sentinel 1 (Figure 6.4b), utilization is very high ($\geq$90%) in the interior PEs during the row- and column-FFT phases, dipping only slightly in the middle row where the twiddle multiply occurs. In Sentinel 2 (Figure 6.4c), utilization is lower overall, with the first row and first few columns showing under 50% activity as those PEs await both data and twiddle factors. Our DSE engine can be used in this way to reveal insights into balancing preload overhead and inter-meta-cell communication to achieve uniform pipeline utilization in a hierarchical mapping such as this one.

6.4 Evaluation of Cholesky Decomposition

We evaluate the performance and resource utilization of Cholesky decomposition on various systolic array configurations using our simulator. The results for one of these experiments is shown in the following subsections.

6.4.1 Problem Formulation and Experimental Setup

The recurrence is given by:

$$L_{i,j} = \begin{cases} \sqrt{A_{i,i} - \sum_{k=0}^{i-1} L_{i,k}^2} & \text{if } i = j \\ \frac{1}{L_{j,j}} \left(A_{i,j} - \sum_{k=0}^{j-1} L_{i,k} L_{j,k} \right) & \text{if } i > j \\ 0 & \text{if } i < j \end{cases}$$

This can be equivalently written as the following loop-nest:

```
for i = 0 to N-1:
  for j = 0 to i:
    sum = A[i][j]
    for k = 0 to j-1:
      sum -= L[i][k] * L[j][k]
    if i == j:
      L[i][j] = sqrt(sum)
    else:
      L[i][j] = sum / L[j][j]
```

We then map this to equivalent PE programs. PEs implement both:

- The next pivot element:

$$L_{k,k} = \sqrt{A_{k,k} - \sum_{j=1}^{k-1} L_{k,j}^2}$$

.

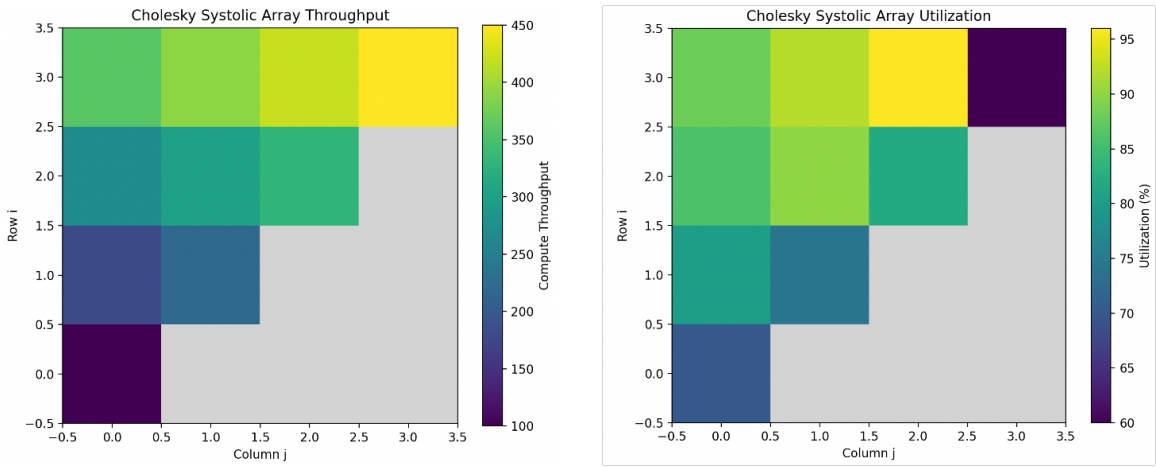
- The update step, given by:

$$L_{i,k} = \frac{A_{i,k} - \sum_{j=1}^{k-1} L_{i,j} L_{k,j}}{L_{k,k}}$$

.

This wavefront pattern of diagonal computation followed by column-wise updates aligns naturally with systolic execution, where each PE progresses in lockstep with its neighbors.

6.4.2 Utilization and Throughput Results and Analysis



(a) Cholesky throughput heatmap

(b) Cholesky utilization heatmap

Figure 6.5: Evaluation of Cholesky decomposition on a 4×4 systolic array: (a) per-PE compute throughput, (b) per-PE utilization.

The throughput heatmap in Figure 6.5a shows a clear "striated" pattern down the rows of the array. PEs in row i process i rank–1 updates (one per pivot) before the final TRSM, so lower rows accumulate more work and achieve higher aggregate throughput (bottom row peaks at 450 units vs. 100 at the top). This reflects the wavefront nature of Cholesky: each column of PEs must wait for the diagonal pivot, then each interior PE executes a GEMM-style update once per pivot.

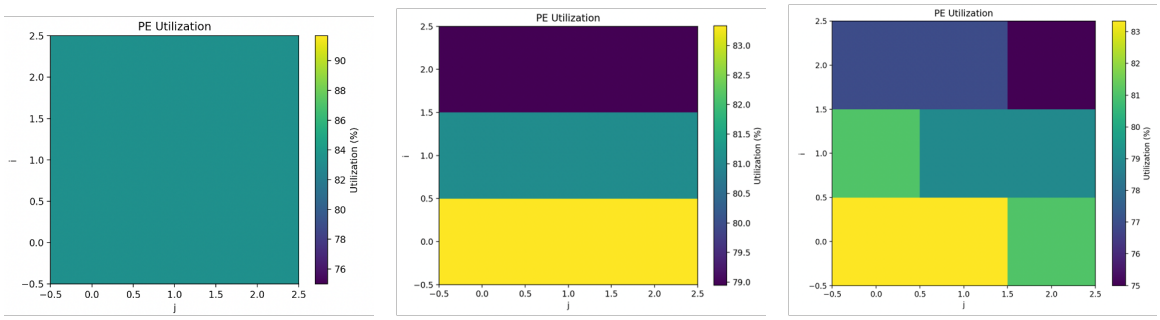
Figure 6.5b reveals that interior PEs sustain very high utilization (above 85%) because they continuously perform kernel updates across nearly every cycle. In contrast, diagonal PEs exhibit lower utilization (60 – 75%) since they only execute one TRSM per pivot and otherwise idle while waiting for their next input dependencies. The bottom-right diagonal cell, in particular, finishes its

final pivot early and drains the pipeline without further work, creating the dark corner in the utility map. These two plots together highlight both the inherent load imbalance of a direct Cholesky mapping and the value of our simulator in quantifying when and where PEs become bottlenecks.

6.5 Evaluation of Communication Strategies, GEMM

As discussed earlier in this thesis, broadcast connections excel when the same data block must be delivered simultaneously to multiple PEs. By constructing a dedicated broadcast tree, each PE can receive its operand with minimal latency, enabling all PEs to begin computation in lockstep. This "free" data distribution maximizes parallelism and hides communication delays behind pipelined transfers.

However, broadcast hardware incurs significant area and power overhead for high-fanout networks or large buffers. In bandwidth-limited designs, a broadcast tree can monopolize shared links, creating contention that slows other traffic. Moreover, static broadcast structures lack flexibility for irregular communication patterns—such as the combine phase in FFT or sparse updates—forcing designers to provision general-purpose links that may go underutilized. A common compromise is a hybrid network: static broadcast for the heavy-data GEMM panels and point-to-point or narrow multicast for less regular workloads, balancing latency, area, and utilization.



(a) Unlimited bandwidth (b) Uniform bandwidth = 1 (c) Bandwidth = [2,1,2]

Figure 6.6: PE utilization under different broadcast-channel bandwidth configurations for Cannon's GEMM on a 3×3 array.

Broadcast Bandwidth Impact. Figure 6.6 illustrates how limiting broadcast-channel bandwidth reshapes the utilization profile, even when the topology and kernel remain fixed. With unlimited bandwidth (a), all PEs sustain roughly equal utilization, since data arrives instantly and the pipeline is never starved. When every link is throttled to one block per cycle (b), a clear diagonal wavefront reappears: the first row completes quickly and idles, the middle row achieves moderate utilization, and the final row is most active only after its turn. Finally, in the heterogeneous setting (c) with bandwidths $[2, 1, 2]$ on successive links, we see a more nuanced pattern—PEs on the middle column suffer the greatest stalls, while those on the side with extra bandwidth recover higher utilization. These results demonstrate that even modest asymmetries in channel capacity can be exploited to smooth pipeline utilization and mitigate the classical wavefront imbalance in systolic-array GEMM.

6.6 Summary of Evaluation

In this chapter, we demonstrated the versatility and fidelity of our DAM-based simulator across a range of canonical kernels and architectural variants. We began with Cannon's GEMM on a 3×3 systolic array, validating that the expected wavefront utilization pattern emerges under ideal bandwidth. We then explored triangular solve (TRSM) on a 4×4 array, showing how asymmetric bandwidth and per-PE register allocations can dramatically improve load balance compared to a fully symmetric design. Next, we mapped the Cooley–Tukey FFT onto a hierarchy of sentinel meta-cells, revealing the impact of twiddle-factor preloads and cross-cell handoffs on even/odd subgrid utilizations. We followed with Cholesky decomposition, where throughput and utilization heatmaps exposed the inherent striations and idling of diagonal cells. Finally, we analyzed communication strategies for GEMM, illustrating how static broadcasts and point-to-point links trade off latency, area, and utilization, and how modest bandwidth asymmetries can smooth the classical wavefront imbalance. Together, these case studies underscore our framework's ability to model fine-grained compute and communication behaviors and to guide rapid design-space exploration of custom systolic architectures.

Chapter 7

Conclusions

In this work, we have introduced a novel, end-to-end framework for rapid prototyping and evaluation of custom systolic architectures driven by recurrence-based workloads. At its core is a flexible, Protobuf-based intermediate representation (IR) that cleanly separates computation, communication, and topology, enabling us to target a wide range of kernels—GEMM, TRSM, FFT, Cholesky—without rewriting simulator internals. By building our simulation backend on the Dataflow Abstract Machine (DAM), we achieve cycle-accurate performance modeling that captures both compute latencies and communication behaviors, including pipeline stalls, backpressure, and broadcast contention.

Our domain-space exploration (DSE) engine leverages these detailed metrics to sweep architecture parameters—PE grid dimensions, interconnect bandwidths, register file sizes—and uncover design trade-offs. We demonstrated that modest asymmetries, such as prioritizing vertical links or provisioning extra registers on diagonal cells, can dramatically smooth utilization and improve load balance in dependency-driven kernels. We also showed how hierarchical "meta-cell" mappings naturally support divide-and-conquer algorithms like FFT, and how broadcast networks must be carefully balanced against area and power to achieve both low latency and flexibility.

To the best of our knowledge, this is the first open-framework that combines a general-purpose, recurrence-centric IR with a scalable, cycle-accurate dataflow simulator and an automated exploration engine. This work lays the foundation for future research in hardware-software co-design,

from compiler-driven mapping of new recurrences to automated RTL generation of optimized accelerator fabrics. In ongoing and future work, we plan to integrate our IR with high-level DSLs, extend the simulator to model power and thermal effects, and close the loop by generating synthesizable hardware descriptions directly from design-space specifications.

Overall, our contributions demonstrate that careful separation of concerns, coupled with fine-grained performance modeling, can unlock rapid iteration on novel accelerator architectures—driving a faster, more efficient transition from algorithm to silicon.

Bibliography

Ahmad, H., Tanvir, M., Hanif, M. A., Javed, M. U., Hafiz, R., & Shafique, M. (2019). *Systimator: A design space exploration methodology for systolic array based cnns acceleration on the fpga-based edge nodes.* (arXiv preprint arXiv:1901.04986)

Cong, J., & Wang, J. (2018). Polysa: Polyhedral-based systolic array auto-compilation. In *Proceedings of the 2018 ieee/acm international conference on computer-aided design (iccad)* (pp. 1–8).

Genc, H., Haj-Ali, A., Iyer, V., Amid, A., Mao, H., Wright, J., ... Shao, Y. S. (2019). *Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration.* (arXiv preprint arXiv:1911.09925)

Kung, H. T., & Leiserson, C. E. (1979). Systolic arrays for vlsi. In *Sparse matrix proceedings* (pp. 256–282). Society for Industrial and Applied Mathematics.

Mathur, R., Kumar, A. K. A., John, L. K., & Kulkarni, J. P. (2021). Thermal-aware design space exploration of 3-d systolic ml accelerators. *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, *7*, 70–78.

Samajdar, A., Zhu, Y., Whatmough, P., Mattina, M., & Krishna, T. (2018). *Scale-sim: Systolic cnn accelerator simulator.* (arXiv preprint arXiv:1811.02883)

Wang, J., & Cong, J. (2021). *Search for optimal systolic arrays: A comprehensive automated exploration framework and lessons learned.* (arXiv preprint arXiv:2111.14252)

Wang, J., Guo, L., & Cong, J. (2021). Autosa: A polyhedral compiler for high-performance systolic arrays on fpga. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, *14*(3), 1–26.

Wu, D. (2022). *usystolic-sim: A systolic array simulator for multi-bit and unary computing.* (GitHub repository)

Xu, R., Ma, S., Guo, Y., & Li, D. (2023). A survey of design and optimization for systolic array-based dnn accelerators. *ACM Computing Surveys*, *56*(1), Article 20.

Yüzügüler, A. C., Sönmez, C., Drumond, M., Oh, Y., Falsafi, B., & Frossard, P. (2023). Scale-out systolic arrays. *ACM Transactions on Architecture and Code Optimization*, *20*(2), Article 27.