Cyclotron: Compilation of Recurrences to Distributed and Systolic Architectures

SHIV SUNDRAM, Stanford University, USA
AKHILESH BALASINGAM, Stanford University, USA
NATHAN ZHANG, Stanford University, USA
KUNLE OLUKOTUN, Stanford University, USA
FREDRIK KJOLSTAD, Stanford University, USA

We present Cyclotron, a framework and compiler for using recurrence equations to express streaming dataflow algorithms, which then get portably compiled to distributed topologies of interlinked processors. Our framework provides an input language of recurrences over logical tensors, which then gets lowered into an intermediate language of recurrences over logical iteration spaces, and finally into programs of send, receive, and computation operations specific to each individual processor. In Cyclotron's IR, programs are optimized such that external memory interactions are confined to the boundaries of the iteration space. Within inner iteration spaces, all data accesses become local: data accesses target values residing in local fast memory or on neighboring processing units, avoiding costly memory movement. We provide a scheduling language allowing users to define how data gets streamed and broadcasted between processors, enabling pipelined execution of computation kernels over distributed topologies of processing elements. We demonstrate the portability of our approach by compiling our IR to a reconfigurable simulator of systolic arrays and chiplet-style distributed hardware, as well as to distributed-memory CPU clusters. In the simulated reconfigurable setting, we use our compiler for hardware design space exploration in which link costs and latencies can be specified. In the distributed CPU setting, we show how to use recurrences and our scheduling language to express various matrix multiplication routines (Cannon, SUMMA, PUMMA, weight stationary) and solvers (Triangular solve and Cholesky). For matrix multiplication and the triangular solve, we generate distributed implementations competitive with ScaLAPACK.

1 Introduction

Generations of architectural improvements have augmented the performance gap between data movement and computation. Performance optimization is now increasingly concerned with optimizing memory movement and network traffic, and less on optimizing and reducing the number of floating operations performed. Modern architectures, whether it be a single GPU/accelerator, or a cluster of CPUs, share a common motif of being an interconnected grid of individual processing elements (PEs), arranged either in a lattice or a tree of wires.

These architectures emphasize the need to keep data close to processing elements, and to move data exclusively amongst neighbors, thus avoiding expensive accesses to offchip memory. Therefore, optimized kernels, like those for matrix multiply, are now designed as programs where each timestep consists of three phases. A *receive* phase to import data from nearest neighbors. A *compute* phase to operate on this data. And a final *send* phase to send data to neighboring PEs for the subsequent timestep. Such a design allows all data movement to take the form of less expensive nearest neighbor exchanges (which can often be overlapped with the computation), ameliorating the need for deeper memory movement across machine hierarchies.

The majority of work in this space has focused on optimizing matrix multiply. In a distributed memory setting, communication-avoiding matrix multiply algorithms like Cannon's, SUMMA, and PUMMA all fit within this three-phase motif. On a single chip setting, this has manifested in the

Authors' Contact Information: Shiv Sundram, shiv1@stanford.edu, Stanford University, Stanford, California, USA; Akhilesh Balasingam, avb03@stanford.edu, Stanford University, Stanford, California, USA; Nathan Zhang, stanfurd@stanford.edu, Stanford University, Stanford, California, USA; Kunle Olukotun, kunle@stanford.edu, Stanford University, Stanford, California, USA; Fredrik Kjolstad, kjolstad@stanford.edu, Stanford University, Stanford, California, USA.

Table 1. Features of different programming frameworks for recurrences and matrix multiplies. A ✔denotes the framework contains that feature.

Framework	Space-time mappings	Multi-node	Chiplet/simulator	Matmul	Recurrences	Scheduling Language
GEMMINI	✔		✔	✔		
DISTAL		✔		✔		✔
Rajopadhyaye et al.	✔			✔	✔	
Recuma				✔	✔	✔
AutoSA	✔		✔	✔		
Cyclotron (this work)	✔	✔	✔	✔	✔	✔

development of systolic architectures, like the Tensor Processing Unit, in which a grid of multiply accumulator units collectively compute matrix products, with a similar dataflow to Cannon's algorithm, but where all the PEs are on a single chip, as opposed to being different CPUs entirely.

However, we lack a general framework for co-designing both a portable programming language and an architectural design space framework to encapsulate the full space of these algorithms, which reaches far beyond matrix multiply. Table 1 illustrates the state of current frameworks for such algorithms. Systems like DISTAL [Yadav et al. 2022] and architectural design space frameworks like GEMMINI [Genc et al. 2021], for example, focus on distributed matrix multiply operations. Furthermore, routines like triangular solve, the Cholesky Decomposition, and Flash Attention, all exhibit similar dataflow behavior to matrix multiply, and all fit in a general language of recurrences. They differ in terms of what is computed on each PE, which data is sent on the wires, and in which directions. These computations can have different data dependencies, which must be considered when scheduling computation and data movement. Nevertheless, they fall within this three phase pipeline.

A common framework and programming languages for dataflow computations, would allow us to describe all such computations succinctly in this language, while making it easy to explore the design space of each computation itself. The affine recurrence model and the Warp processor [Lam and Wolf 2004] demonstrated this idea in the context of hardware-oriented systolic compilation, providing a tool-chain for mapping imperative and affine loop nests onto a specific processor template. However, their compilation model was tightly coupled to a fixed hardware target, making it difficult to generalize beyond that setting or to distributed implementations. Other mathematical frameworks, like the polyhedral model [Feautrier 1991], and the systolic synthesis framework of Rajopadhye and Fujimoto [Rajopadhye and Fujimoto 1990], provide a mathematical foundation for dependency management and software pipelining. However, they are largely theoretical,they do not show how to generate processor-level code. They rely on successive user-specified linear transformations to schedule and optimize computation, an assumption unnecessary for practically compiling and optimizing recurrences onto distributed architectures.

We show how to describe all these dataflow algorithms in a language of recurrence equations, which along with our scheduling language, embed the dependencies, data-reuse, and dataflow of the underlying computations. We also show how the schedule is needed to add essential information on how to specify streaming behavior of the computations. These languages are then compilable to both an MPI based distributed memory architecture, where we get competitive performance with handwritten libraries, as well as a single chip dataflow simulator.

2 Motivating Example

To demonstrate how recurrences can elegantly describe a computation, its data reuse, dataflow, and local communication, we use a matrix multiply and triangular solve as our running examples; each illustrates how to handle recurrences without dependencies and recurrences with dependencies, respectively. Figure 1 contrasts the two recurrences: matrix multiply, whose outputs are independent

Matrix Multiplication (Tensor Algebra)

$$C_{ij} = \sum_k A_{ik}\, B_{kj}$$

or equivalently $C_{ij} = A_{ik}B_{kj}$

Triangular Solve with Multiple Right Hand Sides (Recurrence Form)

$$X_{ri} = \frac{1}{L_{ii}}\left(B_{ri} - \sum_{k<i} L_{ij}X_{rj}\right)$$

where $j < i$

Fig. 1. Left: Tensor algebra for matrix multiplication, which contains no dependencies between outputs. Right: Recurrence relation for forward triangular solve, which contains output dependencies in which X_{ri} is dependent on X_{rj} where $j < i$

and can be computed in parallel, and triangular solve, whose outputs depend on previously computed values. In the latter, already calculated entries of output tensor X are reused to compute future ones.

Recurrences are equations involving mathematical operations over indexed tensors. Representative examples of such recurrences are shown in Figure 2, and the grammar used to describe them formally is shown in Figure 3. They form a superset over the language of Tensor Algebra, which is a language of elementwise operations and summations over indexed tensors. The typical Tensor Algebra form of a matrix multiply is simply $C_ij = \sum_k A_{ik}B_{kj}$. Recurrences extend tensor algebra with additional constructs

(1) Constraints, in which iteration variables (i.e. i and j) can be related to each other via inequalities (e.g., $j < i$ in the triangular solve in Figure 1)
(2) Dependencies among outputs, in which the result tensor appears on the left and right side of the equation, meaning output values are needed to calculate future outputs (e.g. X_{ij} in the triangular solve depends on all X_{rj} where $j < i$.

Both the matrix multiply recurrence and the triangular solve recurrence are declarative equations that show how to mathematically compute the outputs, given the dependencies. Both recurrences, when augmented with a schedule denoting a desired loop ordering (e.g., ijk, jki etc. for matrix multiply), can be lowered to imperative programs of loops and compute statements. The RECUMA [Sundram et al. 2024] and REPTILE [Tariq et al. 2025] showed how user-provided recurrences, along with user-provided schedules can be lowered to C code that runs on a single processor and achieves performance parity with hand-optimized code. In the generated C code, tensors can be randomly accessed directly with index expressions (e.g. C_{ij} can be accessed as C[i*N+j]), and these accesses translate into random access requests to memory.

While RECUMA and REPTILE can lower these recurrences to efficient single-node code, distributed execution introduces a new challenge: data cannot be accessed arbitrarily. Random accesses to memory are either impermissible or impossible when tensors are sharded across memory; there is no guarantee that an arbitrary processor has access to an arbitrary element of a tensor. In both a distributed matrix multiply (over tensors A B and C) and a distributed triangular solve (over tensors L and X and B), a tensor can be either statically distributed over processors, or may stream across processors, such that the same piece of data passes through every processor where it is needed for a computation.

Regardless if the target architecture is an interconnected lattice of chiplets on the same chip/node, or a grid of CPU processors in a supercomputer, implementing either recurrence in a distributed setting thus requires decisions about how and when data is sharded, streamed, or broadcasted across processing elements. Dependencies in the computations must be respected, and in the cases

Matrix multiply

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Triangular solve (lower L, forward)

$$X_{ri} = \frac{1}{L_{ii}} \left(B_{ri} - \sum_{j<i} L_{ij} X_{rj} \right)$$

Prefix sum

$$P_i = P_{i-1} + A_i$$

Cholesky (lower L)

$$L_{ii} = \sqrt{A_{ii} - \sum_{k<i} L_{ik}^2} \quad L_{ij} = \frac{A_{ij} - \sum_{k<j} L_{ik} L_{jk}}{L_{jj}} \quad : j < i$$

Fig. 2. Representative recurrences

ConstInt	n		
Tensor	t		
IndexVar	v		
Index	i	::=	$v + n \mid n$
TensorAccess	ta	::=	$t_{i+}^{i^*}$
Expr	e	::=	$ta \mid \sum_v e \mid const$
			$\mid \sqrt{e} \mid e + e \mid e\ e \mid \cdots$
Recurrence	r	::=	$ta = e$
Constraint	c	::=	$v < i \mid v \leq i \mid v = i$
Constraints	cs	::=	$c \mid c, cs$
ConstrainedRec	cr	::=	$r : cs$
Program	p	::=	$cr \mid cr, p$

Fig. 3. Recurrence Grammar

where multiple pieces of input data are needed for a single computation (e.g. A_{ik} and B_{kj} in matrix multiply), a program must guarantee that all pieces are available at the same processor at the same time, in order to compute with them. In the matrix multiply example, compiling recurrences to distribute processors requires careful handling of:

(1) **Communication.** The matrix multiply involves three iteration variables (i, j, k), while each tensor uses only two. For instance, A_{ik} lacks j, meaning that once A_{ik} is loaded, it can be reused across all j for a fixed (i, k). In a distributed program, this reuse manifests as a communication pattern: A_{ik} may be broadcast or streamed systolically to the processors computing the corresponding C_{ij}. Similarly, B_{kj} may be distributed across processors along the i dimension.
(2) **Dependencies**: Each element C_{ij} can be computed in parallel, but each C_{ij} relies on A_{ik} and B_{kj}. Therefore any processor computing C_{ij} at time t must guarantee that A_{ik} and B_{kj} are resident on the same processor at time t.
(3) **Spatial Mapping**: On a 2D grid indexed by (i, j), we can simply assign the computation C_{ij} to processor (i, j). This identity mapping makes C stationary; all computations necessary for computing C_{ij} reside on processor (i, j). Alternatively, it is possible to map (i, k) spatially, making A stationary, or mapping (j, k) spatially, making B stationary

Such decisions regarding,dependencies, communication, and spatial mapping result in a large space of distributed programs (including triangular solve and matrix multiply) that compute the same result but with different dataflows. Figure 13 illustrates how algorithm's like Cannons, PUMMA, and weight-stationary systolic multiplication involve different dataflows but all fundamentally conduct a matrix multiply. They all share the same base recurrence $C_{ij} = \sum A_{ik} B_{kj}$, meaning this recurrence in itself is insufficient for uniquely describing each of these matrix multiplication algorithms.

The same lack of specificity applies to triangular solve and other recurrences. This warrants a more *specific* form of recurrence equations that define not only the computations but the involved dataflows. These specific recurrences can either serve as a unique input description of the distributed recurrence, or as an intermediate representation that result from feeding the original recurrences through a lowering algorithm. In the latter case, these more specific, intermediate recurrences can be generated by coupling the original recurrence with additional user-provided scheduling commands that describe data movement.

These more specific recurrences describe *iteration spaces with dataflow*, which we refer to as *dataflow recurrences*. These more specific *dataflow recurrences* specify not only computation but data movement across the iteration space. Figure 4 shows the dataflow recurrences that are shared by any form of Cannon's algorithm. In these recurrences, values are owned not by tensors, but by points in the iteration space. Cannon's algorithm, for example, is describable as a dataflow recurrence, in which A_{ik} and B_{kj} are streamed through nearest neighbors in the 3D iteration space. To illustrate how values of A_{ik} and B_{kj} are owned by different points in the iteration space, we use a notation $(ijk)_A$ to refer to the value of A (e.g., A_{ik}) that lives within point ijk in the iteration space. To illustrate how A_{ik} is streamed from the neighboring point (corresponding to the previous j), we use the following notation.

$$(ijk)_A = (i, j-1, k)_A.$$

This dataflow is valid because for a particular i and k, A_{ik} is shared/broadcasted across all values of j. In the output-stationary Cannon's algorithm, the most basic algorithm for a distributed matrix multiply, we instantiate a 2D grid of processors which are similarly indexed by both i and j. This indexing states that processor (i, j) will conduct and own the calculation of output value C_{ij}. This dataflow recurrence is amenable to describing the computation in a distributed setting, as it is easy to define a mapping between the 3D iteration space indexed by (i, j, k) and the 2D grid of processors indexed by (i, j).

Of course, the reference to $(i, j-1, k)_A$ cannot be defined at points where $j = 0$, as it breaches the border of the iteration space. Therefore, the value of $(i, 0, k)_A$ must be defined as a feeder that reads from tensors. This hence forms a base case for the recurrence, in which all other access to memory otherwise come from neighboring iteration points. Figure 4 shows the full set of dataflow recurrences needed to implement Cannon's algorithm, with the additional necessary specification that any two of the variables are mapped spatially (usually i and j), and the remaining variable is mapped temporally (usually k). Similarly, we can decide to index the processing grid by i and k, meaning processor (i, k) will own input element A_{ik}, such that A_{ik} is *stationary*, and does not move across PEs. When A_{ik} is stationary, output elements C_{ij} and input elements B_{kj} will *stream* through the grid of processing elements. Therefore, any two of the variables from (i, j, k) can be mapped spatially, resulting in different variants of Cannon's algorithm which have the same dataflow among the iteration space but different dataflows between the physical processing elements.

Alternatively, instead of streaming inputs like A_{ik} across the grid of processors, they can be broadcasted directly to each processor from a common source. In SUMMA's algorithm, A_{ik} and B_{kj} are broadcasted globally, in a single step, from a source to all points in the iteration space that require the data. Both algorithms are used in practice, including several hybrid algorithms of the two. In PUMMA's algorithm, for example, A_{ik} is broadcasted amongst nearest neighbors (as in Cannon's), but B_{kj} is broadcasted globally, as it is in SUMMA. Triangular solve similarly has a large space of distributed algorithms, as do any recurrence over a multidimensional indexed tensors.

Cyclotron implements these principles in a compiler infrastructure. It allows users to specify recurrences and a set of scheduling primitives describing dataflow, which finally get lowered to a distributed architecture of processing elements. These examples illustrate that even simple recurrences expand into rich spaces of dataflow choices when distributed across processors. Cyclotron's goal is to make those choices explicit, composable, and compilable.

3 Overview

Cyclotron is a compiler that lowers declarative recurrences into programs that execute on distributed architectures. A declarative recurrence is a global equation describing the entire computation,

$$
\begin{aligned}
(i,0,k)^{mul}_{A} &\leftarrow A_{ik} && \text{inject } A\\
(0,j,k)^{mul}_{B} &\leftarrow B_{kj} && \text{inject } B\\
(i,j,0)^{mul}_{C} &\leftarrow 0 && \text{init. accumulator}\\
(ijk)^{mul}_{A} &\leftarrow (i,j-1,k)^{mul}_{A} && \text{receive } A \text{ along } j\\
(ijk)^{mul}_{B} &\leftarrow (i-1,j,k)^{mul}_{B} && \text{receive } B \text{ along } i\\
(ijk)^{mul}_{C} &\leftarrow (ijk)^{mul}_{A} \cdot (ijk)^{mul}_{B} + (i,j,k-1)^{mul}_{C} && \text{multiply–accumulate}\\
(ijk)^{mul}_{A} &\rightarrow (i,j+1,k)^{mul}_{A} && \text{forward } A \text{ to next } j\\
(ijk)^{mul}_{B} &\rightarrow (i+1,j,k)^{mul}_{B} && \text{forward } B \text{ to next } i
\end{aligned}
$$

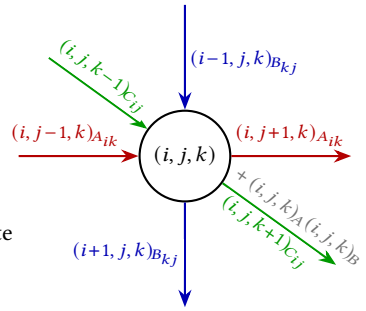


Fig. 4. Left: Cyclotron's IR for Cannon's algorithm style GEMM with communication and compute steps annotated. Each line specifies a logical dataflow recurrence over a multi-dimensional iteration space. In this IR, the *loop index tuple* (i,j,k) is the primary entity. The *tensor name* appears as a subscript, indicating which array or value is being updated, and the *iteration space name* appears as a superscript, indicating which recurrence or computation this update belongs to. Loads to memory (e.g., A_{ik}) are specified in tensor index notation, which occur at boundaries of the iteration space (e.g., $(i,0,k)$.) Right: Visualization of an internal point of the iteration space

whereas the output programs are local and unique to each processing element (PE) in the distributed system. Each output program consists of compute, send, and receive statements that together implement both computation and communication.

Central to Cyclotron is an intermediate representation (IR) of recurrences defined over the iteration space, rather than over tensors as in the input. This intermediate form expresses computation across both spatial and temporal dimensions, yielding an IR of dataflow recurrences that is portable across architectures. Because it abstracts only in terms of send, receive, and compute primitives, the IR can be theoretically lowered to any distributed PE architecture that implements these primitives. Cyclotron demonstrates this by lowering to two distinct targets: (1) a simulator of a dataflow processor composed of interconnected chiplets, and (2) a multinode CPU cluster. Figure 5 presents an overview of the compiler pipeline, from the declarative recurrence input through the lowering stages to the final backends.

The Cyclotron compiler accepts three user inputs. The first is a set of declarative recurrences defined over tensors. The second is a schedule that specifies which iteration dimensions are mapped to space and time, and how data is streamed across them. Together, these first two inputs describe both the mathematical computation and the dataflow of dependencies and broadcasted values.

Cyclotron compiles the recurrences and schedule into an intermediate representation (IR) of dataflow recurrences that expresses the computation over the program's iteration space rather than its tensors. This IR is architecture-agnostic. To generate executable programs for a concrete system, Cyclotron requires a third input: a specification of the target architecture and the topology of its processing elements (PEs). The intermediate IR, combined with this architectural specification, is then lowered into a final set of PE-local programs that implement the computation on the target distributed architecture.

Compilation consists of two distinct lowering passes. The first lowering pass takes in the input recurrences and schedule, and then determines what must be computed, sent and received (i.e., communicated) across the program's iteration space. The result of this lowering phase is the aforementioned architectural agnostic IR of *dataflow recurrences*. Here, data accesses come in two flavors: accesses to tensors, and accesses to neighboring points in the iteration space. Tensor reads are denoted by indexed tensor accesses, where the indices are subscripts to the tensor name

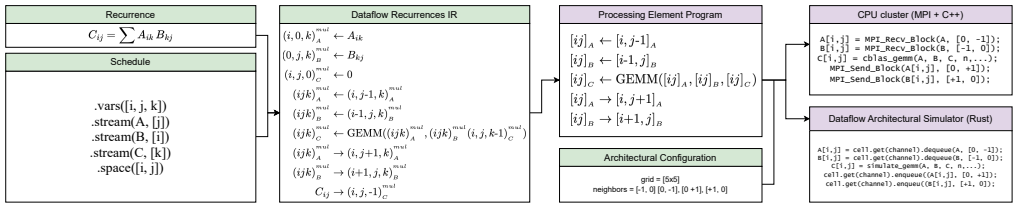


Fig. 5. Cyclotron Overview. The system diagram shows the compilation flow for a output-stationary CANNON-style dataflow. By mapping i and j are mapped across space, Cij is stationary, meaning it's locked resident onto each PE and does not move during the computation. k is mapped implicitly across time, such that A_{ik} streams systolically over j, and B_{kj} is stramed over i. The IR can be executed on a data-flow simulator of a chiplet style architecture or on an MPI-enabled cluster of processors.

(e.g., A_{ik}). IR statements that read from neighboring points in the iteration spaces are denoted by tensors accesses in which the tensor's name is a subscript to the current index (e.g., $(ijk)_A$) in the iteration space. This emphasizes that the value is actually owned by a point in the iteration spaces, rather than the tensor, meaning the value can be accessed via communication with the processing element's neighbors. Users can use scheduling commands that determine whether to *stream*, *broadcast*, or permanently assign (i.e., *prefetch*) data to processing elements.

The second lowering pass transforms the IR of dataflow recurrences into a set of PE specific programs. Here, users specify which dimensions will be mapped in space (in the diagram, i and j are mapped in space, and k in time). The final IR still uses neighbor access style reads, but in which the time dimension now manifests as a loop over a list of instructions, rather than inside the equations as an additional dimension. The tensor accesses in the output IR can thus be lower dimensional than in the input (e.g., $(ijk)_A$ vs $[ij]_A$, as the time dimension is now removed from the output IR's tensor accesses.

Cyclotron generates a separate program for each PE. Having PE-specific programs is necessary, as different PE's may work on different forms of the computation, may execute different kernels, and have different boundary conditions dictating what gets sent and received. Receives can be directly inferred from data dependencies in the input. Sends are induced via the compiler, which determines where to insert send statements baed on the placement of the receive statements. For any specific receive operation on a PE's IR, Cyclotron staticaly determines which other PE launches the corresponding send, and then appends a send operation on that other PE's IR.

During execution, each processing element interprets each instruction, representing a *megakernel* architecture. One persistent kernel (i.e., function) loops through and executes each instruction. These send, receive, and compute statements are executed in the order they appear on the PE-specific IR. Cyclotron ensures that each instruction in the IR has a corresponding operation in the current backend, whether it be a dataflow simulator that simulates a grid of chiplets, or a cluster of processors. Each target architecture therefore has an ISA, and Cyclotron includes a one-to-one mapping from IR statements to instructions in the architecture's ISA. This ensures the same executable binary can be run on both targets. This strategically allows for easy porting to different architectures, in which each architecture requires a light runtime (few hundred lines of code) in which each processing element reads in, decodes, and executes each instruction.

4 Dataflow Recurrences

As stated, normal recurrences (of which tensor algebra is a subset) express what must be calculated but do not delineate the data movement and IO necessary to implement the recurrence on a spatial

architecture. *Dataflow Recurrences*, which form Cyclotron's intermediate representation, do not have this limitation, as they make communication explicit. Dataflow recurrences, which are defined on top of iteration spaces, have several differences with and features on top of normal recurrences, which are defined on top of tensors.

(1) **Iteration Spaces Owning Data:** Data can be own either by tensors or points in the iteration space, whereas in normal recurrences all data is held within tensors.
(2) **Explicit Reduction Dimensions:** The dimensionality of all collections is the dimensionality of the enclosing iteration space, not the dimensionality of the underlying tensor.
(3) **Explicit Communication and Loads into Iteration Space:** Dataflow recurrences define how data is loaded from tensors and onto the iteration space.
(4) **Multiple Iteration Spaces:** A computation may involve several iteration spaces, which interact through explicit interfaces.
(5) **Composability with Space-time Mappings** The dataflow recurrences can be lowered to per-processing-element programs with a simple mapping of each variable into space or time.

Consider the canonical matrix multiplication in tensor algebra:

$$C_{ij} = \sum_k A_{ik}\, B_{kj}.$$

We use this example to illustrate how to transform a *recurrence over tensors* into a *recurrence over iteration spaces*. The equation above is purely *declarative*: it specifies what value each C_{ij} must hold, but not how to schedule the computation or where data must be moved. To derive an executable schedule, we lower this expression into dataflow recurrences, handling all of the aforementioned features.

4.1 Iteration Spaces Owning Data

In normal recurrences, all data is owned by tensors. To illustrate, in matrix multiply, the output values are specified as C_{ij} in which the tensor C is the primary entity, and in which indices appear as a subscript. While dataflow recurrences allow data to be owned by tensors, they importantly also allow data to be owned by the iteration space itself. To denote this, dataflow recurrences use an *indexing first*-notation, in which the loop index tuple (i, j, k) is the primary entity. In this notation, the matrix multiply outputs (including partial outputs) are denoted as $(i, j, k)_C$. A tensor name appears as a subscript, indicating which array or value is being updated, while an iteration-space or recurrence name may appear as a superscript to distinguish different computations. This definition thus places the computation at a precise point (i, j, k) in a 3D iteration space. This notation allows for a natural expression of communication and IO.

4.2 Explicit Reduction Dimensions

In the declarative form, the output tensor C_{ij} is indexed by two variables, i and j. The reduction index k appears in the expression but does not uniquely identify any output element. In a recurrence over iteration spaces, however, all three iteration variables—i, j, and k—must appear in every reference to C, because C is computed inside a triply nested loop. This follows naturally from the dataflow recurrences being defined over the iteration space instead of tensors. We therefore introduce k as an explicit loop index, rewriting the summation as an accumulation over k:

$$(i, j, k)_C \leftarrow (i, j, k-1)_C + A_{ik} \cdot B_{kj}, \qquad (i, j, 0)_C \leftarrow 0.$$

The reduction manifests as a data dependency between spatially adjacent points along the k axis: (i, j, k) reads from $(i, j, k-1)$. This is a key advantage of iteration-space recurrences: all dependencies map to *neighboring points* in the iteration space, rather than to arbitrary tensor accesses. While explicit tensor accesses are still permissible in Cyclotron, we want all data access to be accesses to neighboring data points as opposed to relatively expensive tensor accesses.

4.3 Explicit Communication and Loads into Iteration Space

Each tensor element A_{ik} is now treated as a value injected into the iteration space at $j = 0$, giving it an explicit spatial coordinate:

$$(i, 0, k)_A \leftarrow A_{ik}.$$

This distinguishes the *source of the value* from its *current location* in the iteration space through which it flows. To make A_{ik} available at every j where C_{ij} is updated, we introduce a propagation rule along the j axis:

$$(ijk)_A \leftarrow (i, j-1, k)_A,$$

meaning that the copy of A_{ik} needed at (i, j, k) is obtained by forwarding it from the previous j tile. Syntactically, we notice that in the original recurrence the variable j is missing from the tensor access A_{ik}. In other words, for any particular i and k, and for all j, the same value of A_{ik} is broadcasted across the j dimension to all points (i, j, k). Therefore the value A_{ik} must be communicated and propagated along the j dimension of the iteration space.

Symmetrically, B_{kj} is injected at $i = 0$ and propagated along i:

$$(ijk)_B \leftarrow (i-1, j, k)_B.$$

Resulting Space–Time Specification. The result is a set of recurrences that jointly specify compute and communication:

$$\begin{aligned}
(i, 0, k)_A &\leftarrow A_{ik} & (ijk)_A &\leftarrow (i, j-1, k)_A \\
(0, j, k)_B &\leftarrow B_{kj} & (ijk)_B &\leftarrow (i-1, j, k)_B \\
(i, j, 0)_C &\leftarrow 0 & (ijk)_C &\leftarrow (i, j, k-1)_C + (ijk)_A \cdot (ijk)_B.
\end{aligned}$$

Notably all communication is with nearest neighbors, with any offsets to index variables being –1. This formulation makes the dataflow explicit. Output-stationary Cannon's algorithm implements these dataflow recurrences such that i and j are mapped across space, and k across time. This is visualized by Figure 13 . Here, A tiles stream rightward across j, B tiles stream downward across i, and each (i, j) processing element accumulates C_{ij} over k. The final write to the result tensor C can then be optionally specified with the recurrence $C_{ij} \leftarrow (i, j, k_{end})_C$, which represents exporting data from the iteration space to memory.

4.4 Multiple Iteration Spaces

Many computations require more than one iteration space to express their structure and data dependencies. In Cyclotron, each distinct operation in the recurrence's abstract syntax tree—such as `GEMM`, `TRSM`, or `SQRT`—is assigned its own iteration space, defining when and where that operation occurs in space–time. A simple matrix multiplication involves only one such operation and therefore a single iteration space. In contrast, composite computations like block triangular solve (`TRSM`) involve multiple sub-operations, each with its own spatial and temporal pattern.

$$\textbf{Spaces:}\quad \mathcal{D}_{\text{tri}} = \{(i,j) \mid 0 \le j \le i < n\}, \qquad \mathcal{D}_{\text{diag}} = \{(ii,ii) \mid 0 \le ii < n\}.$$

Initialization / loads:

$$(i,j)^{\text{tri}}_{L} \leftarrow L_{ij} \qquad \text{stream tiles of } L$$

$$(ii,ii)^{\text{diag}}_{L} \leftarrow L_{ii,ii}, \quad (ii,ii)^{\text{diag}}_{B} \leftarrow B_{ii} \qquad \text{diag loads}$$

Boundary handshakes (tri ↔ diag):

$$(ii,ii)^{\text{diag}}_{X_a} \leftarrow (ii, ii-1)^{\text{tri}}_{X_a} \qquad \text{TRSM reads end-of-row from tri}$$

$$(i,j_e)^{\text{tri}}_{X_b} \leftarrow (i-1,j)^{\text{diag}}_{X} \qquad \text{optional: seed broadcast at } j_e$$

Intra-space data movement:

$$(i,j)^{\text{tri}}_{X_b} \leftarrow (i-1,j)^{\text{tri}}_{X_b} \qquad \text{vertical broadcast in column } j$$

$$(i,j)^{\text{tri}}_{X_a} \rightarrow (i,j+1)^{\text{tri}}_{X_a} \qquad \text{forward accumulator along } j$$

Computation (tri space: GEMM)

$$(i,j)^{\text{tri}}_{X_a} \leftarrow \text{GEMM}\Big((i,j)^{\text{tri}}_{L},\ (i,j)^{\text{tri}}_{X_b}\Big) \qquad \text{for } i > j,$$

Computation (diag space: TRSM):

$$(ii,ii)^{\text{diag}}_{X} \leftarrow \text{TRSM}\Big((ii,ii)^{\text{diag}}_{L},\ (ii,ii)^{diag}_{B} - (ii,ii)^{\text{diag}}_{X_a}\Big)$$

Publish (optional):

$$X_{ii} \leftarrow (ii,ii)^{\text{diag}}_{X}.$$

Fig. 6. Dataflow recurrences i.e. IR for block lower-triangular solve $LX = B$ with two iteration spaces: triangular space (streamed GEMM) and diagonal space (TRSM).

$$X_i \;=\; \text{TRSM}\Big(L_{ii},\ B_i - \sum_{j<i} \text{GEMM}\big(L_{ij},\ X_j\big)\Big)$$

```
X_i = B_i
for j < i:
    X_i -= GEMM(L_ij, X_j)
X_i = TRSM(L_ii, X_i)
```

Figures 7 and 8 visualize the dataflow of a tiled lower-triangular solve. Figure 6 shows the corresponding dataflow recurrences. The computation decomposes into two coupled spaces. There's a *triangular space* performing streamed GEMM updates, and a *diagonal space* performing smaller, recursive TRSM solves.

Each space carries its own recurrences, loads, and local communications. The triangular space accumulates updates along rows and columns, while the diagonal space periodically consumes a boundary of the triangular space to perform a smaller triangular solve. Once a diagonal solve completes, its result is broadcast back into the triangular space to seed the next wave of updates. This interaction is captured by explicit *interface maps* between the iteration spaces, denoted Φ_1 and Φ_2, which define how coordinates and data are translated between their domains.

Interface maps:

$$\Phi_1 : \mathcal{D}_{\text{diag}} \;\rightarrow\; \mathcal{D}^{\rightarrow}_{\text{tri}}, \qquad \Phi_1(ii,ii) = (i{=}ii,\ j{=}j_e{+}1),$$

$$\Phi_2 : \text{diag}(\mathcal{D}_{\text{tri}}) \;\rightarrow\; \mathcal{D}^{\searrow}_{\text{tri}}, \qquad \Phi_2(j,j) = (j{+}1,\ j).$$

The resulting dataflow recurrences (Figure 6) show two iteration spaces operating concurrently but exchanging data through explicit, one-to-one interfaces. Each space defines its own local schedule, while the interface maps Φ_1 and Φ_2 synchronize their boundaries in both time and index coordinates. Together they form a unified spacetime representation of the block triangular solve: every transfer, update, and synchronization is made explicit in the IR, allowing Cyclotron to reason about both computation and communication at the same level of abstraction.

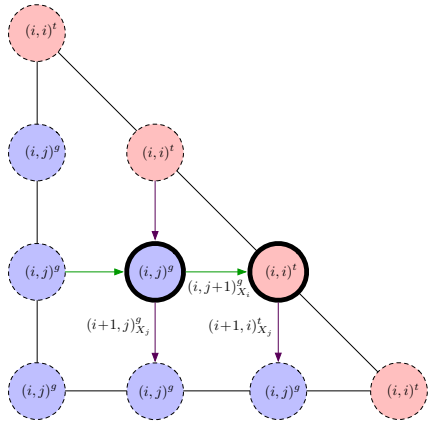


Fig. 7. Interlinked iteration spaces in the triangular solve, with sends labeled for the two bold nodes. The diagonal spa ce denoted by (i, i) points conduct local triangular solves. The lower triangular space denoted by (i, j) points conduct local GEMMs. The two spaces interact at the boundary. The two bold points(1 gemm point, and 1 trisolve point) are also illustrated in Figure 8

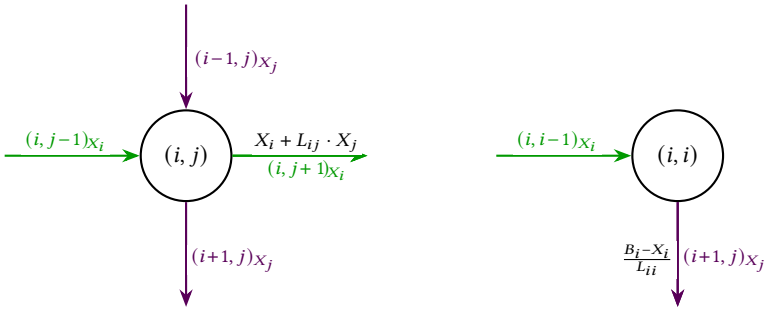


Fig. 8. Dataflow of each operation in the triangular solve. Left: dataflow of the GEMM within the triangular solve. Right: dataflow of the smaller triangular solve within the triangular solve. The triangular solve node ingests outputs of the GEMMS, and then produces values that seed future GEMMs

Figure 6 shows the dataflow recurrences for these two iteration spaces, one for the GEMM part of the computation, and one for the triangular solve (TRSM) part of the computation. Each iteration space has it's own dataflow, with the internal GEMM computation receiving data other GEMM computations. The boundary GEMMS require special handling; initial GEMMS receive their data from TRSM, and terminal GEMMs send their partial outputs to TRSM. This is made explicit by the **boundary-handshakes** in Figure 6, in which the two spaces exchange values via the interface maps. The exchange can also bee seen in the interface between the diagonal (i, i) nodes and triangular (i, j) nodes in Figure 7 , showing the full triangular iteration space.

5 Scheduling

Cyclotron provides a few scheduling commands to the user: *stream*, *prefetch*, and *broadcast*. We show how all three can be used in the context of a tiled triangular solve, resulting in systolic arrays with different performance characteristics. Here, the equation for the tiled triangular solve is:

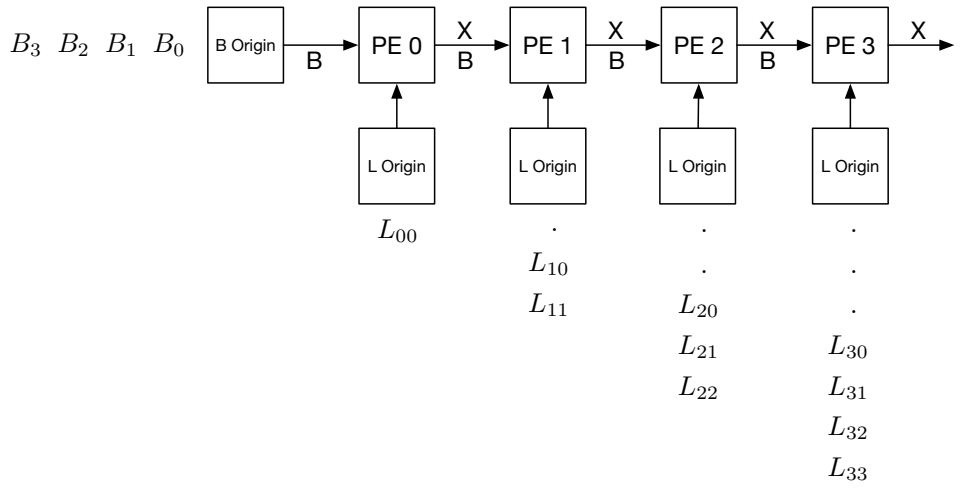


Fig. 9. Distributed triangular solve across processing elements.Each PE reads in one block row of L and the corresponding partition of B. Diagonal blocks L_{ii} perform local solves, while off-diagonal L_{ij} blocks perform matrix-vector multiplies and propagate partial results x_j to subsequent PEs. Arrows indicate the flow of x and b values and the sequential dependencies implied by the recurrence.

$$x_i \;=\; \mathrm{TRISOLVE}\Big(L_{ii},\; b_i \;-\; \sum_{j<i} \mathrm{GEMV}\big(L_{ij},\; x_j\big)\Big)$$

The triangular solve can be scheduled in different ways, such that the user can configure where data for B or L is stationary or streamed. If B is *streamed*, the ith PE consumes the ith portion of B as it arrives, performing local triangular solves using its resident L_{ii} block before forwarding the newly computed X_i downstream. In this configuration (Figure 9) L is also streamed, and each PE consumes portions of L_{ij} as they arrive, performing a matrix vector multiply between tile L_{ij} and vector slice x_j and then forwarding x_j downstream to other PEs where it will be needed. This pattern is expressed in Cyclotron using the `stream` directive:

```
stream(B, [i])
```

Figure 9 shows the 1D PE grid implementing a triangular solve, where L, B and X are all streamed. Alternatively, as i is mapped across space, we can presave all input B_is across PEs in the i dimension, such that B becomes *stationary*: each PE holds its B_i segment locally and consumes incoming blocks of L_{ij} as they arrive. This pattern becomes useful when reusing B across different L matrices, and can be expressed with the `prefetch` directive:

```
prefetch(B, [i])
```

Figure 10 shows a 1D PE grid created by scheduling command prefetch(B, [i]), meaning B_i is now stationary, as opposed to streamed into the grid as it was in Figure 9/ Finally, if there is a broadcast network for B, such that every PE has a connection to the source of B (i.e., the *feeder*), then each PE can receive B directly from the feeder. This is analogous to the broadcasts seen in the PUMMA (and SUMMA) algorithm; the matrix is broadcasted directly from the feeder, rather then streamed through PEs. Figure 11 shows a 1D PE grid in which B is broadcasted individually to all processors. This can be accomplished with the `broadcast` directive:

```
broadcast(B, [i])
```

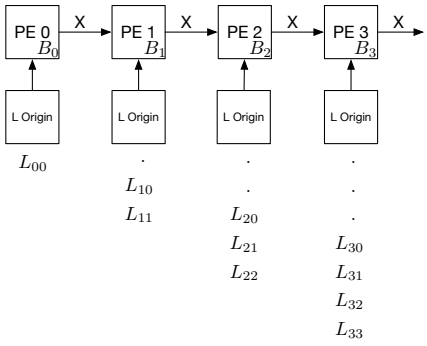


Fig. 10. prefetch B

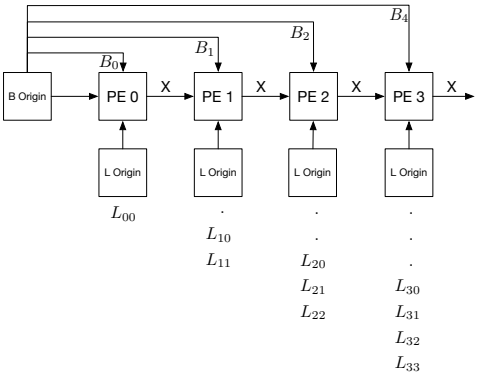


Fig. 11. broadcast B

Fig. 12. Two systolic arrays of a blocked triangular solve. Left: B is stationary, and L is streamed. Right, L is streamed and B is broadcasted.In both cases, L_{ij} is skewed by it's column offset j in order to ensure it arrives at PE i at the same time as X_j

These three schedules are simply different implementations of the same base recurrence for a tiled triangular solve. In practice, the optimal choice between them depends on the existence of a broadcast network, whether B or L can be reused, and the bandwidth of all of the links.

6 Space-time Mappings of Dataflow Recurrences

The recurrences above define dataflow over a three-dimensional *logical* iteration space, not over any concrete hardware. Given, for example, a two-dimensional array of physical processors, any two iteration dimensions may be mapped to *space*, with the remaining one implicitly mapped to *time*. Intuitively, spatial dimensions correspond to parallelism across processors, while temporal dimensions correspond to values that are accumulated locally over time. Choosing which dimensions to assign to space versus time is a user-specified scheduling decision, and different mappings are advantageous in different settings.

If we choose to map i and j in space, then k is implicitly mapped to time. Communication along the i or j axes therefore manifests as physical communication across the processor array, whereas communication along the k axis simply corresponds to reading a value from a previous time step. Folding k into time yields the following program. This program represents the dataflow of all PEs, using constant indices (e.g. $[i, 0]$) to indicate special handling of boundary PEs:

$$
\begin{aligned}
&[i,j]_C \leftarrow 0 \\
&\textbf{for } k = 1, 2, \ldots \\
&\qquad [i,0]_A \leftarrow A_{ik} \\
&\qquad [0,j]_B \leftarrow B_{kj} \\
&\qquad [i,j]_A \leftarrow [i,j-1]_A \\
&\qquad [i,j]_B \leftarrow [i-1,j]_B \\
&\qquad [i,j]_C \leftarrow [i,j]_C + [i,j]_A \cdot [i,j]_B \\
&\qquad [i,j]_A \rightarrow [i,j+1]_A \\
&\qquad [i,j]_B \rightarrow [i+1,j]_B
\end{aligned}
$$

In this schedule, A and B stream across hardware links, while C remains *stationary*. However, the roles can be inverted. In scenarios where A is much larger than B and C—as in neural networks where A contains the weights— it is preferable to keep A resident and instead move the activations. This is achieved by mapping i and k in space, and j in time, yielding the well-known *weight-stationary* schedule from neural network accelerators, where partial outputs of C traverse the systolic array at each timestep:

$$
\begin{aligned}
&[i,k]_W \leftarrow A_{ik} \\
&\textbf{for } j = 1, 2, \ldots \\
&\qquad [0,k]_B \leftarrow B_{kj} && \text{(inject at } i{=}0 \text{ i.e., north boundary)} \\
&\qquad [i,0]_C \leftarrow 0 && \text{(init. accumulator at west boundary)} \\
&\qquad [i,k]_B \leftarrow [i-1,k]_B && \text{(receive } B \text{ from north, non-boundary case)} \\
&\qquad [i,k]_C \leftarrow [i,k-1]_C \ + \ [i,k]_W \cdot [i,k]_B && \text{(receive } C \text{ and perform GEMM)} \\
&\qquad [i,k]_C \rightarrow [i,k+1]_C && \text{(forward } C \text{ east along } k\text{, non-boundary case)} \\
&\qquad [i,k]_B \rightarrow [i+1,k]_C && \text{(forward } B\text{)}
\end{aligned}
$$

Lowering from the dataflow recurrences to per-PE programs proceeds by a simple *pattern matching* of each point in the processor grid against the iteration-space templates defined by the recurrences. Each recurrence specifies a dataflow rule over a 3D iteration space (i, j, k); during lowering, Cyclotron instantiates these rules for each processor coordinate, selecting only those whose index patterns mostly closely match the processor's spatial location and boundary conditions.

In the *weight-stationary* variant of Cannon's algorithm, (i, k) are mapped spatially and j advances in time. A processor at cell $(i, k = 0)$ matches the rule $(i, j, k = 0)^{mul}_{C} \leftarrow 0$, which initializes the local partial sum C_{ij} along the west boundary. Likewise, processors at $(i = 0, k)$ on the northern boundary bind to the inject rule $(i = 0, j, k)^{mul}_{B} \leftarrow B_{kj}$ rather than the streaming form $(ijk)^{mul}_{B} \leftarrow (i-1, j, k)^{mul}_{B}$.

This *template-matching* process produces one localized program per cell type (corner, edge, or interior). When the temporal dimension j is folded into an outer loop, the resulting programs correspond to the per-PE kernels shown in Figure 14.

Conceptually, this lowering stage replaces dataflow recurrences with concrete per-processor programs obtained by instantiating only the rules that match each cell's coordinates.

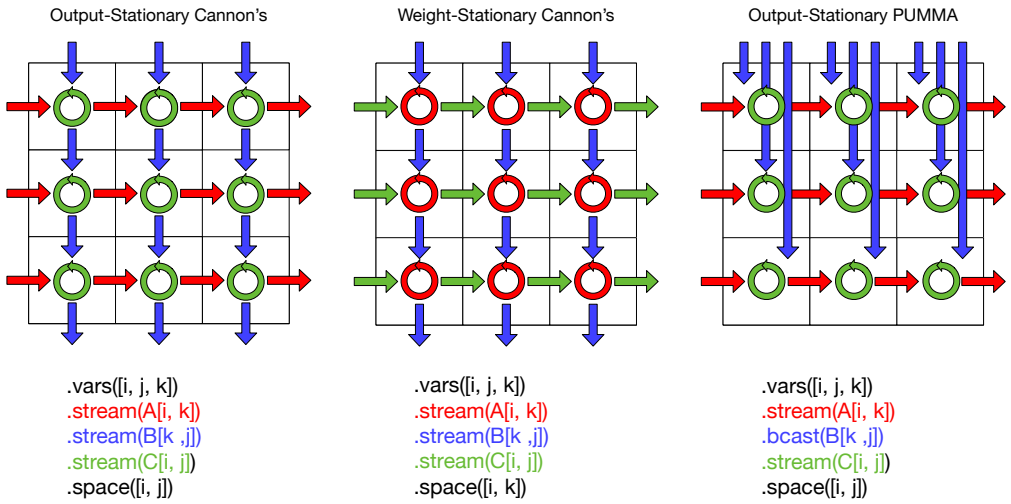


Fig. 13. Dataflow of various matrix multiply algorithms. Each algorithm implements the recurrences $C_{ij} = A_{ik}B_{kj}$, but with different dataflow and communication. Left, output stationary Cannon's algorithm, in which A and B both stream systolically across the grid of processing elements. Middle: Weight (e.g., A_{ik} stationary Cannon's algorithm, in which B and C are streamed in. Right: Output stationary PUMMA algorithm. This algorithm broadcasts B down the i dimension, and systolically streams A in the j direction.

7 Timing

For any systolic execution, correctness requires that all operands of each recurrence—such as A_{ik} and B_{kj} in matrix multiplication—arrive at the same processing element (PE) and at the same time step. Misalignment would cause a stall or the use of stale data. Intuitively, the schedule defines a space–time wavefront that sweeps through the array: each PE on that wavefront receives exactly the operands it needs from orthogonal directions at the same clock tick.

The relative timing between operands arises from a *skew* in the schedule.When one loop index is mapped to time, values associated with different spatial coordinates are emitted at staggered offsets so that their trajectories through space intersect at the correct moment. In other words, temporal skew encodes how far a value must travel before being consumed. In matrix multiplication, for instance, A_{ik} and B_{kj} are launched at different phases: A_{ik} is delayed by its horizontal offset i, while B_{kj} is delayed by its vertical offset j. These coordinated delays ensure that both operands reach $\mathrm{PE}(i, j)$ simultaneously, inducing the characteristic diagonal wavefront of systolic execution.

This alignment can be expressed geometrically. Mapping any two indices to space and the remaining one to time induces a global execution time

$$t = i + j + k,$$

which represents the *Manhattan distance* from the origin of the iteration space to point (i, j, k) under unit communication latency. Each operand's coordinates determine its emission time and propagation delay so that all dependencies for (i, j, k) meet at $\mathrm{PE}(i, j)$ precisely at $t = i+j+k$. Cyclotron verifies this invariant algebraically when generating distributed schedules.

The same reasoning extends to dependent recurrences such as triangular solve. In Figure 9, i is mapped in space, j in time, and tensors L, X, and B are streamed into the array. Operand L_{ij} originates with an offset of i and advances for j steps before reaching PE i, arriving at time $i+j$.

Cell $(0,0)$. $[i,k]_W \leftarrow A_{ik}$ for j $[i,k]_B \leftarrow B_{kj}$ $[i,k]_C \leftarrow 0 + [i,k]_W \cdot [i,k]_B$ $[i,k]_B \rightarrow [i+1,k]_B$ $[i,k]_C \rightarrow [i,k+1]_C$	*Cell* $(0,k)$. $[i,k]_W \leftarrow A_{ik}$ for j $[i,k]_B \leftarrow B_{kj}$ $[i,k]_C \leftarrow [i,k-1]_C + [i,k]_W \cdot [i,k]_B$ $[i,k]_B \rightarrow [i+1,k]_B$ $[i,k]_C \rightarrow [i,k+1]_C$	*Cell* $(0,k_{end})$. $[i,k]_W \leftarrow A_{ik}$ for j $[i,k]_B \leftarrow B_{kj}$ $[i,k]_C \leftarrow [i,k-1]_C + [i,k]_W \cdot [i,k]_B$ $[i,k]_B \rightarrow [i+1,k]_B$ $C_{ij} \leftarrow [i,k]_C$
Cell $(i,0)$. $[i,k]_W \leftarrow A_{ik}$ for j $[i,k]_B \leftarrow [i-1,k]_B$ $[i,k]_C \leftarrow 0 + [i,k]_W \cdot [i,k]_B$ $[i,k]_B \rightarrow [i+1,k]_B$ $C_{ij} \leftarrow [i,k]_C$	*Cell* (i,k). $[i,k]_W \leftarrow A_{ik}$ for j $[i,k]_B \leftarrow [i-1,k]_B$ $[i,k]_C \leftarrow [i,k-1]_C + [i,k]_W \cdot [i,k]_B$ $[i,k]_B \rightarrow [i+1,k]_B$ $[i,k]_C \rightarrow [i,k+1]_C$	*Cell* (i,k_{end}). $[i,k]_W \leftarrow A_{ik}$ for j $[i,k]_B \leftarrow [i-1,k]_B$ $[i,k]_C \leftarrow [i,k-1]_C + [i,k]_W \cdot [i,k]_B$ $[i,k]_B \rightarrow [i+1,k]_B$ $[C_{ij} \leftarrow [i,k]_C$
Cell $(i_{end},0)$. $[i,k]_W \leftarrow A_{ik}$ for j $[i,k]_B \leftarrow [i-1,k]_B$ $[i,k]_C \leftarrow 0 + [i,k]_W \cdot [i,k]_B$ $[i,k]_C \rightarrow [i,k+1]_C$	*Cell* (i_{end},k). $[i,k]_W \leftarrow A_{ik}$ for j $[i,k]_B \leftarrow [i-1,k]_B$ $[i,k]_C \leftarrow [i,k-1]_C + [i,k]_W \cdot [i,k]_B$ $[i,k]_C \rightarrow [i,k+1]_C$	*Cell* (i_{end},k_{end}). $[i,k]_W \leftarrow A_{ik}$ for j $[i,k]_B \leftarrow [i-1,k]_B$ $[i,k]_C \leftarrow [i,k-1]_C + [i,k]_W \cdot [i,k]_B$ $C_{ij} \leftarrow [i,k]_C$

Fig. 14. Localized per-cell programs for a weight-stationary Cannon's , showing both the internal PE and all eight boundary conditions. Each boxed cell represents the computation executed by a processing element (PE) at spatial coordinates (i,k) over time coordinate k . Red statements denote loads of A_{ik} into the edge of the systolic array, whereas blue statements denote loads of B_{kj} into the systolic array, and green statements denote write of C_{ij} among the output edge. For a matrix multiply, cyclotron will emit these 9 types of programs, with programs duplicated across PEs that play the same role (e.g., all internal cells will have the same program denoted by Cell (i,j), as they do not require I/O with memory.

Meanwhile, X_j is produced at time $2j$ (after j internal reductions) and then traverses $i-j$ PEs to reach PE i at the same time $i+j$. Thus both L_{ij} and X_j arrive concurrently to perform $L_{ij}\,X_j$.

This unified geometric interpretation—where communication latency corresponds to Manhattan distance in the iteration space—ensures that all operands intersect at the correct PE and time step.

8 Lowering Recurrences to Dataflow Recurrences

The input recurrences are lowered to dataflow recurrences via a set of rewrite rules, which dictate how to transform accesses to indexed tensors into the streaming dataflow. In the matrix multiply example, our rewrite rules rewrite the tensor access A_{ik} into streaming dataflow recurrences, which include $(i,j,k)_A \leftarrow (i,j-1,k)$ and the boundary case $(i,j,k)_A \leftarrow A_{ik}$

Consider output indices $\vec{i}$, reduction index r, output tensor C, and input operands $X^{(t)}$ with index maps $g_t(\vec{i},r)$ (i.e., it maps $(ijk)_A$ to ik) Each operand chooses a stream axis $s_t \in \vec{i}$. Finally, let $r_{\text{end}}(\vec{i})$ be the exclusive upper bound of reduction variable r, and let $r_{\text{final}}(\vec{i})$ be it's inclusive upper bound.

$$C_{o(\vec{i})} \;=\; \mathcal{E}_{\vec{i}}\left[\sum_{\vec{r}} F\big(X^{(1)}_{g_1(\vec{i},\vec{r})},\ldots,X^{(m)}_{g_m(\vec{i},\vec{r})},\; C_{g_C(\vec{i},\vec{r})}\ \text{(optional)}\big)\right].$$

$$\frac{s_t \in \vec{i} \qquad s_t \notin \text{indices}(X^{(t)})}{\text{stream}(X^{(t)}, s_t)} \textsc{ChooseStream}$$

$$\frac{}{\begin{array}{l}(\vec{i}, \mathbf{0})_{\widetilde{C}} \leftarrow 0 \qquad (\vec{i}, \vec{r})_{\widetilde{C}} \leftarrow (\vec{i}, \vec{r}-1)_{\widetilde{C}} \; + \; F\big(X^{(1)}_{g_1(\vec{i},\vec{r})}, \ldots, X^{(m)}_{g_m(\vec{i},\vec{r})}, \; C^{(\text{opt.})}_{h(\vec{i},\vec{r})}\big) \\ (\text{If } \vec{r} = \emptyset, \text{ interpret } (\vec{i}, \mathbf{0})_{\widetilde{C}} := F(\ldots).)\end{array}} \textsc{LowerSum}$$

$$\frac{\text{stream}(X^{(t)}, s_t) \qquad \mathbf{p}_t = g_t(\vec{i}, \vec{r})}{(\vec{i}[s_t \mapsto 0], \vec{r})_{X^{(t)}} \leftarrow X^{(t)}_{\mathbf{p}_t}} \textsc{Inject} \qquad \frac{\text{stream}(X^{(t)}, s_t)}{(\vec{i}, \vec{r})_{X^{(t)}} \leftarrow (\vec{i}[s_t \mapsto s_t - 1], \vec{r})_{X^{(t)}}} \textsc{Propagate}$$

$$\frac{\text{stream}(X^{(t)}, s_t)}{(\vec{i}, \vec{r})_{X^{(t)}} \rightarrow (\vec{i}[s_t \mapsto s_t + 1], \vec{r})_{X^{(t)}}} \textsc{Send}$$

$$\frac{\mathcal{E}_{\vec{i}}[\cdot] \neq [\cdot] \quad \text{i.e. summation has outer expression}}{\begin{array}{l}(\vec{i}, \vec{r}_{\text{final}}(\vec{i}))^{inner}_{\widetilde{C}} \rightarrow (\vec{i}, \vec{r}_{\text{end}}(\vec{i}))^{outer}_{\widetilde{C}} \quad (\vec{i}, \vec{r}_{\text{end}}(\vec{i}))^{outer}_{\widetilde{C}} \leftarrow (\vec{i}, \vec{r}_{\text{final}}(\vec{i}))^{inner}_{\widetilde{C}} \\ (\vec{i}, \vec{r}_{\text{end}}(\vec{i}))^{outer}_{C} \leftarrow \mathcal{E}_{\vec{i}}\big[\, (\vec{i}, \vec{r}_{\text{end}}(\vec{i}))^{outer}_{\widetilde{C}} \,\big]\end{array}} \textsc{OuterSend}$$

$$\frac{C \text{ used in } F \text{ and } \; s_t \in \vec{i} \qquad \text{s.t. } \; r_{\text{end}}(\vec{i}) = s_t}{(\vec{i}, \; r_{\text{final}}(\vec{i}))^{inner}_{C} \; \leftarrow \; \big(\vec{i}[s_t \mapsto s_t - 1], \; r\big)^{outer}_{C}} \textsc{InnerSend}$$

The transformation proceeds by a sequence of syntactic rewrites, shown formally in the inference rules. The key idea is to replace the implicit summation with a running accumulator over the reduction index, and to materialize the flow of each operand as a stream through the output iteration space.

The first five rules are the *core* rules that define how to handle lowering a *stream()* command within the single iteration space. These rules (which do not include the inner send and outer) are shared between recurrences with and and without dependencies. They define how data moves within a single iteration space. The outer/inner send rules are novel, and show how communication happens between iteration spaces.

Of these rules, (1) **LowerSum** is one of the two top-level entry rules. It replaces the high-level summation with an explicit accumulator variable $\widetilde{C}_{(\vec{i},r)}$ and initializes it to zero, producing a recurrence that iterates stepwise over r. For the matrix multiply example, this rule identifies that there is a summation over k necessary to calculate $C_{ij} = \sum_k A_{ik} \cdot B_{kj}$, and then emits the recurrence $\widetilde{C}_{ij} \leftarrow (i, j, k-1)_{\widetilde{C}} + A_{ik} \cdot B_{kj}$

(2) **ChooseStream** is the other top level rule, being an entry point to compilation. It chooses a stream axis $s_t \in \vec{i}$ such that $s_t \notin \text{indices}(X^{(t)})$. This ensures that a single injected value can be propagated along s_t and reused across all dependent lattice points, thus establishing the direction of its propagation within the iteration space. (3) **Inject** introduces the base of each stream: at the start of the chosen stream axis, we inject the operand's source value $X^{(t)}_{g_t(\vec{i},r)}$ into the iteration space. For the matrix multiply, this rule would identify j as the stream for argument A_{ik}, as j is not included in $A'_{ik}s$ indexing expression. It would similarly choose i as the stream for argument B_{kj}. (4) **Propagate** defines how that value is received at subsequent lattice points, typically by shifting one coordinate:

$$(\vec{i}, r)\, X^{(t)} \;\leftarrow\; (\vec{i}[s_t \mapsto s_t - 1], r)\, X^{(t)}.$$

For the matrix multiply, this rule converts the input term A_{ik} into $(i, j, k)_A \leftarrow (i, j{-}1, k)_A$, and then applies a similar transformation to B_{kj}. (5) **Send** expresses the dual forward movement, being the other side of propagate.

The outer send rule handle cases where there are non-distributive outer expressions around a summation. The inner send rule handles recurrences with dependencies, and dictates how an output gets reused as an input for further computations. For a triangular solve, the presence of an outer expression and output dependencies results in the boundary-handshake equations seen in Figure 6. The triangular solve contains two operations, and thus two iteration spaces; one for the GEMM, and one for the smaller triangular solve, which can be visualized with nodes representing the dataflow in Figure 8

These rewrite rules show how a *stream*() scheduling command gets lowered. The *broadcast()* command follows a slightly modified version of these rewrite rules; all streaming rules stay the same, except the propagate rule becomes irrelevant, and all inject/send instructions are from a source/feeder cell at the boundary of the PE grid, instead of from a neighboring processor.

Conceptually, these rewrites make the spatial temporal structure of each recurrence explicit. The resulting program operates not as a pure reduction but as a wavefront of local updates, each consuming values streamed from its neighbors and producing results for subsequent iterations.

9 Compilation Targets

Cyclotron compiles its input recurrences into a set of per-PE programs, each specifying the local computation and communication behavior of an individual processing element. These programs can be executed on one of two backend targets: a cycle-accurate simulator that models a lattice of chiplets on a single die, or a multinode backend that employs MPI for inter-PE communication across distributed systems. Adding a new target is straightforward: each backend implements a lightweight runtime in which every PE runs an interpreter that iterates through its instruction list and executes operations in order. The runtime need only provide primitives for sending and receiving messages, along with implementations of the arithmetic kernels used by the program. Both existing backends follow this model and remain compact—each under roughly a thousand lines of code—making it easy to extend Cyclotron to new architectures or communication substrates. This modular design decouples code generation from execution, allowing the same compiled program to run unmodified on both simulated and distributed environments.

9.1 Simulator Target

Cyclotron builds on the Dataflow Abstract Machine (DAM) [Zhang et al. 2024], a cycle-accurate runtime for spatial architectures that models each PE as an independent *context* executing its own instruction stream while communicating over bounded channels. DAM follows Communicating Sequential Processes with Time (CSPT): each context advances a local clock, exchanges timestamped messages, and synchronizes through lightweight atomic or futex-based primitives rather than a global event queue. This distributed notion of time enables accurate simulation of large systolic fabrics without centralized scheduling overhead.

A DAM configuration defines the grid topology of PEs (1D, 2D, or toroidal), the latency and bandwidth of each inter-PE link, and the capacity of each PE's local register file, which limits how many intermediate values can be retained before backpressure occurs. Channels act as bounded FIFOs—if a producer sends into a full channel or a consumer reads from an empty one, execution naturally stalls, emulating the pipeline bubble and flow-control effects of real hardware. This

makes DAM particularly well suited for modeling spatial accelerators where computation and communication interleave tightly.

Cyclotron lowers its per-PE systolic IR directly into DAM programs consisting of a minimal instruction set: `LOOP`, `SEND(chan, src)`, `RECV(chan, dst)`, and arithmetic kernels such as `GEMM`, `TRSM`, and `SOFTMAX`. Each PE executes its own sequence of these instructions while communicating with neighbors through explicit channels, preserving the original space–time dependencies encoded by the compiler's schedule. The simulator executes all PE programs concurrently, collecting per-cycle metrics such as utilization, stall rates, and active instruction counts.

This backend grounds Cyclotron's design-space exploration in hardware-realistic behavior. By sweeping DAM parameters such as link latency, FIFO bandwidth, register file size, and mesh topology, we can quantify how spatial mappings, stationary choices, and kernel fusions impact throughput and utilization.

9.2 Multinode Target

In addition to the DAM simulator, Cyclotron provides an MPI backend and runtime that executes the same per-PE programs across distributed memory systems. Each PE's instruction stream is interpreted by the runtime, mirroring the structure of the DAM backend's instruction set and runtime. Each Cyclotron PE is mapped to an MPI rank, and each channel in the dataflow graph becomes an explicitly allocated communication buffer between ranks. The backend supports point-to-point primitives, including both blocking MPI calls and nonblocking MPI_Isend and MPI_Irecv pairs for data exchanges and the SUMMA-style broadcasts. Local compute kernels are implemented as BLAS or LAPACK calls.

Unlike the DAM runtime which models cycle timing,the MPI backend executes at message granularity, prioritizing scalability with HPC clusters over cycle accuracy. This allows Cyclotron to run large-scale workloads on real systems. Our backend targets CPU clusters, and could be easily extendible to support GPU clusters as well, in which communication would simply use NCCL's send/receive primitives instead of MPI's. The DAM simulator and MPI backend ingest the exact same per-PE program emitted by the Cyclotron compiler. Together with the DAM simulator, the MPI backend provides a continuum from detailed hardware modeling to scalable execution, enabling end-to-end evaluation of systolic schedules—from simulated cycles to distributed performance on real clusters.

10 Evaluation

10.1 Distributed Matrix Multiply

We compare different implementations of distributed matrix multiply using our MPI-backend, comparing performance against ScaLAPACK, a standard handwritten library for distributed linear algebra (itself built on top of MPI). Using Cyclotron, we generate programs for output stationary Cannon's algorithm, weight stationary Cannon's algorithm, as well as PUMMA and SUMMA. Figure 15 shows that Cyclotron's performance generally outperforms ScaLAPACK, measured in GFLOPS/processor. We note that SUMMA, in which all processors receive all their inputs during an initial broadcast phase, performs best. It is the most asynchronous of the matrix multiply algorithms; processors do not wait on neighboring processors that are busy computing GEMMs, as they do in Cannon's algorithm. Because all data is initially broadcasted from the source nodes, the dependencies on neighboring processors that are inherent to Cannon's algorithm are now nonexistent. As PUMMA is an interpolation between Cannon's algorithm and SUMMA, it's performance is between the two. We note that in this case the weight stationary algorithm performs the slowest. We postulate this is due to the additional dependency weight stationary Cannon's algorithm has

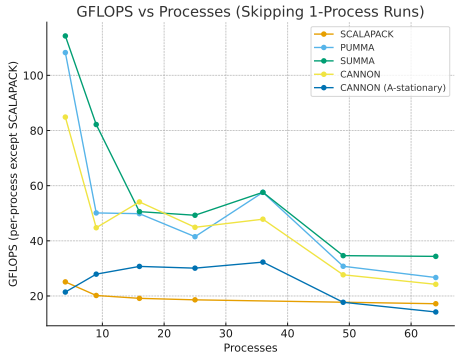
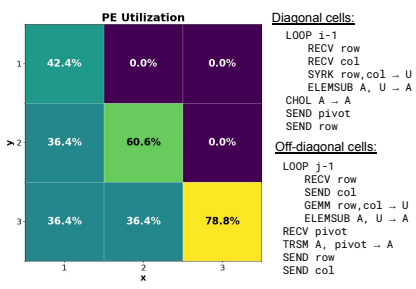
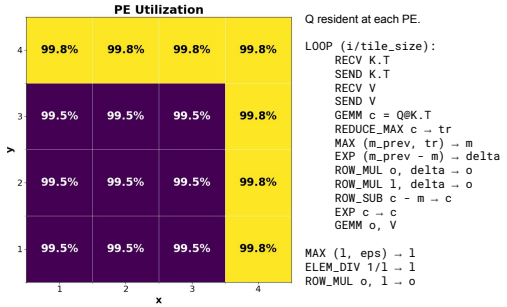


Fig. 15. Cyclotron matrix multiply algorithms vs ScaLAPACK

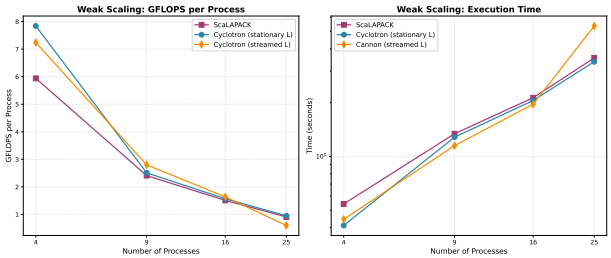


(a) Cholesky Decomposition utilization under DAM backend.

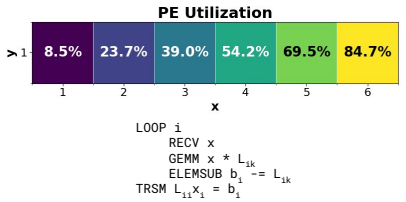


(b) FlashAttention-2 PE activity and operation sequence.

Fig. 16. (a) Example Cholesky mapping showing PE utilization. (b) FlashAttention-2 mapping with corresponding operation sequence.



(a) Weak scaling of TRSM compared to ScaLAPACK.



(b) TRSM PE utilization under DAM backend.

Fig. 17. TRSM performance and utilization.

over output stationary Cannon's algorithm. In the weight stationary Cannon's algorithm, partial outputs C must be computed by a GEMM, after which these C blocks are sent to neighboring processors, representing a WRITE-READ dependency. In the output stationary algorithm, C is kept local, and A and B shift across processors, meaning the dependency between the two can be

relaxed, allowing the runtime to use asynchronous versions of the send and receive instructions. We postulate performance improvement over ScaLAPACK is due to the much lighter runtime of Cyclotron vs. ScaLAPACK, as both use MPI in their communication layer and MKL-BLAS to perform local computations.

10.2 Distributed and Simulated Triangular Solve

We show utilization results in DAM for triangular solve in Figure 17b. As we can see, the utilization increases further down the array, since the number of GEMM updates grows longer as we progress further along i. We show multinode results for the triangular solve in Figure 17a, in which we compare two schedules of the triangular solve against ScaLAPACK. In one schedule, L is kept stationary with the *prefetch* command, while L is streamed in the other with the *stream* command. All three implementations are comparable, showing the benefits of keeping an input matrix stationary in a triangular solve, whereas it did not help for the distributed matrix multiply.

10.3 Simulated Cholesky and Flash Attention

First, we introduce the tiled Cholesky decomposition and Flash Attention recurrences.

Cholesky Recurrence. Let $A \in \mathbb{R}^{n\times n}$ be symmetric positive definite and partitioned into $b \times b$ tiles A_{ij}. For each panel $j = 1 \dots T$, the diagonal block accumulates all prior Schur-complement updates via blocked SYRK:

$$A_{jj}^{(j)} = A_{jj} - \sum_{k<j} \mathrm{SYRK}(A_{jk}), \qquad L_{jj} = \mathrm{CHOL}(A_{jj}^{(j)}).$$

Each off-diagonal tile $i > j$ performs its corresponding update using blocked GEMM, followed by a triangular solve:

$$A_{ij}^{(j)} = A_{ij} - \sum_{k<j} \mathrm{GEMM}(A_{ik}, A_{jk}), \qquad L_{ij} = \mathrm{TRSM}(A_{ij}^{(j)}, L_{jj}).$$

The resulting tiles $\{L_{ij}\}$ form the lower-triangular Cholesky factor.

FlashAttention-2 Recurrence. Let $Q \in \mathbb{R}^{N\times d}$, $K, V \in \mathbb{R}^{M\times d}$ and $\alpha = 1/\sqrt{d}$. Keys/values are processed in tiles J_t, maintaining per-query state $(m_i^{(t)}, l_i^{(t)}, o_i^{(t)} \in \mathbb{R}^d)$:

$$S_{i,J_t} = \alpha\, Q_{i,:} K_{J_t,:}^{\top}, \qquad \hat{m}_i^{(t)} = \max\big(m_i^{(t-1)},\ \max S_{i,J_t}\big),$$

$$l_i^{(t)} = e^{m_i^{(t-1)} - \hat{m}_i^{(t)}}\, l_i^{(t-1)} + \sum_{j\in J_t} e^{S_{i,j} - \hat{m}_i^{(t)}},$$

$$o_i^{(t)} = e^{m_i^{(t-1)} - \hat{m}_i^{(t)}}\, o_i^{(t-1)} + \sum_{j\in J_t} e^{S_{i,j} - \hat{m}_i^{(t)}}\, V_{j,:}, \qquad m_i^{(t)} = \hat{m}_i^{(t)}.$$

Final outputs are $O_{i,:} = o_i^{(T)} / l_i^{(T)}$.

We compare the utilizations for these two recurrences in DAM, as shown in Figure 16. Choleskey's tiled recurrence is triangular and panel-structured, yielding a wavefront of parallelism that expands down the column. Each diagonal cell performs a Cholesky decomposition on a tile, preceded by SYRK updates if necessary. These SYRK updates grow in length as the systolic array dimensions grow, leading to higher utilization on diagonal cells and higher utilization lower on the systolic array. In contrast, the Flash-Attention-2 recurrence is relatively balanced, as each PE holds a query tile and computes a series of matrix multiplications, followed by elementwise activations in place.

11 Related Work

Systolic Arrays were first introduced by Kung and Leiserson [Kung 1982; Kung and Leiserson 1979]. Classic work by Rajopadhye and Fujimoto [Rajopadhye and Fujimoto 1990] established how to synthesize systolic arrays from (uniform/affine) recurrence equations, including conditions for linear schedules and array allocation. Their framework is entirely theoretical, and does not include compiler infrastructure. Our treatment generalizes these foundations to modern multi-tensor kernels, making skews and inter-space interfaces first-class in the IR.

Our IR is an extension and re-imagining of the *assignment free* notation from the*Algorithms of Informatics* textbook by Gyires [Gyires 2013]. Their notation is similar to Halide's input language [Ragan-Kelley et al. 2013], in which all logical access are array-style (e.g. $A(i, j, k)$, as opposed to our $(ijk)_A^{mul}$). This array notation is is shared by the Stellar [Genc et al. 2024] and Gemmini [Genc et al. 2021] work for designing matrix-multiply systolic arrays. Our notation treats the indices and iteration space as the first-class citizens, and extends the notation to handle recurrences, multiple iteration spaces, and inter-space communication. The Wavefront Array Processor [Kung et al. 1982] was a programmable systolic array. It included a language that looked similar to the per-PE programs that Cyclotron generates. Early work on codesigning a processor and a systolic compiler (using the afine model) was done by Lam and Wolf [2004]. There is a long history of dataflow compilers based on the polyhedral and affine models. AutoSA [Wang et al. 2021] and PolySA [Cong and Wang 2018] use the polyhedral model to find unit-length read distances amongst input equations, and then compile the equations to dataflow. However, they are primarily targeted at matrix multiply and convolutions, and do not handle recurrences (i.e., equations with dependencies) beyond the LU decomposition, as the LU decomposition having a similar dataflow to matrix multiply.

DISTAL [Yadav et al. 2022], the closest related project to this work, decouples tensor formats from distributed schedules, spanning both classic and modern matrix-multiplication algorithms (e.g., Cannon, COSMA) and compiling to a task runtime across CPU/GPU clusters. Its core contribution is a scheduling language that maps computation onto distributed machines, where operations "rotate" and move across processors. In contrast, our work focuses on the temporal and spatial alignment of operands—representing inter-processor data movement as streams within a unified recurrence abstraction rather than as discrete cluster-level tasks. Whereas DISTAL is primarily a tensor-algebra compiler and cannot express recurrences with output dependencies, our IR directly captures such dependencies and treats them as first-class citizens. Moreover, DISTAL supports only output-stationary algorithms, while our space–time mapping generalizes to input-stationary and mixed-stationary executions, enabling a broader class of distributed and systolic schedules.

The classic distributed matrix multiply algorithms implemented by Cyclotron and DISTAL include Cannon's algorithm [Cannon 1969], SUMMA [Van De Geijn and Watts 1997], and PUMMA [Choi et al. 1994]. Early work by Heath and Romine [1988] explored different forms of the distributed triangular solve.

12 Conclusion

Cyclotron unifies the specification, scheduling, and execution of systolic algorithms within a single compiler framework. By expressing computations as recurrences over indexed tensors, it exposes both spatial and temporal structure directly in the IR, enabling the compiler to generate hardware-like schedules while remaining architecture-agnostic. Its per-PE compilation model and dual backends—one for simulation, the other distributed via MPI—bridge the gap between theoretical mappings and executable implementations. Through these abstractions, Cyclotron shows that high-level recurrence semantics can provide a practical foundation for reasoning about dataflow, synchronization, and communication across a wide range of architectures.

Looking ahead, Cyclotron opens several avenues for exploration: integrating automatic schedule synthesis, synthesizing hardware on reconfigurable architectures, and coupling the compiler with hardware cost models for co-design. More broadly, it points toward a unified programming model for spatial computing—spanning single-chip kernels to distributed accelerators—grounded in a single, recurrence-based view of computation.

References

Lynn Elliot Cannon. 1969. *A cellular computer to implement the Kalman filter algorithm.* Montana State University.

Jaeyoung Choi, David W Walker, and Jack J Dongarra. 1994. PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience* 6, 7 (1994), 543–570.

Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-based systolic array auto-compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

Paul Feautrier. 1991. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming* 20, 1 (1991), 23–53.

Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, et al. 2021. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 769–774.

Hasan Nazim Genc, Hansung Kim, Prashanth Ganesh, and Yakun Sophia Shao. 2024. Stellar: An Automated Design Framework for Dense and Sparse Spatial Accelerators. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 409–422.

János P. Gyires. 2013. Systolic Systems. In *Algorithms of Informatics, Volume 2*. University of Debrecen. https://gyires.inf.unideb.hu/KMITT/a52/ch16.html Accessed: 2025-11-10.

Michael T. Heath and Charles H. Romine. 1988. Parallel Solution of Triangular Systems on Distributed-Memory Multiprocessors. *SIAM J. Sci. Statist. Comput.* 9, 3 (1988), 558–588. arXiv:https://doi.org/10.1137/0909037 doi:10.1137/0909037

Hsiang-Tsung Kung. 1982. *Why systolic architecture?* Design Research Center, Carnegie-Mellon University Pittsburgh, PA, USA.

Hsiang Tsung Kung and Charles E Leiserson. 1979. Systolic arrays (for VLSI). In *Sparse Matrix Proceedings 1978*, Vol. 1. Society for industrial and applied mathematics Philadelphia, PA, USA, 256–282.

Sun-Yuan Kung, Arun, Gal-Ezer, and Bhaskar Rao. 1982. Wavefront Array Processor: Language, Architecture, and Applications. *IEEE Trans. Comput.* C-31, 11 (1982), 1054–1066. doi:10.1109/TC.1982.1675922

Monica S Lam and Michael E Wolf. 2004. A data locality optimizing algorithm. *ACM SIGPLAN Notices* 39, 4 (2004), 442–459.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

Sanjay V Rajopadhye and Richard M Fujimoto. 1990. Synthesizing systolic arrays from recurrence equations. *Parallel computing* 14, 2 (1990), 163–189.

Shiv Sundram, Muhammad Usman Tariq, and Fredrik Kjolstad. 2024. Compiling Recurrences over Dense and Sparse Arrays. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 103 (April 2024), 26 pages. doi:10.1145/3649820

Muhammad Usman Tariq, Shiv Sundram, and Fredrik Kjolstad. 2025. REPTILE: Performant Tiling of Recurrences. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 296 (Oct. 2025), 27 pages. doi:10.1145/3763074

Robert A Van De Geijn and Jerrell Watts. 1997. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience* 9, 4 (1997), 255–274.

Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 93–104.

Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. DISTAL: the distributed tensor algebra compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 286–300. doi:10.1145/3519939.3523437

Nathan Zhang, Rubens Lacouture, Gina Sohn, Paul Mure, Qizheng Zhang, Fredrik Kjolstad, and Kunle Olukotun. 2024. The Dataflow Abstract Machine Simulator Framework. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 532–547.