Cyclotron: The Streaming Multiprocessor Abstraction is Broken

Shiv Sundram
Stanford University
USA

Akhilesh Balasingam
Stanford University
USA

Thierry Tambe
Stanford University
USA

Kunle Olukotun
Stanford University
USA

Fredrik Kjolstad
Stanford University
USA

ABSTRACT

The abstraction of the GPU as a collection of isolated Streaming Multiprocessors (SMs) is broken. While modern hardware has evolved into static topologies of interlinked processing elements—resembling distributed systems more than stream processors—programming models have not kept pace. Novel features like the Tensor Memory Accelerator (TMA), Tensor Core, Cooperative Groups, and Distributed Shared Memory are retroactively grafted onto legacy streaming APIs, obscuring the reality that fixed-function units are increasingly decoupled from warp-synchronous execution. This forces developers to navigate a fractured ecosystem of synchronization and communication primitives.

We introduce Cyclotron [5], a unified programming language that redefines accelerator programming through tiles and abstract iteration spaces. Rather than defining computation between distinct physical processing elements, Cyclotron defines the flow of data tiles across a logical coordinate system, which is then projected onto physical hardware via a flexible space-time mapping. We demonstrate that this abstraction is scale-invariant: a single Cyclotron program can be efficiently lowered to MPI-based clusters, architectural simulators for interlinked chiplets, and even high-level synthesis (HLS) for custom hardware that is performance-competitive with commercial TPUs and NVIDIA L40S on workloads like Flash Attention, matrix multiply, and the Cholesky Decomposition.

1 INTRODUCTION

The streaming multiprocessor abstraction is broken. Modern GPUs and purpose-built accelerators (e.g., TPUs, Wafer-Scale Engines) have converged into being static topologies of interlinked processing elements. This necessitates a novel kernel-level programming model for ML accelerators that 1) treats inter-processor communication as a first class citizen 2) can be used to program the machine at any level of the hierarchy, whether it be between chiplets on a single chip or between individual processors in a datacenter, and 3) uses tiles as the fundamental unit of computation. For context, a tile represents a multi-dimensional block of data (e.g., a sub-matrix) that acts as the atomic unit of work, allowing programmers to orchestrate the movement of granular blocks across the abstract space rather than micro-managing scalar values. It must also support parallel neighbor exchanges, broadcasts, and reduction primitives to maintain ease of programmability.

We introduce Cyclotron, a unified programming language that can be used to program distributed architectures, but at any level of the hierarchy. The key idea of Cyclotron is that instead of defining communication and computation between distinct PEs, it defines communication of tiles between nodes of an abstract iteration space. This program is then composed with a simple space-time mapping that allows for the same program to be flexibly mapped to different architectures. We demonstrate how the same Cyclotron program can be used to portably program MPI-based clusters, an architectural simulator of interlinked chiplets, and even emit HLS to define hardware that is competitive with A100s and TPUs.

LATTE '26, March 23, 2026, Pittsburgh, PA
2026.

2 CYCLOTRON SEMANTICS

Cyclotron is a recurrence compiler [4, 6, 7] that decouples the specification of data movement from the underlying hardware topology. By treating the execution as a flow of tiles across an abstract iteration space, distinct distributed algorithms become mere variations in coordinate access patterns.

Figure 1 demonstrates this unification. We implement two distinct matrix multiplication algorithms—Cannon's [1] (shift-based) and SUMMA [8] (broadcast-based)—using the same Cyclotron kernel semantics. The difference lies solely in the communication line: Cannon requests data from a neighbor (e.g., $j-1$), while SUMMA requests data from a stationary feeder column (0). This trivial modification in Cyclotron would require entirely different MPI primitives (e.g., `MPI_Sendrecv` vs. `MPI_Bcast`) in a traditional model.

Listing 1: Cannon (Shift)

```
def CANNON(self, ctx, Ats, Bts, Cts):
  #reads from physical memory
  Ats(i,0,k) = A[i,k]
  Bts(0,j,k) = B[k,j]

  # Communication: Neighbor Shift
  Ats(i,j,k) = Ats(i, j-1, k)
  Bts(i,j,k) = Bts(i-1, j, k)

  # Compute: Local GEMM
  res = ctx.GEMM(Ats(i,j,k), ...)
  Cts(i,j,k) = res
```

Listing 2: SUMMA (Broadcast)

```
def SUMMA(self, ctx, At, Bt, Ct):
  #reads from physical memory
  Ats(i,0,k) = A[i,k]
  Bts(0,j,k) = B[k,j]

  # Communication: Broadcast
  Ats(i,j,k) = Ats(i, 0, k)
  Bts(i,j,k) = Bts(0, j, k)

  # Compute: Local GEMM
  res = ctx.GEMM(Ats[i,j,k], ...)
  Cts(i,j,k) = res
```

Figure 1: Cyclotron unifies distinct distributed algorithms into coordinate changes. Cannon uses relative indexing ($j-1$), while SUMMA uses absolute indexing (0).

In Cyclotron, data resides in the *abstract iteration space*, distinct from physical tensor storage. Accesses like $Ats(i, j-1, k)$ do not index physical memory; rather, they reference the value of tensor A flowing through the coordinate $(i, j-1, k)$. While standard tensor notation implies static storage ($C[i,j] = \sum_k A[i,k]B[k,j]$), Cyclotron lifts these operands into the higher-dimensional iteration space to make data dependencies and data movement explicit.

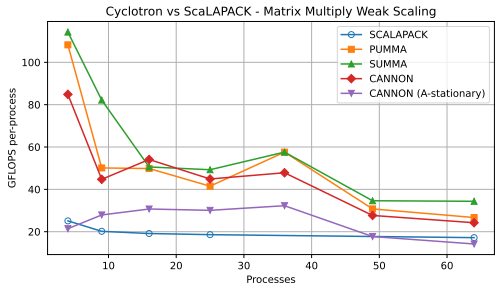
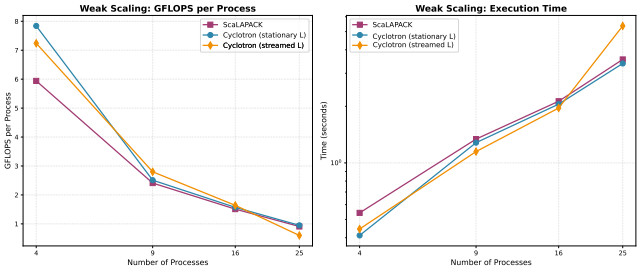
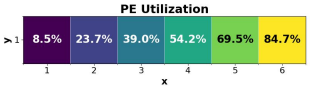


Figure 2: Cyclotron matrix multiply algorithms vs ScaLAPACK. Each process calculates a 1024x1024 block.



(a) Weak scaling of TRSM compared to ScaLAPACK.



(b) TRSM PE utilization under DAM backend.

```
LOOP i
    RECV x
    GEMM x1=x*L_ik
    ELEMSUB b_i-=x1
TRSM L_ii x_i = b_i
```

(c) Emitted PE instructions for TRSM

Figure 3: TRSM performance and utilization.

This decoupling allows a single kernel to generate distinct architectural variants via *space-time mappings*. For example, mapping (i, j) to space and k to time yields an Output-Stationary systolic array (where C is resident). Conversely, mapping (i, k) to space yields a Weight-Stationary dataflow—common in DNN accelerators—where weights (A) remain resident while inputs and partial sums and activations (C, B) stream through the array. In the spirit of tile-centric programming, all data-access refer to tiles; $Ats(i, j, k)$ thus references a tile, in which tile sizes are specifiable by the user.

The concept of space-time mappings builds upon a rich history of synthesizing hardware from uniform recurrence equations, such as the mappings explored in SuSy [3]. While such frameworks support tiling, they are typically constrained to perfectly nested rectangular loops with uniform dependencies, such as standard matrix multiplication. Cyclotron breaks this limitation by natively supporting imperfectly nested bounds and complex triangular dependencies (such as $i < j < k$ found in Cholesky and TRSM algorithms), while maintaining strict portability across MPI clusters, chiplets, and custom silicon.

Furthermore, Cyclotron's tile-centric abstraction gracefully extends to modern hardware features. By operating on tiles rather than scalars, operations like `ctx.GEMM` (as seen in Figure 1) natively invoke optimized library routines (e.g., BLAS) or fixed-function hardware, such as Tensor Cores, without requiring fragile, vendor-specific pragmas. This abstraction also scales to asynchronous memory hierarchies. For instance, while traditional dataflow relies on shifting data between neighboring points in the iteration space, a Tensor Memory Accelerator (TMA) read represents a direct, asynchronous access to physical memory. Cyclotron allows developers to seamlessly compose both spatial neighbor-exchanges (curved brackets) and direct physical memory reads (square brackets) within the same abstract coordinate system, maximizing utilization without sacrificing programmability.

This level of expressivity highlights why the traditional SM abstraction is fundamentally broken when attempting to exploit spatial parallelism. Because its block-to-SM scheduling algorithm is opaque and hidden from the user, traditional inter-block (and inter-SM) communication in GPUs must be routed through slow DRAM writes at strict kernel boundaries. While newer features attempt to enable inter-block synchronization from within the kernel, they do not compose cleanly with the legacy streaming API. In fact, launching a Cooperative Groups kernel capable of inter-block synchronization requires a completely separate host API call (`cudaLaunchCooperativeKernel`), explicitly breaking the standard triple-chevron («<...»>) syntax used for traditional kernels with unlimited blocks. Recurrences offer a superior alternative by natively allowing for direct inter-PE communication from within a single kernel. By describing the pure, algorithmic flow of data through an abstract space, Cyclotron avoids the rigid, opaque execution constraints of legacy GPU APIs.

Cyclotron can thus be used to explore a vast architectural design space without modifying the underlying kernel and dataflow logic, ensuring correctness across diverse hardware targets. The programming model is well suited to targeting any architecture composed of a mesh of physical processing elements. It prioritizes processor-to-processor communication as a first class citizen. At the same time, it provides the user flexibility of specifying a space-time mapping that allows for exploration of algorithmic variants and portability over architectures.

3 ARCHITECTURAL BACKENDS AND RESULTS

Cyclotron currently supports three distinct targets, proving its ability to scale from intra-chip kernels to inter-node clusters.

3.1 Distributed Clusters (MPI)

For scale-out systems, Cyclotron lowers tile movements into MPI primitives. The compiler maps the spatial coordinates (i, j) of the iteration space directly to MPI ranks in a grid communicator. Crucially, because Cyclotron captures the *geometry* of data movement (e.g., neighbor shift vs. broadcast), the backend maps all dataflow over the iteration space to machine-specific communication primitives: a shift becomes `MPI_Sendrecv`. The compiler manages the space-time mapping from iteration space to processor coordinates, eliminating the manual "rank algebra" typically required in high-performance distributed kernels. Figure 2 shows that Cyclotron emitted matrix multiplies match or outperform ScaLAPACK.

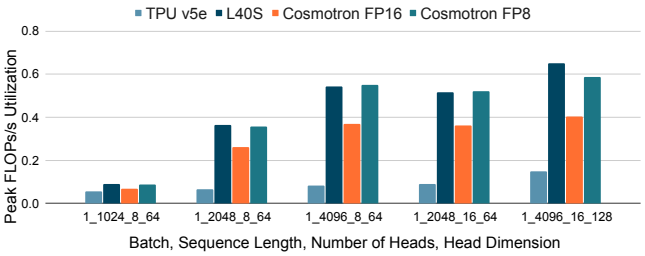


Figure 4: Peak FLOPs/s Utilization. Cosmotron MXFP8 and FP16 designs are compared against L40S and TPUv5e JAX library implementations

3.2 Chiplet Simulator (Rust)

When targeting accelerators of interlinked chiplets, Cyclotron maps logical tiles to physical Processing Elements (PEs). Since the dataflow is known at compile-time, Cyclotron emits code targeting DAM [9], a cycle-accurate simulator with specifiable link bandwidths, link latencies, and cycle counts for mathematical operations, useful when needing to measure the PE utilization for new workloads. Figure 3 shows utilization of a 6-PE TRSM running on DAM. Its load imbalance is inherent to TRSM, not Cyclotron, whose distributed TRSM matches ScaLAPACK in Figure 3a.

TRSM is an ideal workload to evaluate in this distributed chiplet context because it exposes the severe synchronization bottlenecks of traditional streaming abstractions. In legacy models (e.g., cuBLAS or ScaLAPACK's PBLAS), TRSM's sub-operations—the triangular solve and the subsequent GEMM updates—must be invoked as separate kernels. Because each kernel launch forces a global synchronization, the critical communication path between the triangular solve and GEMM stages becomes severely bottlenecked. Currently, the only native mechanism to bypass these kernel boundary overheads in CUDA is via Cooperative Groups and Distributed Shared Memory (DSM). However, these retrofitted APIs fundamentally violate the traditional streaming multiprocessor abstraction. Cooperative Groups force the programmer to guarantee that a specific set of thread blocks are simultaneously resident on the GPU, breaking the independent, asynchronous execution model of legacy CUDA blocks. Furthermore, communicating via DSM imposes heavy-handed synchronization across an entire SM cluster, rather than lightweight, point-to-point coordination between a specific sender and receiver. Cyclotron eliminates this friction entirely. Because the programming model is defined by the flow of tiles across the iteration space, processing elements simply exchange messages with their nearest neighbors asynchronously and compute as soon as dependencies are met, gracefully handling complex distributed coordination without falling back on rigid, synchronous hardware constraints.[1]

3.3 High-Level Synthesis (HLS)

For custom silicon (FPGA/ASIC), Cyclotron includes an HLS backend known as *Cosmotron*, which contains a rich set of scheduling and architectural primitives that allow Cyclotron to act as a hardware generator. The space-time mapping determines which dimensions are parallelized (spatial unrolling) and which are serialized (temporal loops). The backend emits synthesizable HLS C++, generating the necessary FIFOs, double-buffers, and systolic control logic. This allows a software engineer to explore architectural tradeoffs—such as changing a systolic array from weight-stationary to output-stationary—by changing a single line in the mapping file rather than rewriting low-level code. In Figure 4, we show that Cosmotron-generated designs for FlashAttention [2] can achieve peak FLOPs/s utilization that exceed a JAX TPU implementation's and on par with that of a published NVIDIA L40S implementation.

[1] A comprehensive performance evaluation of complex triangular kernels, such as the Cholesky decomposition, against state-of-the-art distributed baselines is the subject of ongoing work.

4 CONCLUSION

The era of the isolated streaming multiprocessor is over. Modern accelerators are effectively distributed systems that demand new programming models. Cyclotron answers this call by elevating inter-PE communication to a first-class citizen, unifying execution from chiplets to clusters.

REFERENCES

[1] Lynn Elliot Cannon. 1969. *A cellular computer to implement the Kalman filter algorithm.* Montana State University.
[2] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems* 35 (2022), 16344–16359.
[3] Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wang, Brendan Sullivan, Zhiru Zhang, Yun Liang, et al. 2020. SuSy: A programming model for productive construction of high-performance systolic arrays on FPGAs. In *Proceedings of the 39th International Conference on Computer-Aided Design*. 1–9.
[4] Sanjay V Rajopadhye, S Purushothaman, and Richard M Fujimoto. 1986. On synthesizing systolic arrays from recurrence equations with linear dependencies. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 488–503.
[5] Shiv Sundram, Akhilesh Balasingam, Nathan Zhang, Kunle Olukotun, and Fredrik Kjolstad. 2025. Cyclotron: Compilation of Recurrences to Distributed and Systolic Architectures. *arXiv preprint arXiv:2511.09987* (2025).
[6] Shiv Sundram, Muhammad Usman Tariq, and Fredrik Kjolstad. 2024. Compiling Recurrences over Dense and Sparse Arrays. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 250–275.
[7] Muhammad Usman Tariq, Shiv Sundram, and Fredrik Kjolstad. 2025. REPTILE: Performant Tiling of Recurrences. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 670–696.
[8] Robert A Van De Geijn and Jerrell Watts. 1997. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience* 9, 4 (1997), 255–274.
[9] 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.