Infinity Stream: Portable and Programmer-Friendly In-/Near-Memory Fusion

Zhengrong Wang
seanzw@cs.ucla.edu
UCLA, USA

Christopher Liu
chrisliu@cs.ucla.edu
UCLA, USA

Aman Arora
aman.kbm@utexas.edu
UT Austin, USA

Lizy John
ljohn@ece.utexas.edu
UT Austin, USA

Tony Nowatzki
tjn@cs.ucla.edu
UCLA, USA

ABSTRACT

In-memory computing with large last-level caches is promising to dramatically alleviate data movement bottlenecks and expose massive bitline-level parallelization opportunities. However, key challenges from its unique execution model remain unsolved: automated parallelization, transparently orchestrating data transposition/alignment/broadcast for bit-serial logic, and mixing in-/near-memory computing. Most importantly, the solution should be programmer friendly and portable across platforms.

Our key innovation is an execution model and intermediate representation (IR) that enables hybrid CPU-core, in-memory, and near-memory processing. Our IR is the tensor dataflow graph (tDFG), which is a unified representation of in-memory and near-memory computation. The tDFG exposes tensor-data structure information so that the hardware and runtime can automatically orchestrate data management for bit-serial execution, including runtime data layout transformations. To enable microarchitecture portability, we use a two-phase, JIT-based compilation approach to dynamically lower the tDFG to in-memory commands.

Our design, infinity stream, is evaluated on a cycle-accurate simulator. Across data-processing workloads with fp32, it achieves 2.6× speedup and 75% traffic reduction over a state-of-the-art near-memory computing technique, with 2.4× energy efficiency.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**.

KEYWORDS

Stream-Based ISAs, Programmer-Transparent Acceleration, In-Memory Computing, Near-Memory Computing

ACM Reference Format:
Zhengrong Wang, Christopher Liu, Aman Arora, Lizy John, and Tony Nowatzki. 2023. Infinity Stream: Portable and Programmer-Friendly In-/Near-Memory Fusion. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3582016.3582032

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9918-0/23/03.
https://doi.org/10.1145/3582016.3582032

1 INTRODUCTION

As multicore systems scale, growing data movement bottlenecks incentivize a memory-centric paradigm over the traditional core-centric paradigm. One realization of this is *near-memory computing*, where specialized hardware is added near memory banks to decouple computation from core pipelines and reduce communication demand. An alternative is *in-memory computing*, which augments memory arrays with the ability to perform simple computations at massive parallelism.

While in-memory computing has been applied at different hierarchy levels and technologies, the trend of incorporating extremely-large L3 caches has made the proposition of in-SRAM computation quite attractive. For example, the latest AMD EPYCs have >100MB of L3, which would translate to multiple millions of bitwise processing elements. As prior work has shown, bit-serial SRAM [32] has a computation density that is significantly higher than possible on SIMD vector units, and the energy benefits are substantial [17].

But there are still barriers to broad adoption. An ideal in-memory system would be as programmer-transparent as possible, be compatible with existing core-centric and near-data execution without adding much overhead, and also preserve program compatibility with future microarchitectures. No existing in-memory system has achieved all three due to the challenges of the unique paradigm:

- **Transparent Orchestration**: Bit-serial logic requires transposing large arrays, managing on-chip space, and enforcing bitline alignment. A suitable data layout, tiling, and explicit reuse are critical to reducing data traffic. Also, distributing computation to bitlines requires massive vector parallelism. Ideally, this orchestration would be done without any programmer involvement.
- **Fused In-/Near-Memory Computing**: Sometimes it is better to split the work between in-/near-memory computing. E.g. an in-memory reduction to produce partial results, which are reduced to the final value by near-memory computing; or a phase with both irregular and regular data structures, where only the latter is suitable for in-memory. This requires a unified execution model and low-overhead hardware implementation.
- **Program Portability**: High-performance implementations require exploiting both low-level microarchitecture details and software parameters, but fixing them would prevent portability and compatibility.

Existing in-memory works have not fully addressed these, as they are either somewhat domain specific (e.g. [9, 15, 16]) or put a significant burden on programmers (e.g. [1, 17, 30]).

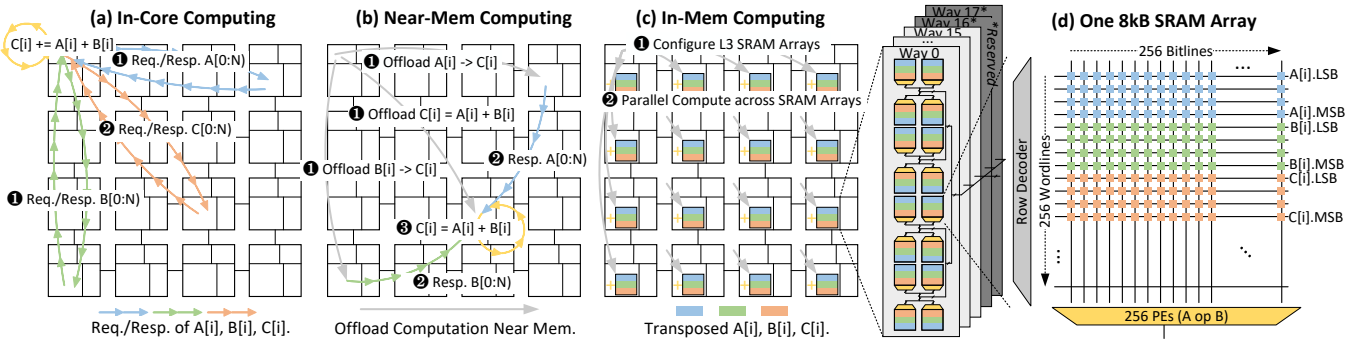

Figure 1: Overview of In-Core/Near-Mem/In-Mem Computing Paradigms

One source of inspiration is prior work on transparent near-memory called near-stream computing (NSC) [64], which augments the ISA with explicit abstractions for memory access patterns (called streams) and associated computation. In NSC, streams are offloaded to execute near-L3 when there is little locality in private caches. However, streams do not convey enough information and semantics for in-memory computing. They are inherently sequential, they lack information about data size and reuse which are needed to decide the best layout and tiling, and they also lack the necessary information to guarantee bitline alignment between data structures.

To solve this problem, our insight is that the portions of workloads that can benefit from in-memory computation have very simple parallelism and reuse patterns that can be analyzed perfectly: generally affine access to multidimensional tensors. This information is sufficient to determine an optimized data layout and tiling, as well as for generating array-level data-movement commands to exploit reuse. Thus, our approach is to make parallel tensor access and relative memory alignment to be first-class primitives of program execution. The augmented program representation is called a *tensor dataflow graph* (tDFG). To first order, each tensor element is mapped to a bitline, and the dataflow instructions are mapped to in-memory commands.

Further, the tDFG is a unified abstraction for near-data and in-memory, as it defines the semantics when near-data streams have dependencies on in-memory tensor operations, and vice versa. For example, a load stream may broadcast the CNN weights to all bitlines (stream to tensor), or a reduction stream can execute near L3 banks to collect partial results from each SRAM (tensor to stream).

Finally, to enable portable binaries, we adopt a two-phase compilation approach. The tDFG serves as the compilers' intermediate representation (IR) and the program representation, and in-memory commands for SRAMs are generated by dynamic compilation of the tDFG. This enables the binary to be independent of microarchitecture, and for tensor programs to take advantage of runtime constants (tensor size/shape). Any difficult analysis happens while generating the optimized tDFG, thus lowering is fast.

Our overall approach is called *infinity stream*, which transparently and flexibly enables offloading to either in-/near-memory, fusing these paradigms. We implement our framework using LLVM and a custom dynamic compiler, and evaluate with gem5 [45]. For data-parallel workloads with in-memory phases using fp32, using a 64-core system with 128MB L3, infinity stream achieves 2.6× speedup and 75% traffic reduction over near-memory only [64] with 2.4× energy efficiency, and 5.1× (up to 8.9×) speedup over a high-performance multicore. Specifically, our contributions are:

- Execution model for fused and general in-/near-memory computing, with automated data-layout transformations.
- In-memory compiler from plain-C, with optimizations for parallelism and data movement, enabling a programmer-friendly interface to efficient in-/near-memory execution.
- tDFG abstraction and ISA with μarch/runtime/JIT support for enabling portable in-memory execution.
- Quantifying the benefits of in-memory vs near-memory for bit-serial SRAM acceleration.

Paper Organization: §2 gives background on in-/near-memory and overviews our approach, followed by the execution model and tDFG IR in §3. §4 details the runtime and dynamic compilation, with the μarch in §5 and limitations in §6. Methodology and evaluation are in §7 and §8, and related work is in §9.

2 BACKGROUND AND OVERVIEW

Here we overview the three computing paradigms with a simple vector addition example. This characterizes in-memory computing and its challenges, which motivate this work.

2.1 Near-Memory Computing

Conventional systems adopt a core-centric view: all computation is centralized in the core, with data fetched from the memory subsystem. Fig 1(a) shows a tiled multi-core system. Each tile contains a core with a private L1/L2 and a shared L3 cache bank, and is connected by a mesh network-on-chip (NoC). To perform `C[i]=A[i]+B[i]`, the core issues multiple requests to fetch `A[i]` and `B[i]`, as well as writing back `C[i]`. Vectorization and multi-threading can be used to exploit the massive data parallelism in this example. One major overhead here is the unnecessary data movement, as all three arrays `A[]`, `B[]` and `C[]` have no reuse at all. Techniques like prefetching and cache bypassing can only partially help, as the data movement is inevitable and incurs a high energy cost. Such overheads are only going to be more severe as the system scales up and the data grows.

Near-Memory Computing: To fundamentally eliminate unnecessary data movement, near-memory computing moves computation closer to the data, and has been applied in many contexts: e.g. near on-chip SRAM [53, 64], within the NoC [28, 56], near memory controller [3, 14, 44]. They also offload computation at different granularities from coarse-grained kernel-level [5, 31, 33, 42, 68, 73] to fine-grained short instruction sequences [3, 28, 56].

Near-Stream Computing: For the near-memory computing baseline, we use near-stream computing [64], which offloads long-term memory accesses (i.e. streams) with computations near the L3 cache. In Fig 1(b), the memory accesses are decoupled into three streams `A[i]`, `B[i]`, `C[i]`, and offloaded to the shared L3 banks where the data resides. Stream `A[i]` and `B[i]` directly forward their data to stream `C[i]`. Stream `C[i]` coordinates with the remote CPU core to perform SIMD ops on a spare thread, and then writes directly to L3. This significantly reduces the data traffic and control overheads.

2.2 Bit-Serial In-Cache Computing

Near-L3 approaches still read the data out from the L3 SRAM arrays, hence are still bound by the L3 cache's bandwidth. To fully unlock the massive potential data parallelism, in-memory computing moves the computation *inside* SRAM arrays. For this work, we assume the same compute SRAM technology as Neural Cache [15].

In Fig 1(c), SRAM arrays are configured to add `A[i]` and `B[i]` in *parallel* and directly write back to `C[i]`, with no sequential reads and writes at all. Fig 1(d) demonstrates how in-memory computing works in one 8kB SRAM array with 256 wordlines (row) and 256 bitlines (column). Specifically, it requires the data being transposed and bit-serial logic for computation.

Transposed Data Layout: In Fig 1(d), array elements (4 bits each) are transposed from a horizontal layout across columns to a vertical layout on the same column. E.g. the least significant bit (LSB) of `A[0]` is stored in the cell indexed by wordline 0 and bitline 0, and the most significant bit (MSB) of `A[0]` by wordline 3 and bitline 0.

Bit-Serial Compute: In-memory computing leverages bit-serial logic to compute the result. This requires operands to be aligned in the same column. In the example in Fig 1(d), `A[i]`, `B[i]`, and `C[i]` are all placed in the same bitline. To start the computation, we activate the wordlines of `A[i].LSB` and `B[i].LSB` at the same time, and the 256 PEs perform the bit operation on the sensed bit (e.g. `AND` for carry, `XOR` for addition). The PEs have cells holding intermediate results (e.g. carry of addition). The result bit is then written back to `C[i].LSB` by activating wordline 8 with the write signal. This process repeats to compute the result one bit at a time (hence "bit-serial"). It takes $O(n)$ cycles to perform integer addition and $O(n^2)$ for integer multiplication, where n is the data type width. However, this is amortized by the massive parallelism it provides.

Max System Speedup: Assuming a 64-core system with 16-way 2MB L3 banks (total 128MB) and 16 256×256 SRAM arrays/way, the peak throughput of `int32` addition is:

$$\begin{aligned} T &= N_{bank} \times N_{way} \times N_{array/way} \times N_{bitline}/Latency^{1} \\ &= 64 \times 16 \times 16 \times 256/32 = 131072 \text{ ops/cycle} \end{aligned} \quad (1)$$

Assuming each baseline core can issue one 512-bit vector op per cycle (64 × 16 = 1024 ops/cycle), in-memory provides 128× peak speedup. Fig 2 shows the speedup of two microbenchmarks with various input sizes on the baseline (AVX-512 and 1 or 64 OpenMP threads), near-L3, and in-L3 computing using bit-serial logic. We assume data is cached in L3 and already transposed for in-memory computing. in-L3 computing usually favors larger input sizes as they amortize the overhead of bit-serial operation. Despite this, in-L3 achieves the best performance for `vec_add` across all input sizes. With 4M elements, it achieves 21× over Near-L3, making it a promising approach to exploit the available data parallelism.

[1]We adopt the integer addition from [17]. System params in §7. See In-/Near-Memory Computing [18] for more details, and §9 for related works.

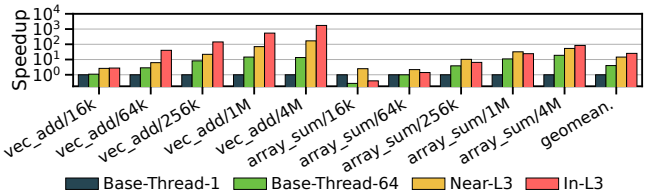

Figure 2: Speedup of Different Paradigms (Fp32)

2.3 Infinity Stream Approach Overview

We overview our approach by revisiting each of our driving requirements from the introduction.

Automated Orchestration: The data layout and movement orchestration – i.e. allocation, alignment, transposition, and tiling – are critical to the performance and applicability of in-memory computing. Thus, the system must automate this management and ease integration with conventional code. The key challenge is expressing sufficient information to the hardware and software runtime.

Our approach: We develop a program representation called the tensor dataflow graph (tDFG). The tDFG operates over tensors with explicit data-parallel semantics, and represents inter-data structure alignment with the concept of a global lattice space. Reuse can be determined precisely, and the tDFG can be annotated with hints about optimal tiling patterns. The tDFG is embedded as an extension to a traditional ISA, and gives the runtime sufficient information to make good decisions.

Fused In-/Near-Memory Computing: As suggested by Fig 2, in-memory struggles with small input sizes. Also, many code patterns like irregular control and memory (e.g. `A[B[i]]`) are only potentially suitable for *near*-memory. This motivates both a runtime selection between in-/near-memory computing, and a fused in-/near-memory paradigm.

Our approach: The tDFG can express both in-memory and near-memory opportunities in a unified representation. This generalizes the near-data approach from near-stream computing [64]. At runtime, the system decides the offload target (in-/near-memory) based on data size and access behavior. One key hardware feature is to integrate the transposed data layout with the coherence protocol to allow data communication between the two paradigms.

Portability: High-performance in-memory code requires exploiting both low-level hardware details (e.g. # of bitlines/array, SRAM-level instructions) and runtime values, e.g. array dimensions, compute constants. Thus, it is difficult for a single low-level binary to be compatible with all software parameters and future microarchitectures without sacrificing performance.

Our approach: We take a just-in-time (JIT) approach, with the tDFG playing a similar role to PTX virtual assembly for CUDA GPUs. A JIT runtime is in charge of quickly lowering the tDFG "virtual" ISA into in-memory computing commands and managing the transposed data layout. This requires carefully splitting the job between the compiler and the runtime to maintain compatibility while keeping JIT overheads reasonable.

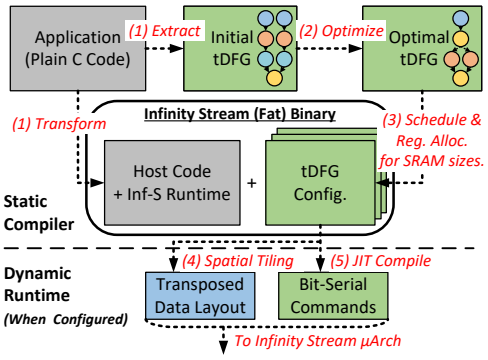

Figure 3: Infinity Stream Workflow Overview

Programmability: Ideally, the system should be easy to program, without programmers writing multiple code versions, worrying about data orchestration, and switching between paradigms. This requires a unified compiler and ISA abstraction, as well as a flexible runtime library and microarchitecture support.

Our approach: The tDFG is constructed purely by the compiler using plain C code. The algorithm and program transformations (e.g. inner vs. outer product) can of course affect the performance, so we discuss programming implications in §3.5. Overall, infinity stream requires only minimal programmer intervention.

Infinity Stream Workflow Overview: Fig 3 summarizes the overall workflow: our static compiler first extracts an initial tDFG from plain C code and optimizes it for compute reuse and less data traffic. The optimal tDFG is scheduled for common SRAM sizes (we use 256×256 and 512×512). This generates a fat binary with multiple tDFG configurations, which reduces the complexity of JIT compilation. At runtime, when an infinity stream region is configured, the runtime dynamically decides the transposed data layout with tiling based on the data size and hardware parameters. The matched version of tDFG is JIT lowered into bit-serial commands. The infinity stream μarch transposes the data and executes the commands to perform in-memory computing.

3 INFINITY STREAM ABSTRACTION

This section shows how the proposed abstraction captures the unique properties of in-memory computing to enable helpful optimizations while simplifying programming complexity.

3.1 Stream Dataflow Graph

We first extract the stream dataflow graph (sDFG) from the program, which embeds memory access patterns as *streams* with associated near-stream computations. We leverage the sDFG as the foundation and later extend it to support in-memory computing.

Stream: The compiler decouples access patterns into streams. E.g. Fig 4(a) contains three load streams `A[i-1]`, `A[i]`, `A[i+1]`, and one store stream `B[i]`, with linear access patterns. Streams may be extracted from outer loops if the access pattern is supported. Irregular access patterns (e.g. `A[B[i]]` and `p=p->next`) are also streams but are inefficient for pure in-memory computing.

Near-Stream Computation: Computation can also be associated with streams. E.g. in Fig 4(b) the reduction is associated with stream `A[i]`. Although the operation is applied to all elements, streams still

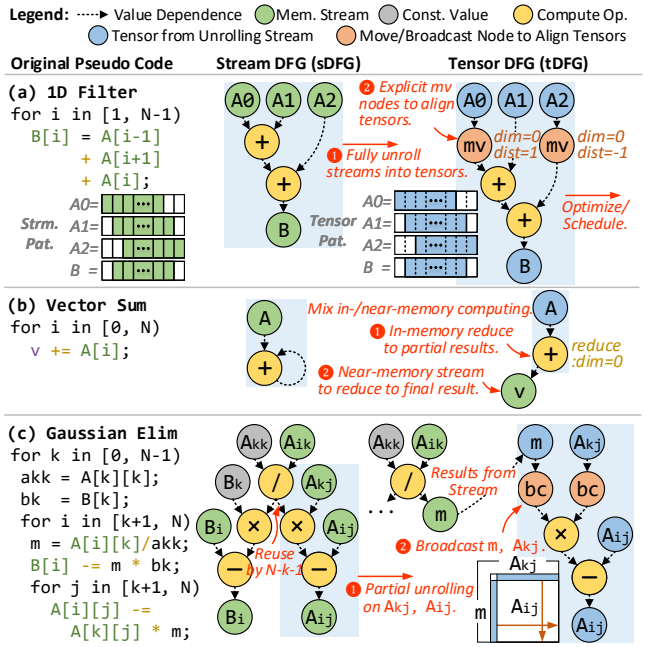

Figure 4: Examples of Infinity Stream Abstractions

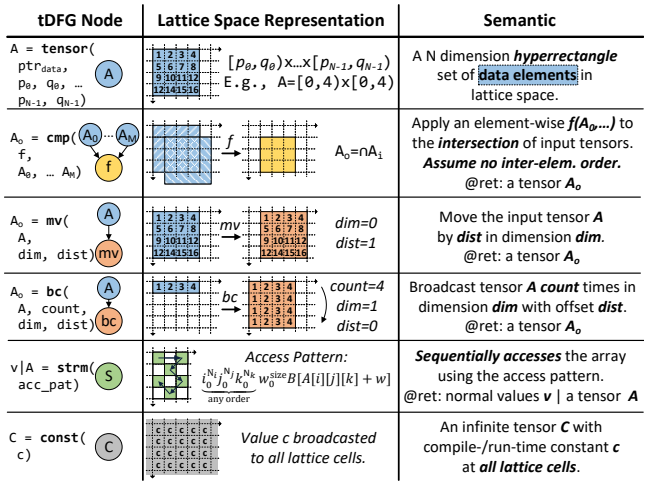

Figure 5: tDFG Node Semantics

implicitly define the access order and preserve sequential semantics. In hardware, each stream (and associated computation) can be independently moved near the L3 if more locality there.

Stream Dataflow Graph: Streams and near-stream computations form the stream DFG. Streams can have dependences: data from the outer loop can be reused by the inner loop, e.g. in Fig 4(c) where the value `m` is reused (N-k-1) times.

3.2 Tensor Dataflow Graph

Intuition: In-memory computing requires unrolling computation across all bitlines. Inspired by this observation, if the domain of the stream is a hyperrectangle (i.e. N-dimensional rectangle) of the data structure, we can fully unroll the stream into a *tensor*. We can then reformulate the computation as a dataflow graph where

the operands are tensors; we call this the tensor DFG (tDFG). Fig 4 shows three example tDFGs, and Fig 5 summarizes all types of tDFG nodes. We now define the key concepts and semantics of the tDFG.

Global Lattice Space: A key feature of the tDFG is the ability to reason about the relative location of different tensors in memory, so that data can be aligned at the bitline level. To enable abstract reasoning about relative locality, we introduce a global lattice space to the tDFG. All tDFG tensors are positioned on an N-dimension global lattice space (its dimensionality is that of the data structure with the highest dimension), shown as the dashed grid in Fig 5. Each lattice cell can hold an arbitrary number of data elements. At runtime, cells are mapped to physical locations, e.g. SRAM bitlines. More importantly, the lattice space serves as a homogeneous coordinate system to abstract away the complex underlying hardware hierarchy, including bitlines, SRAM arrays, banks, NoC, etc. This helps keep the tDFG abstraction portable across platforms.

Tensor: As in Fig 5, a tDFG tensor is a hyperrectangle set of data in the lattice space, denoted by $[p_0, q_0) \times ... \times [p_{N-1}, q_{N-1})$ where p_i and q_i are the start and end coordinate in dimension i. Each data element of a tensor resides in its own lattice cell. An N dimensional array is by itself a tensor with $p_i = 0, q_i = S_i$ where S_i is the array size on dimension i. Unlike streams, tensors do not imply a temporal sequential order but are fully expanded in the lattice space.

Compute with Tensors: A compute node takes one or more input tensors, applies the computation to a domain which is the intersecting hyperrectangle (see Fig 5), and produces an output tensor. The tDFG uses a static single-assignment form (SSA), i.e. nodes always produce a new tensor without overwriting existing ones. There are two key characteristics of tensor computation:

- **Data Parallelism:** Since tensors are fully unrolled, the tDFG does not assume an elementwise order within one tensor computation, exposing massive data parallelism.
- **Data Alignment:** Tensor computation requires operand elements from different tensors to be exactly aligned within the same lattice cell. This captures the data alignment requirement for in-memory computing.

Explicit Tensor Alignment: We introduce two types of node in the tDFG to facilitate explicit tensor alignment, which is crucial to optimize and compile data movement for in-memory computing:

- **Move:** A move node (`mv`) in Fig 5 shifts a tensor along a dimension by a certain distance. E.g. in Fig 4(a), tensor `A[0,N-2)` is moved to the right by 1 to align with `A[1,N-1)`.
- **Broadcast:** To capture reuse spatially, a broadcast node (`bc`) in Fig 5 broadcasts a small reused tensor along the reuse dimension to align with the larger tensor. In Fig 4(c) `A[k,k+1)x[k+1,N)` is broadcast downwards to align with `A[k+1,N)x[k+1,N)`.

Global Bounding Hyperrectangle: Due to the finite hardware resources, not every lattice cell has a valid physical location. we define the global bounding hyperrectangle as the minimal one that contains all involved data structures. semantically, data elements outside the bounding hyperrectangle have undefined values, so data moved or broadcasted outside is discarded. For now, we implicitly assume all data structures are aligned to the origin, but this can be relaxed to placing the array anywhere in the lattice.

Optimizing tDFG: We leverage equality graphs (e-graphs) [47, 48]

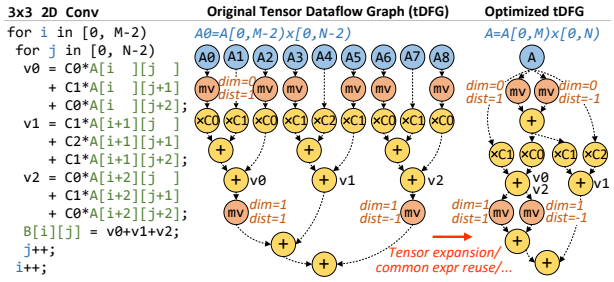

Figure 6: Example of Optimized tDFG

to search for an optimized tDFG. E-graphs are a representation of all possible re-writes to a graph in a compact form, which leverages equality relationships between different re-writes. To construct an e-graph for our case, we start from the initial tDFG, then repeatedly grow the e-graph by applying re-writes and maintaining equivalence points between them. The final tDFG selection is based on architecture-informed cost metrics (e.g. estimated latency of move vs. compute node), and can be exhaustive or terminated early to reduce compile time. Fig 6 shows the initial and optimized tDFG for Fig 4(c). Besides the basic associative, commutative, and distributive rules, two transformations are widely applicable (see the Appendix for a full list of transformation rules):

- **Tensor Expansion**: We can merge two `mv`s with same distance and dimension but on slightly different patterns. In Fig 6, A_0`:[0,M-2)x[0,N-2)` and A_3`:[1,M-1)x[0,N-2)` are both shifted to the right by 1, and can be merged into one `mv` on the expanded tensor `[0,M-1)x[0,N-2)`.
- **Reuse Common Comp.**: We can also reuse common computations. In Fig 6, instead of multiplying by $C0$ four times, we can reuse the result by shifting it to where it is needed in the lattice.

3.3 Hybrid In-/Near- Memory

tDFG is also general and flexible to support hybrid in/near memory execution by embedding streams.

Embedding Streams in tDFG: Some streams/ops in the tDFG are not unrolled into tensors, e.g. alias, non-hyperrectangle accesses, etc. Keeping streams in the tDFG enables data to be read or written in a strided affine pattern or an indirect pattern, providing a better setup for tensor computation (e.g. a stream performs an indirect access and lays out the data in a tensor format). We allow up to three dimensions for affine access and dependent one-level indirect access (see the access pattern in Fig 5). A stream node can produce:

- **Normal Values:** Load and reduce streams generate normal values (non-tensor) consumed by the core or other streams. E.g. the reduction in Fig 4(b) is split into two nodes: a tensor compute node to perform partial in-memory reduction, and a stream node to perform the final reduction, as in-memory computing is inefficient for the final rounds.
- **Tensor Values:** Store streams produce a new tensor with the bounding hyperrectangle of all touched lattice cells. Semantically, this can be as large as the entire accessed array, e.g. an indirect stream updates a subset of the elements. However, in implementation, this is just updating an existing tensor and does not allocate a new one. In Fig 4(c), stream B_i is not unrolled due to low parallelism, and stream `m` writes the division result into a tensor `m`, which is later consumed by in-memory computing.

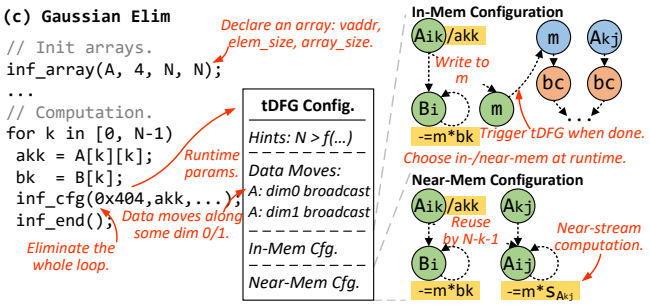

Figure 7: Example of Compiled Infinity Stream Program

Supporting Irregularity: Hybrid in-/near-memory execution enables infinity stream to handle some forms of irregularity, i.e. streams in tDFG can have irregular access patterns (e.g. A[B[i]]). For example, in `kmeans`, in-memory computes the closest centroid for each point using tensor operations, while near-memory performs the indirect update to recalculate centroids' coordinates. For future work, the tDFG can also be extended with control flow and predication to handle control irregularity.

3.4 ISA Interface

Both the sDFG and tDFG for each relevant program region are encoded in the binary, to enable a dynamic choice between near-memory and in-memory respectively. Fig 7 shows the compiled Fig 4(c) with both DFGs and data layout hints.

Infinity Stream Configuration: The `inf_cfg` instruction marks the beginning of infinity stream regions, and passes in the runtime parameters (e.g. constant values). This triggers the runtime library to read in the configuration and configure the microarchitecture (details in §4 and §5). As in prior work [64], near-stream computations are compiled into conventional functions in the native ISA. A pointer to this function is stored in the sDFG.

Layout Hints for Tiling: We add layout hints into the configuration to help the runtime quickly make good decisions about tiling: e.g. which dimensions the array would be shifted along (favoring tiling along those dimensions), as well as which arrays are used for the same computation (and should be bitline-aligned). The compiler generates the layout hints by analyzing the tDFG's data movement patterns. The runtime also requires the array sizes, which are passed in using the `inf_array` API. Fig 7 demonstrates using `inf_array` to declare a 2D array `A[N][N]`, where the infinity stream configuration defines that array `A` is broadcast in both dimensions. The runtime combines this information and picks a suitable data layout to reduce the traffic (see §4.1). Currently, we manually insert `inf_array` calls in the initialization phase.

tDFG Backend Compilation: To generate a tDFG configuration, the backend compiler serializes the tDFG and allocates values to wordlines (once for each SRAM array size in the fat binary). In this work we use a straightforward approach of scheduling instructions in topological order, and using a local register allocation scheme [4]. Though there are few effective registers (e.g. 8 32-bit registers in a 256-wordline SRAM array), no register spilling was observed in the studied workloads. Fusing multiple physical SRAM arrays into a larger virtual array with more registers is possible, but left for future work.

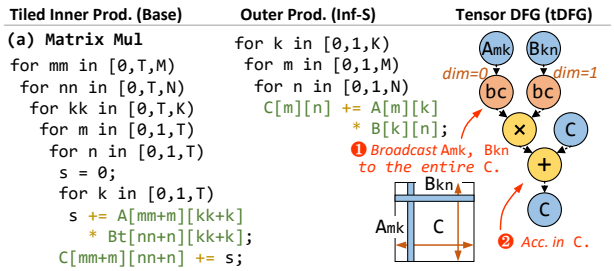

Figure 8: Programming GEMM for Infinity Stream

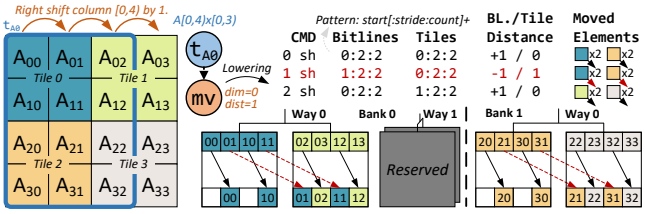

Figure 9: Moving a Tensor in Tiled Layout (View in Color)

3.5 Programming Infinity Stream

Due to its special execution model, programmers face different trade-offs when programming an in-memory system, with tiling and dataflow being the two major design choices.

Tiling: Since in-L3 computing flattens the memory hierarchy, it becomes unnecessary to tile for L1/L2 caches at the programming interface. The runtime will handle the tiling across SRAM arrays using microarchitecture support. E.g., Fig 8 shows the baseline 2-level tiled code for matrix multiplication `mm`, while infinity stream's implementation has no tiling with only 3 loop levels.

Inner vs. Outer Product: Another critical design choice is the dataflow. In-core computing usually favors inner product as it accumulates the result in the register (see Fig 8). However, as in Fig 2, in-memory computing does not handle reduction well as the data parallelism is halved after each round of reduction, and prefers outer product to convert the reduction to element-wise operations. In Fig 8, during each round of `k`, one column of `A[]` and one row of `B[]` is broadcast to the entire `C[]`, followed by multiplication and accumulation. We evaluate both dataflow choices in §8.

Best Practice: Programmers should choose outer product or a similar dataflow that exposes more parallelism for inner loops and move reduction to outer loops. Also, there is no need to tile for private caches as in-memory computing is performed at L3. As in standard practice, programmers should still tile for L3 to provide a suitable working set for in-memory computing.

4 RUNTIME SUPPORT

The tDFG is neutral to hardware details and input sizes to maintain compatibility. Instead, a runtime library manages the transposed data layout, lowers the tDFG into in-memory commands, and decides between in-/near-memory computing, described as follows.

4.1 Transposed Data Layout

The transposed data layout is left to runtime as it requires information that is usually unavailable at compile time, e.g. input sizes, SRAM array sizes, NoC bandwidth, etc.

A trivial data layout would treat the data structure as a 1D array and map elements to contiguous bitlines. However, tensors are often shifted/broadcast along higher dimensions. Therefore, to reduce data traffic across SRAM arrays, the data layout within an SRAM is modified through tiling. Here, a *tile* is defined by the data dimensions mapped to one SRAM array. In Fig 9 we consider a 4-bit-wide SRAM array, where a 4x4 2D software array is split into 4 2x2 tiles, and mapped to SRAM arrays (some SRAMs belong to ways reserved for conventional cache). We only transform the data layout through tiling at the SRAM array level, as it captures most of the traffic reduction benefits, and keeps the mapping between physical address and bitlines simple. Applying further data-layout tiling at a coarser level could further reduce data traffic.

Tiling Constraints: Assume an N-dimensional $S_0 \times ... \times S_{N-1}$ array with L elements per cache line, B bitlines per SRAM array and W SRAM arrays per L3 bank used for in-memory computing. The tile size $T_0 \times ... \times T_{N-1}$ must ensure that:

(1) $\prod_{i=0}^{N-1} T_i = B$: Each tile occupies all bitlines in one SRAM array. This simplifies the logic for intra-tile data movement.
(2) $T_0 \times W \bmod L = 0$: For dimension 0 (continuous in address space), tiled elements at each L3 bank ($T_0 \times W$) aligns with elements per cache line (L). This ensures that each line is mapped to only one L3 bank.

The runtime gets the array's element size and shape from the `inf_array` API, and searches for a valid tile size meeting the constraints. If none is found, the array is not transposed and in-memory computing is disabled. Notice that the array size is not required to align to tile size; boundary tiles with unused bitlines require special handling (see §4.2 and §5). In addition, it checks that the array's innermost dimension aligns to the cache line ($S_0 \bmod L = 0$). Along with constraint 2, this guarantees a transposed cache line is not split across L3 banks, and is still accessible by normal requests (with longer latency to transpose back, see §5). This rarely fails for large arrays, as they are often padded for cache line alignment.

When multiple arrays are used by the same computation, e.g. the input and output array of 2D convolution, the runtime picks one primary array (the output or the reduced array) and uses its tile size for others. Using the same tile sizes eases the complexity to align tensors at runtime.

Tiling Heuristics: The runtime picks one valid tile size using hints in the configuration. Shifts favor a close-to-square tile size, as it keeps most traffic within the same tile. For reduction, a larger tile size on the reduced dimension allows more rounds of in-memory reduction. Broadcast reads favors a smaller innermost tile size if it can spread one row to more L3 banks to avoid the hotspot. When tensors are used for multiple kinds of data movement, we prioritize by the order of reduction, shift, and broadcast, as reduction is usually more expensive due to low compute intensity, while broadcast is inexpensive, as it can reuse the read data. The runtime can pick the best data layout for each program phase. Our heuristic is within 2% of an oracle configuration (see §8).

4.2 JIT Lowering tDFG

The runtime also lowers the tDFG into in-memory commands. In Fig 9, an example `mv` node (right shift columns [0, 3) by 1) is lowered through the following steps.

Algorithm 1: Decompose Tensor

Input: A N-dim tensor $A = [p_0, q_0) \times ... \times [p_{N-1}, q_{N-1})$ where $p_i < q_i$
Input: A list of tile size of each dim $ts = [t_0, ..., t_{N-1}]$
Result: A list of decomposed tensors ret initialized as [ ]
1 **if** $N > 0$ **then** // Decompose dimension 0
2 // head middle tail; 0 a p_0 b c q_0 d
3 $a \leftarrow \lfloor \frac{p_0}{t_0} \rfloor \times t_0$, $b \leftarrow \lfloor \frac{p_0+t_0-1}{t_0} \rfloor \times t_0$ // Align p_0 to tile boundary
4 $c \leftarrow \lfloor \frac{q_0}{t_0} \rfloor \times t_0$, $d \leftarrow \lfloor \frac{q_0+t_0-1}{t_0} \rfloor \times t_0$ // Align q_0 to tile boundary
5 // Recursively decompose remaining dimensions
6 $rs \leftarrow$ Decompose($[p_1, q_1) \times ... \times [p_{N-1}, q_{N-1})$, $[t_1, ..., t_{N-1}]$)
7 **forall** $A' \leftarrow rs$ **do** // Construct final decomposed tensors
8 **if** $b <= c$ **then** // $a \le p_0 < b \le c \le q_0 < d$
9 **if** $a < p_0$ **then**
10 ret += $[p_0, b) \times A'$ // Head interval
11 **if** $b < c$ **then**
12 ret += $[b, c) \times A'$ // Possible middle interval
13 **else**
14 ret += $[a, c) \times A'$ // p_0 aligns with a
15 **if** $c < q_0$ **then**
16 ret += $[c, q_0) \times A'$ // Add possible tail interval
17 **else** // $a = c \le p_0 < q_0 < b = d$
18 ret += $[p_0, q_0) \times A'$ // Same tile, no decomposition
19 **else** // No more dimension to decompose
20 ret += A

1. Tensor Decomposition: As tensors may not align to the tile boundary (e.g. moving a subregion of the array), they are decomposed into smaller ones to separately handle those tiles at the boundary. Alg 1 recursively decomposes an N-D tensor along the tile boundary at each dimension. For the start and end position p_0, q_0 of dimension 0, it identifies their respective tile boundaries $[a, b)$, $[c, d)$ such that $p_0 \in [a, b), q_0 \in [c, d), \{a, b, c, d\} \mod t_0 \equiv 0$ (line 3-4). Depending on the relative positions of p_0 and q_0, it decomposes the 1D tensor $[p_0, q_0)$ into one to three new ones: additional subtensors for the head and/or tail if p_0 and/or q_0 do not align with the tile boundary. For multiple dimensions, we take the cross product of all decomposed tensors (line 8-18). When the tensor aligns with the tile boundary in every dimension, no decomposition is needed.

For example in Fig 9, `A[0,4)x[0,3)` is decomposed into two subtensors A_L`[0,4)x[0,2)` made of full tile 0 and 2, and A_R`[0,4)x[2,3)` made of partial tile 1 and 3. Since dimension 0 is perfectly aligned, the original range $[p_0 = 0, q_0 = 4)$ is kept (line 13). For dimension 1, the range $[p_1 = 0, q_1 = 3)$ means the tail is not aligned ($t_1 = 2 \implies q_1 \mod t_1 \not\equiv 0$). Therefore dimension 1 is decomposed into $[p_1 = 0, 2)$ and $[2, q_1 = 3)$. The cross product between decomposed dimensions 0 and 1 yields two subtensors $[0, 4) \times [0, 2)$ and $[0, 4) \times [2, 3)$.

2. Intra-/Inter-Tile Shifts: Alg 2 lowers a decomposed `mv` node into intra-/inter-tile shift commands. Each shift command takes five arguments: 1) a tensor A, 2) a shift dimension k, 3) a shift mask that selects the bitlines to shift, and 4,5) the inter-/intra-tile shift distances that indicate the direction and number of tiles/bitlines to shift (intra-tile shifts always have 0 inter-tile shift distance). Depending on whether the shift distance aligns with the tile boundary ($d_{intra} == 0$), we may generate an inter-array shift command and optionally an extra intra-array shift command (line 5-12). Notice that not all shift commands will necessarily generate traffic, as the intersection of the shift mask and the tensor may be the empty set. Such shift commands are filtered out later (ommitted in Alg 2).

Algorithm 2: Compile `mv` to Shift Commands

Input: A N-dim tensor $A = [p_0, q_0) \times ... \times [p_{N-1}, q_{N-1})$ where $p_i < q_i$
Input: Tile size t_k of move dimension k and move distance d
Result: A list of shift commands ret initialized as []

1 $d_{inter} \leftarrow \lfloor \frac{\text{abs}(d)}{t_k} \rfloor$ // Inter-tile shift distance
2 $d_{intra} \leftarrow \text{abs}(d) \bmod t_k$ // Intra-tile shift distance
3 $\overline{d}_{intra} \leftarrow t_k - d_{intra}$ // Complement of d_{intra}
4 // `Shift(tensor, dim, mask, inter_tile_dist, intra_tile_dist)`
5 **if** $d > 0$ **then** // Shift forward
6 ret += Shift$(A, k, [0, \overline{d}_{intra}), d_{inter}, d_{intra})$
7 **if** $d_{intra} > 0$ **then**
8 ret += Shift$(A, k, [\overline{d}_{intra}, t_K), d_{inter} + 1, -\overline{d}_{intra})$
9 **else if** $d < 0$ **then** // Shift backward
10 **if** $d_{intra} > 0$ **then**
11 ret += Shift$(A, k, [0, d_{intra}), -(d_{inter} + 1), \overline{d}_{intra})$
12 ret += Shift$(A, k, [d_{intra}, t_K), -d_{inter}, -d_{intra})$

As an example, in Fig 9, shifting A_L`[0,4)x[0,2)` to the right by one requires one intra-tile shift to move the column 0 (`CMD 0`, Alg 2 line 6), and one inter-tile shift to move the column 1 across the tile boundary (`CMD 1`, Alg 2 line 8). Each command has the bitline/tile pattern generated by intersecting the tensor with the shift mask. These patterns are applied to bitlines/tiles, specified using the `start[:stride:count]+` format. E.g. `CMD 1` has bitline pattern `1:2:2` and tile pattern `0:2:2`, therefore shifts bitline 1, 3 of tile 0, 2 (red arrow). These patterns are expanded into masks by the hardware when executed (see §5). Activated wordlines are also encoded, but are omitted in Fig 9 for simplicity. Shift commands also have the bitline/tile distance to determine the destination bitline/tile. Similarly, A_R`[0,4)x[2,3)` is shifted to the right by one intra-tile shift (`CMD 2`, Alg 2 line 6), but requires no inter-tile shift (skipped Alg 2 line 8). The runtime ensures data is not shifted beyond the array boundary by checking the tensor size and the shift distance.

3. Map to L3 Banks: Some commands, e.g. those for boundary tiles, may be skipped by some banks. The runtime intersects the commands' tile pattern and the tiles mapped to each L3 bank. If the intersection is empty, the command can be skipped at that L3 bank. In Fig 9, since `CMD 0` operates on tile 0 (mapped to L3 bank 0) and tile 1 (mapped to L3 bank 1), it is mapped to both L3 banks.

Other tDFG Nodes: Element-wise compute nodes do not move the data and can skip step 2, but still needs step 1 and 3 to handle the boundary tiles and to be mapped to L3 banks. The compute commands also encode the opcode and the wordlines of the operands and result. Reduction nodes are lowered into a sequence of interleaving compute and intra-tile shift commands to fully reduce each tile on the reduced dimension. Broadcast nodes are handled similarly to move nodes, with the broadcast destination encoded.

Synchronization: All commands are synchronous at L3 banks (i.e. do not issue until the previous one finished) except inter-tile shifts, which are considered finished when all data movement within the L3 bank *and* the inter-bank packets are injected into the NoC (but may before they arrive at the destination L3 bank). Therefore, the runtime inserts a sync command between an inter-tile shift command and the consuming command, which serves as a global memory barrier, ensuring that data movements before the sync command are visible to commands after the sync command. (i.e. arrived at the destination bitline). A sequence of pure intra-tile shift and compute commands require no synchronization.

Reducing JIT Overheads: Being on the critical path of offloading, JIT lowering can incur significant overheads. Thus, we co-design the software and hardware for JIT performance:

- **Division of labor:** The static compiler handles register allocation and scheduling (see §3.4), so the JIT compiler only needs to map the scheduled tDFG according to the tiled data layout and lower into bit-serial commands. This is possible by scheduling for common SRAM array sizes (256x256 and 512x512), forming a fat binary similar to CUDA. Note that our fat binary does not expose any microarchitecture beyond the SRAM array sizes, and we believe there will only be a small handful that are useful over many generations of hardware.
- **Memoization:** We reuse JIT results when the same tDFG is re-executed with the same parameters by adding a small hardware cache (see §5) for intermediate reuses and software memorization for longer-term reuses. This is particularly useful for iterative algorithms (e.g. stencils).
- **Array dimension specialization:** While our JIT compiler can handle higher dimensional arrays, we specialize for common 1-3D arrays by leveraging C++ templates. This enables the compiler to unroll the loop and eliminate expensive recursion (e.g. Alg 1 recursively decomposes the tensor according to the tile boundary).

With these optimizations, we reduced JIT lowering time by more than 1000×, and it takes 12% of overall runtime (see §8). We believe additional optimizations could further reduce the overhead, e.g.:

- **Phase overlapping:** We can overlap JIT compiling with the data preparing phase (to fetch and transpose data, see §5), or lowering for future regions as the core is waiting for the current region to finish.
- **Hardware implementation:** We can broadcast commands after step 2 to all L3 banks and let the hardware skip those not applied to its local tiles, eliminating step 3 (the most time-consuming one as it is $O(N_{bank} \times N_{cmd})$) in software.

4.3 In-/Near-Memory Decision

The runtime also decides between in-/near-memory computing by evaluating the following condition:

$$\frac{N_{elem} \times N_{op}}{TP_{core}} > \Sigma_i Lat_{op_i} + N_{node} \times Lat_{JIT} \quad (2)$$

The LHS models the latency of a core at peak throughput, and the RHS captures the in-memory computing delay (first term, no N_{elem}, as computation is fully parallelized) and the JIT time (second term). The compiler generates aggregate information as hints in the configuration, e.g. # of each op, so that the runtime can make a quick decision without analyzing the tDFG. Other platform-specific parameters can be obtained by querying the hardware or profiling offline. This is just a basic and conservative heuristic (assuming peak core performance), but is sufficient for the studied workloads.

5 MICROARCHITECTURE EXTENSIONS

Fig 10 overviews infinity stream's microarchitecture, with stream engines (SE_{core}/SE_{L3}) handling offloaded near-memory streams, layout override tables (LOT) recording transposed data layout, and tensor controllers (TC_{core}/TC_{L3}) executing in-memory commands and synchronizing with the core.

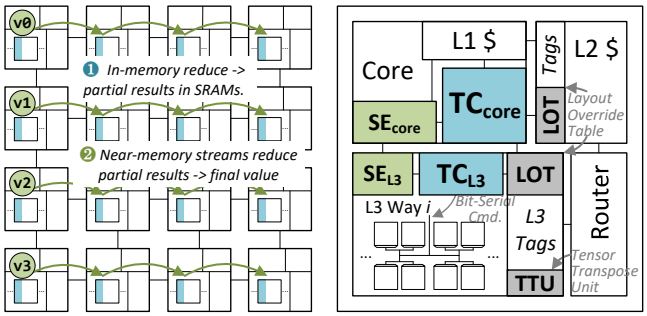

Figure 10: Infinity Stream Microarchitecture

5.1 Near-Memory Computing

We adopt near-memory computing μarch support from NSC [64] to execute streams at the L3 stream engine (SE_{L3}). Streams read/write data directly from L3 banks and forward operands to consuming streams without going back to the core for computing. Streams automatically migrate to the L3 bank where the next data is mapped, with coarse-grained flow control messages (i.e. sync every N cache lines between SE_{core} and SE_{L3}) to reduce coordination.

5.2 In-Memory Computing

During in-memory computing mode, the microarchitecture needs to manage the transposed data layout (LOT and TC_{core}), execute the in-memory commands (TC_{L3}), and synchronize with the core (TC_{core} and TC_{L3}). We assume the SRAM arrays are enhanced to support bit-serial logic and shifts, as well as a buffered H tree to enable efficient broadcast, similar to [15, 17].

Transposed Data Layout: The layout override table (LOT, Table 1) tracks the transposed arrays initialized by the runtime (up to 3D, so higher-dim arrays should have some dimensions fused). It tracks the physical address, as the L2 and L3 caches are indexed by physical addresses. This requires the array to be contiguous in physical address space (with huge pages or special malloc functions). Directly mapping virtual addresses to bitlines is possible by extending the page table and TLB for transposed pages, but is beyond this work.

Map Physical Address ⇔ Bitlines: The LOT essentially overrides how physical addresses are mapped to SRAM arrays. For transposed data structures, the physical address is subtracted by `base` and divided by `size` to get the element index, which is used to find the containing tile and coordinates within that tile. Since tiles are mapped contiguously to SRAM arrays, it is straightforward to locate the actual bitline and wordlines. Reverse mapping from bitlines to physical addresses is similar.

Prepare Transposed Data: Before in-memory computing, TC_{core} prepares the data in transposed format by first issuing flush requests to the L3 cache controller to reserve the cache ways used for in-memory computing (we use 16 ways).

The `trans` field in LOT (initialized to 0) indicates whether the data is currently cached in transposed layout. If `trans=0`, TC_{core} offloads a load stream to fetch the data into transposed format, and sets `trans=2` when finished. During this process, TC_{core} sets `trans=1`, and any core requests to that physical range is blocked. These load streams are executed in SE_{L3} to avoid the traffic overheads between L2 and L3. Our design uses a tensor transpose unit (TTU) to convert between transposed and normal format, similar to prior works [15, 17].

Table 1: Layout Override Table (LOT)

Field	Bits	Description	Field	Bits	Description
`base`	48	Base phys. addr.	`end`	48	End phys. addr.
`size`	8	Element size.	`dim`	2	Array dim (max 3).
S_i	32	Array size (3×).	T_i	32	Tile size (3×).
`wl`	10	Start wordline.	`trans`	2	Transpose state.

Execute Commands: After the data is prepared, TC_{core} sends out commands in a small command cache (2kB) to TC_{L3} at mapped L3 banks. Commands are generated by the runtime (see §4.2) or reused if the same region is executed multiple times. TC_{L3} is a microcontroller to convert the command's bitline and tile pattern to masks for its local tiles and broadcast commands to SRAM arrays. For inter-tile shifts, it generates the control signals to configure the H tree to shift or broadcast the data, and packs the bits into NoC packets if the destination tile is mapped to another L3 bank. For compute commands, it first broadcasts constant operands (if any) to bitlines, and configures the SRAM arrays to perform the bit-serial computation (using algorithms from prior work [17]). Since commands are long latency (n^2+5n for n-bit integer multiply), TC_{L3} can preprocess the next command to hide the processing latency.

Synchronization: For sync commands, TC_{L3} reports to the other TC_{L3} the # of packets sent there since the last sync, and the total sent packets to TC_{core}. Therefore, the receiving TC_{L3} knows how many packets to expect and can report back to TC_{core} when all packets arrived. After hearing back from all TC_{L3}s, TC_{core} checks that # of sent/received packets matches before broadcasting a message to clear the barrier.

Delayed Release of Transposed Data: To release the transposed data, TC_{core} offloads a special store stream to evict data to the memory, which releases the reserved cache ways. To capture the reuse across program regions, e.g. iterative algorithms, TC_{core} delays releasing the data until any of the following conditions:

- Following an in-memory phase, the number of normal requests to the transposed data exceeds a threshold (we use 100k), suggesting that it is now used for in-core/near-mem computing.
- The L3 miss rate exceeds a threshold, suggesting releasing the reserved ways to reduce the pressure on the L3.
- A timer expires (we use 100k cycles).

5.3 Fused In-/Near-Memory Computing

One key advantage of infinity stream is to enable normal core/stream accesses to the transposed data, which allows cores/streams to be *unaware* of the data layout, providing flexibility across paradigms.

Coherence: Tiling constraints in §4.1 guarantees that transposed cache lines are still mapped to a single (but maybe different) L3 bank. Therefore, the coherence state can be tracked in the newly mapped L3 bank, enabling accesses to transposed data structures using normal requests when in-memory computing is not used. Before in-memory computing starts, TC_{core} evicts any dirty copies in private caches to ensure the data in L3 is up-to-date. During in-memory computing, cores are disabled from accessing the data structure by blocking the requests from private caches (setting `trans` in LOT to 1). However, streams at SE_{L3} can still read and write transposed

Table 2: System and μarch Parameters (cy.: cycle)

System	2.0GHz, 8x8 Cores
OOO8 CPU (8-issue)	64 IQ, 72 LQ, 56 SQ+SB 348 Int/FP RF, 224 ROB
Func. Units	8 Int ALU/SIMD (1 cy.) 4 Int Mult/Div (3/12 cy.) 4 FP ALU/SIMD (4 cy.) 4 FP Div (12 cy.)
L1 D/I TLB L2/SE_{L3} TLB	64-entry, 8-way 2k/1k-entry, 16-way, 8 cy.
L1 I/D $ Priv. L2 $	32KB, 8-way, 2 cy. 256KB, 16-way, 16 cy.
Replacement L1 Bingo Pf. L2 Stride Pf.	Bimodal RRIP, $p = 0.03$ 8kB PHT, 2kB region 16 streams, 16 pf./stream
NoC	32B 1 cy. link, 8x8 Mesh 5-stage router, multicast X-Y routing, 16 mem. ctrls
Shared L3 $	20 cycles, MESI Static NUCA, 1kB interleave 256x256 SRAM array (8kB) 5-level H tree, 64B total BW. 16 arrays per way, 18 ways 64 banks, total 144MB
DRAM	3200MHz DDR4 25.6 GB/s
SE_{core}	2kB FIFO, 12 streams
SE_{L3}	768 streams, 64kB buf. 4 cy. compute init. lat.
LOT	16 regions

Table 3: Workloads (BC: Broadcast)

Benchmark	Move	Cmp.	Parameters
stencil1d	Shift	Elem	4M-entry, 10-iter
stencil2d	Shift	Elem	2k×2k, 10-iter
stencil3d	Shift	Elem	512×512×16\ 10-iter
dwt2d	Shift	Elem	2k×2k
gauss_elim	BC	Elem	2k×2k
conv2d	Shift	Elem	2k×2k
conv3d	BC	Elem	H/W=256,K=\ 3×3, I/O=64
mm/in	BC	Reduce	M/N/K=2k
mm/out	BC	Elem	Same
kmeans/in	BC	Reduce	32k-point,dim=128\
kmeans/out	BC	Elem	128-center
gather_mlp/in	BC	Reduce	M=32k,\
gather_mlp/out	BC	Elem	N/K=128

Table 4: PointNet++

Krnl.	$K, N, r, [dims]$
SA1	512, 32, 0.2, [64, 64, 128]
SA2	128, 64, 0.4, [128, 128, 256]
SA3	1, 128, Inf, [256, 512, 1024]
SA4	512, 16, 0.1, [32, 32, 64]
SA5	512, 32, 0.2, [64, 64, 128]
SA6	512, 128, 0.4, [64, 96, 128]
SA7	128, 16, 0.2, [64, 64, 128]
SA8	128, 32, 0.4, [128, 128, 256]
SA9	128, 128, 0.8, [128, 128, 256]
FCx3	1, 1, /, [512, 256, 10]
SSG	SA1 → SA2 → SA3 → FCx3
MSG	[SA4, SA5, SA6] → [SA7, SA8, SA9] → SA3 → FCx3

data, as the dependence between stream and tensor operations is guaranteed through the dataflow graph and synchronization. E.g. the final reduce stream is not offloaded until the partial in-memory reduce is synchronized at TC_{core}. Similarly, if a tensor is generated by a store stream, the dependent in-memory computation will not start until that stream completes.

Context Switch: As in [64], context switches in near-memory computing are delayed until all streams reach a synchronization point (every few cache lines). Similarly, during in-memory computing, context switches are delayed until TC_{core} completes a sync command so that all computation and data movement is committed. The progress of streams (including iteration number) and in-memory computing progress (commands), as well as the LOT, are saved as part of architectural state. The OS may flush transposed data so that LLC space can be reclaimed.

6 IMPLEMENTATION LIMITATIONS

Our implementation of infinity stream has some limitations that can be relaxed in future works: 1. While it is possible to share the L3 to enable in-memory computing in a multi-program scenario, we allow only one thread to reserve the L3 for in-memory computing at a time by locking the LOT. 2. We assume the input data is already tiled to fit in the L3. Otherwise, in-memory computing is disabled. A future work could support automatically tiling at runtime. 3. We currently do not support register spilling because all studied kernels can fit in the available registers. Register spilling can be implemented by a stream writing back and loading from the DRAM.

7 METHODOLOGY

Compiler and Runtime: We extend the open sourced LLVM-based near-stream computing compiler [64] to unroll sDFGs into tDFGs as described in §3. For tDFG optimization, we define the tDFG rewrite rules in the *egg* library [67] to explore the e-graph (see Appendix for details). Optimized tDFGs are serialized back to the x86 backend in LLVM (extended with infinity stream instructions). The compiler inserts calls to a C++ runtime library to JIT compile tDFGs and manage the data layout.

Simulator: We use gem5-20 [45] for execution-driven, cycle-level simulation, extended with partial AVX-512 support. The L3 cache is extended to model the transposed data layout and in-memory bit-serial computation.

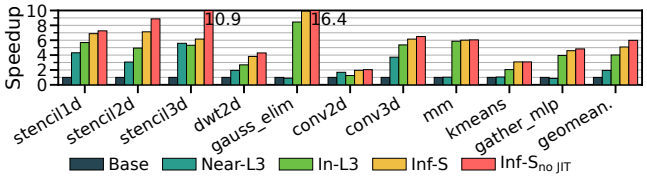

Figure 11: Overall Speedup

Parameters and Configurations: Table 2 lists system parameters. In total, it has 4M bitlines and provides massive parallelism for in-memory computing. The **Base** OOO cores use advanced L1 and L2 prefetchers [6]. For near-memory computing, **Near-L3** offloads streams and the associated computation to SE_{L3}. For infinity stream, we evaluate three configurations:

- **In-L3** invokes a runtime JIT library to manage the data layout and lower tDFG into bit-serial commands to compute with L3 SRAMs, but no near-memory computing support.
- **Inf-S** adds near-memory computing to **In-L3** by offloading sDFG to the SE_{L3}.
- **Inf-$S_{no\ JIT}$** assumes that input and hardware parameters are known, so tDFG is precompiled (no runtime lowering).

Benchmarks: We evaluate 13 dense fp32 OpenMP workloads, compiled with `-O3` and vectorized by AVX-512 for **Base** and **Near-L3**. For infinity stream, a single-thread scalar version is sufficient, as streams are spatially unrolled to all bitlines. Table 2 summarizes the input data sizes and the major data movement (tensor shift vs. tensor broadcast) and computation patterns (element-wise vs. reduction) for each benchmark.

Some benchmarks have different implementations, e.g. inner product vs. outer product for `mm`. We pick the best implementation for each configuration when comparing the performance and energy efficiency, and provide a detailed sensitivity study of the preferences of different paradigms in §8.

We also perform an end-to-end study on PointNet++ [55], a popular hierarchical neural network for point cloud classification and segmentation, in §8.

8 EVALUATION

Overall Performance: Fig 11 shows the overall speedup over **Base**, and Fig 12 shows the NoC utilization and traffic breakdown. The NoC traffic is categorized as the traffic of the coherence control messages (control), the traffic of moving data around (data), and

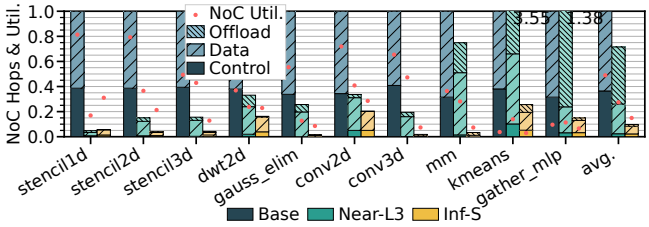

Figure 12: NoC Traffic Breakdown (Bar) and Util. (Dot)

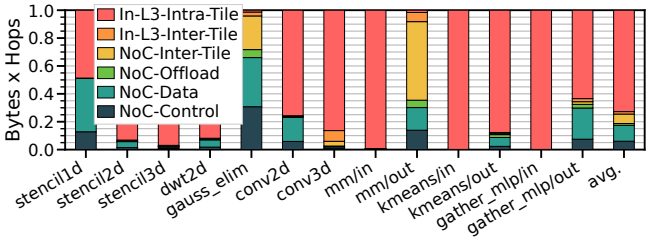

Figure 13: Inf-S Traffic Breakdown

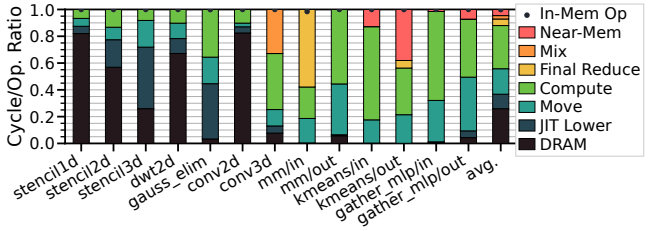

Figure 14: Inf-S Cycle Breakdown

the traffic of all the control messages to manage the offloaded computation, e.g. flow control for streams and synchronization for in-memory computing. For benchmarks with multiple dataflow designs (`mm`, `kmeans`, `gather_mlp`), we pick the best implementation for each configuration (see below for a detailed comparison between dataflow choices). Overall, **Near-L3** achieves 2.0× speedup and 29% traffic reduction by offloading streams near L3 banks, but may hurt the performance as it is unable to capture the reuse; e.g. for `kmeans` **Near-L3** introduces 2.6× extra NoC traffic.

By leveraging massive parallelism in bitlines, **In-L3** achieves 2.1× speedup over **Near-L3**. However, without near-memory computing support, **In-L3** failed to realize the full potential of near-data computing, e.g. in `kmeans`, both aggregation and centroid recomputation are executed by the core and not offloaded. On the other hand, by enabling hybrid in-/near- memory computing, **Inf-S** yields another 24% speedup over **In-L3** (2.6× over **Near-L3**), and 90% NoC Traffic reduction over **Base**. To understand the benefit of traffic reduction, Fig 13 shows the detailed traffic breakdown for **Inf-S**, adding the intra-/inter-tile shift traffic. Notice that some inter-tile shift traffic goes through the NoC if the destination tile is not mapped to the same L3 bank, and is shown separately from NoC-Data as NoC-Inter-Tile. By choosing a reasonable tile size, **Inf-S** converts most of the data movement into intra-tile shifts, leveraging the massive parallelism to shift bitlines within each SRAM array.

Cycle Breakdown: Fig 14 breaks down the cycles of **Inf-S** into transferring and transposing data from/to DRAM (DRAM), lowering tDFG to commands (JIT Lower), moving tensors (Move), bit-serial in-memory computing (Compute), final reduction of the in-memory

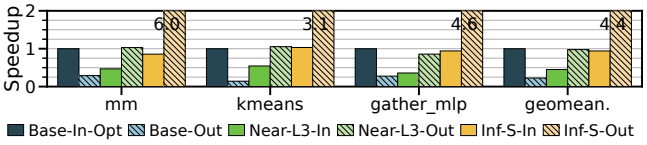

Figure 15: Inner vs. Outer Product Dataflow

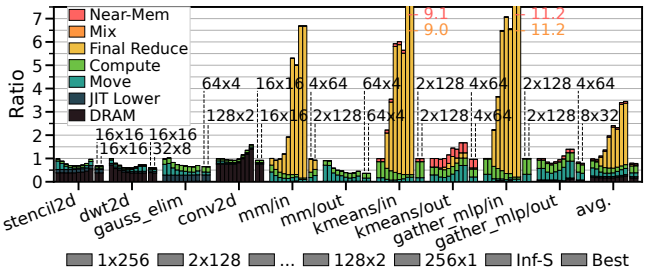

Figure 16: Cycle Breakdown vs. 2D Tile Size

partial results (Final Reduce), hybrid in-/near-memory computing (Mix), as well as pure near-memory computing (Near-Mem). Overall, in-memory computing takes 88% of total cycles, with 26%, 32%, and 19% spent on DRAM transfer, computing, and tensor moving respectively. 4.9% of cycles are spent waiting for the final reduction from near-memory streams, e.g. `mm_inner`. Transposing is cheap when there is high reuse, e.g. `gauss_elim` and `mm`. Dots in Fig 14 indicates the percentage of ops offloaded to bitlines – nearly all computation (99%) are performed in-memory.

JIT Overheads: As shown in Fig 14, JIT lowering contributes 11% of the total runtime, and can be more than 50% when we cannot reuse the lowered commands (51% for `gauss_elim`), or when a high-dimensional tensor is not aligned to the tile size and requires more commands to handle boundary tiles (50% for `stencil3d`). If all input sizes and hardware parameters are known at compile time, the compiler could precompile the tDFG into commands without invoking the JIT runtime. **Inf-S$_{\text{no JIT}}$** in Fig 11 represents such a configuration and yields another 19% speedup over **Inf-S**. The average JIT time is 220us (σ 449us), with `gauss_elim` as the outlier (1616us) as the tensor is shrinking every time. We believe by overlapping JIT lowering with DRAM fetching and command execution, as well as applying more advanced software optimizations, the overheads would be further reduced.

Dataflow Choices: Fig 15 shows the speedup of inner and outer product versions of `mm`, `kmeans`, and `gather_mlp` on different paradigms, normalized to a tiled inner product version for **Base**. As expected, **Base** favors the inner product implementation, as it could accumulate the result in the register file. **Near-L3** generally suffers as it cannot explore the data reuse when offloaded to L3, and favors the outer product version, as the dataflow allows the stream engine to partially recognize the broadcast pattern and save some data traffic (similar to [63]). For **Inf-S**, the outer product is a clear win, as it exposes the maximal data parallelism in the inner loops, and avoids the inefficient in-memory reduction. Overall, it achieves 4.4× speedup over **Base**. Therefore we implement tiled inner product for **Base** and outer product for **Near-L3** and **Inf-S**.

Data Layout: Fig 16 shows the cycle breakdown of all 2D benchmarks with various tiling sizes, annotated with the best and default tile size chosen by the runtime. Similarly, Fig 17 shows the speedup

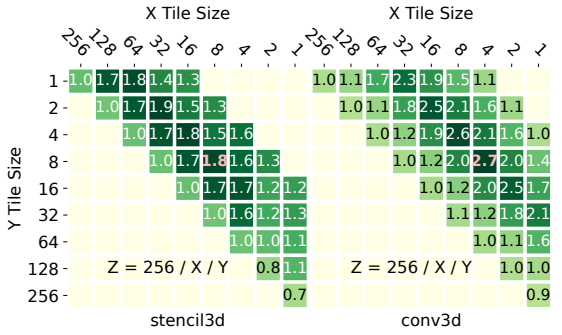

Figure 17: Inf-S Speedup vs. 3D Tile Size (Default as Bold)

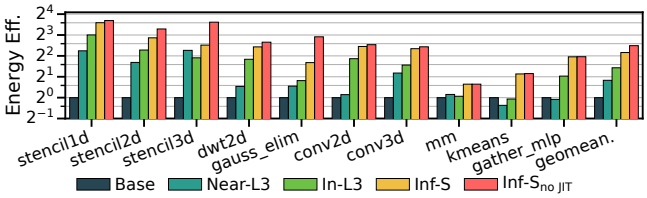

Figure 18: Overall Energy Efficiency

vs. 3D tiling sizes. For benchmarks with shift data movement, e.g. `stencils` and `dwt2d`, picking a balanced tile size (16 × 16 for 2D arrays) usually yields close to optimal performance. When tensors are broadcast, e.g. `gauss_elim` and `mm`, having a smaller innermost tile size helps avoid the hotspot of reading the source row from a single L3 bank. When reduction is needed, a larger tile size at the reduced dimension increases the computation density for in-memory computing and improves the performance. For example, for `kmeans/in` and `gather_mlp/in`, since the size of the reduced dimension is 128, tiling by 128 allows pure in-memory reduction to produce the final results in each SRAM array (hence no Final Reduce bar). Overall, our heuristic achieves within 2% of an oracle, and yields 34% speedup over no tiling (laying the innermost dimension continuously) across all 2D/3D benchmarks.

Energy and Area: The energy breakdown for the SRAM arrays and H tree were obtained from CACTI [7] (22nm) where compute only involves the SRAM arrays while tDFG `mv` node uses both. Fig 18 shows the energy efficiency over **Base**. **Inf-S** yields better energy efficiency for workloads with less reuse by converting NoC traffic into intra-tile shifts. Overall, **In-L3** and **Inf-S** achieve 1.5× and 2.4× energy efficiency over **Near-L3** respectively.

Most of the area overhead comes enhancing existing SRAM caches for compute: additional sense amps and write drivers so every bitline can compute, an extra decoder to read two wordlines simultaneously, and the compute logic. Our area model consists of the overall CPU area reported by McPAT [37] (22nm), the in-memory compute overhead from Neural Cache's [15] die analysis [2], and near-memory support logic [64]. After adding additional logic for in-memory compute (66.75mm^2) and near-memory support (28.16mm^2), the whole chip area overhead is 6.52%.

Case Study of PointNet++: To better understand the benefit of infinity stream on real applications, we perform an end-to-end study on PointNet++ [55], a widely applied hierarchical neural network

[2]We determine the subcircuit area with COFFE [70].

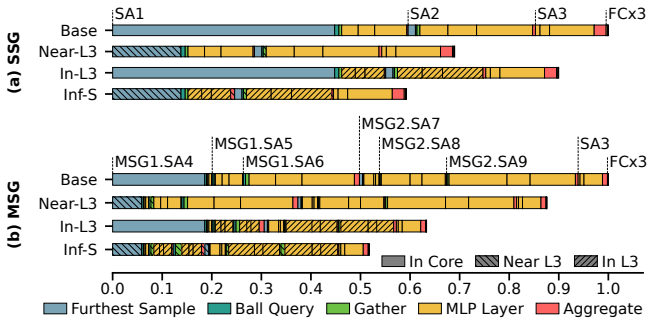

Figure 19: Timeline of PointNet++ SSG/MSG Classifier

for point cloud applications. The basic component of PointNet++ is set abstraction (SA), which consists of the following stages:

- **Furthest Sample:** Iteratively picks K centroids (points) from the input point cloud. For iteration $k+1$, the new centroid is the furthest point from the k prior centroids, with the first one randomly selected.
- **Ball Query:** Searches for N neighbor points within radius r of each centroid. If less than N neighbors are found, the first neighbor is duplicated to fill the remaining spots.
- **Gather:** Performs an indirect gather to collect neighbors' feature vectors. Generates a matrix of $(K \times N) \times D_{in}$ where D_{in} is the dimension of the input feature vector.
- **MLP:** Feeds the gathered matrix into a 3-layer MLP. All layers use ReLU as the activation function. The output matrix is $(K \times N) \times D_{out}$ where D_{out} is the dimension of the last MLP layer.
- **Aggregate:** Reduces the neighbors' feature vectors by taking the max value of each dimension. Outputs a matrix of $K \times D_{out}$.

To perform point cloud classification or segmentation, the authors proposed two network architectures:

- **Single Scale Grouping (SSG):** Multiple SAs are chained with previous output centroids being sampled and grouped by the next SA. This is usually followed by a few fully-connected (FC) layers to produce the final scores for classification.
- **Multiple Scale Grouping (MSG):** To better adapt to various sampling densities, multiple SAs with different radii are applied simultaneously to the input, with their output feature vectors concatenated as the final output. Similar to SSG, this can be chained and followed by more SA/FC layers.

We evaluate both SSG and MSG for classification inference. Table 4 lists the detailed parameters of all SAs and the network structure of SSG/MSG, taken from [55]. The input point cloud contains 4k randomly generated points, normalized to $[0, 1)$.

Fig 19(a) shows the normalized timeline of PointNet++ SSG, broken into different stages with the texture indicating where the computation is executed (in-core, near-L3 cache, or in-L3 SRAM). For SSG, the MLP layers are relatively small with high reuse in the private cache, and with AVX-512 and OpenMP, it only takes 48% of the total runtime in **Base**. This also limits the potential for in-memory computing, e.g. for the first MLP layer in SA1, the amount of data parallelism can only fill 1/4 of the available bitlines, falling short to amortize the long compute latency of bit-serial operation. Therefore, **In-L3** only yields a 10% speedup over **Base**.

On the other hand, furthest sampling takes 46% of the total runtime. This is because it is an iterative algorithm without sufficient

work in each iteration to amortize the synchronization overhead of OpenMP. Also, the working set cannot fit in the private L1 cache, yielding a high miss rate and hurting the performance. These characteristics make it a good candidate for near-memory computing, which achieves 3.1× speedup for sampling, and 31% performance improvement for **Near-L3** over **Base**.

Fig 19(b) shows a similar normalized timeline for PointNet++ MSG. In MSG, sampling is less of a bottleneck as the sampled centroids are shared between SAs within the same MSG. Also, MSG uses larger MLP layers, increasing the amount of data parallelism. This makes in-memory computing more favorable, and **In-L3** and **Near-L3** achieve 37% and 12% speedup over **Base** respectively.

Finally, by leveraging the fused compiler/ISA/runtime abstraction, **Inf-S** can flexibly execute the kernel in the core, near the L3 cache, or in the L3 SRAM. The runtime can avoid offloading small MLP layers to in-memory computing as it hurts the performance, e.g. SA3 and FC layers. Overall, it achieves the highest performance (1.69× and 1.93× over **Base** for SSG and MSG respectively).

*Key Takeaway: **Inf-S** fuses the benefits of in-/near-memory computing, unlocking the full potential of near-data processing.*

9 RELATED WORK

In-memory Computing for CPU Caches: Prior works also augment CPUs for computing in on-chip SRAM caches. Compute cache [1] enables in-memory computation for CPU cache SRAMs, but only supports the less general bit-parallel layout, single-dimension bit-level vector ops (as opposed to multi-dim tensor level). GenPIM adds NVM-based in-memory computing to a general purpose core [30]. Inhale and Sealer enable in-memory encryption at L1 [71, 72]. Neither of the above implements a high-level compiler. Duality cache proposes a bit-serial in-memory approach for CPUs codesigned for CUDA programming [17]. None of these enables portable/transparent support for in-cache computing.

Improving Near-Data Programmability: Various *near-data* approaches have developed techniques to improve programmability. PEI enables programming through instruction intrinsics [3]. SnackNoC [56], Active Routing [28] and Dist-DA [8] specify computation offloads with dataflow graphs. Tesseract uses remote function calls [2]. Livia uses single-cache-line accessing functions [44]. Our work relies on stream abstractions, i.e. long-term memory access patterns, which have been applied both in general purpose processors [49, 61–64] and accelerators architectures [11, 12, 21, 43, 65, 66].

Other near-data programming models are nearly transparent to the programmer. Several are limited to thread-level near-data decisions, programmed with CUDA or OpenMP [27, 46, 52, 59]. Other works enable transparent near-data at a finer grain, but have other limitations, like OmniCompute [53] (only for short RMW instruction chains), EMC [26] (only for address gen.), and Near-stream computing [64]. These cannot be naively applied to enable programmability for PIM, because they do not manage data transposition or guarantee bitline-level alignment.

In-Memory Foundations: Prior works have explored bit-parallel in-memory computing, primarily for bulk bitwise ops [1, 39, 50, 57]. We adopt the bit-serial approach for this work, which enables broader support for more operations, including floating point.

DRAM devices have been the target of both in-memory [19, 20, 24, 38, 54, 57, 69] and near-memory processing [34, 36, 51]. In-DRAM computing provides more parallelism, while in-SRAM computing limits modifications to the CPU. We choose SRAM as the first step due to the trend towards large LLCs and the fact that many algorithms are already tiled for the LLC. However, infinity stream can be applied to both cases, as the abstraction (tDFG) is neutral to the hardware, and the JIT runtime can be extended for in-DRAM computing (e.g. triple-row activation). The memory controller also needs to be extended to support streams. Similar to DMA, coherence could be maintained by evicting cache lines from SRAM.

This work relies heavily on prior efforts to develop the paradigm and circuits of in-SRAM computing devices, including for bit-serial integer [32] and floating point ops [17, 29, 60]. Our contribution is about architecture support for these existing technologies.

Recent works have also proposed offloading to multiple hierarchy levels, leveraging properties like data density (SISA [9]), cache presence (Livia [44]), or offline analysis (MLILP [16]). None of them enable portable targeting of in-memory computing from a general purpose language.

Domain-Specialization: A variety of prior in-memory accelerators are domain-specialized. Many focus on ML [10, 15, 23, 38, 40, 58], while others target graph processing, mining, and physics simulation [2, 9, 13, 25]. Many broader workloads are prime candidates for in-memory computation with infinity stream. For example, several key data center workloads have been adapted to bitvector parallelism. BitWeaving's [41] database column scan produces a comparison bitmask by organizing data to facilitate bit-serial digital comparison. BitFunnel [22] filters documents with a bloom filter, independently computed by determining the hash indices in memory and constructing the bitvector near memory.

10 CONCLUSION

Infinity stream is a new approach that makes in-memory computing programmer-friendly: We proposed an execution model that fuses in-/near-memory, using an IR called the tensor dataflow graph (tDFG) to capture parallelism, reuse, and layout optimizations; we built an optimizing compiler and JIT-approach to enable long-term portability without sacrificing performance, with a microarchitecture that transparently orchestrates data management and performs data-layout transforms at runtime. Our optimizations provide integer-factor improvement for data processing for only a modest area overhead. More broadly, we believe that rethinking how to compute throughout the memory hierarchy will be critical for enabling extreme system scaling.

ACKNOWLEDGMENTS

This work was supported by funding from NSF grants CCF-1751400 and CCF-2200831. We sincerely thank our shepherds and the anonymous reviewers for their insights and suggestions.

A tDFG OPTIMIZATION

Here we discuss the rewrite rules and equality-graph approach to optimizing the tDFG.

Intuition: A unique aspect of optimizing the tDFG is the need

to reason about the tensor domains (i.e. the hyperrectangle in lattice space). For example, two same element-wise computations on tensor `A[1, n)` and `A[0, n-1)` can be merged into a single computation on `A[0, n)`, provided that the tensor size information is correctly tracked after we slightly expanded the computed tensor. This cuts the computation by half. More generally, there is a large transformation space with many equivalent tDFGs producing the same result, and the compiler needs to efficiently search for the optimal tDFG with less data traffic and computation. We first introduce the tDFG equivalence rules used to rewrite the tDFG, followed by an optimized example and details in our implementation.

tDFG Equivalence Rules: We define two tDFG nodes to be equivalent if they represent the same result and share the same domain in the lattice space. To transform the tDFG, we now formalize the tDFG equivalence rules, with these notations:

- $\mathbb{T}, \mathbb{C}, \mathbb{M}, \mathbb{B}$: Tensor, compute, move, and broadcast node respectively, with their definition and semantics in Fig 5 (page 4). Note that all these nodes produce a *tensor*, while $\mathbb{T}$ constructs the input tensor from the input array.
- A, B, C: Arbitrary tensors in the tDFG, e.g. compute, move, broadcast node.
- i, j: Operated dimension, e.g. move, broadcast.
- p_i, q_i: Range of the i^{th} dimension $[p_i, q_i)$.
- f: Computation applied to input tensors.

As a simple example, Eq. 3a defines the associative rule for compute node, when the operation f is associative by itself, i.e. $f(f(a,b),c) \Leftrightarrow f(a,f(b,c))$. Similarly, Eq. 3b defines the commutative rule for compute node when the operation f is commutative, e.g. addition, multiplication. We can also define the distributive rule similar to $a \times (x+y) \Leftrightarrow a \times x + b \times y$ (Eq. 3c).

$$\mathbb{C}(f, \mathbb{C}(f, A, B), C) \Leftrightarrow \mathbb{C}(f, A, \mathbb{C}(f, B, C)) \tag{3a}$$

$$\mathbb{C}(f, A, B) \Leftrightarrow \mathbb{C}(f, B, A) \tag{3b}$$

$$\mathbb{C}(f, \mathbb{C}(g, A), \mathbb{C}(g, B)) \Leftrightarrow \mathbb{C}(g, \mathbb{C}(f, A, B)) \tag{3c}$$

Exchanging Compute and Move/Broadcast: Eq. 4a defines the commutative rule to exchange a unary compute node and a move node. Recall that a move node shifts the tensor along a certain dimension by some distance in the lattice space. Therefore, the move operation can happen before or after the computation, i.e. it is commutative with compute nodes. Similarly, when the compute node takes multiple operands, a move node is applied to every input tensor. Also, Eq. 4b shows the commutative rule for a compute node and a broadcast node.

$$\mathbb{C}(f, \mathbb{M}(A, i, dist)) \Leftrightarrow \mathbb{M}(\mathbb{C}(f, A), i, dist) \tag{4a}$$

$$\mathbb{C}(f, \mathbb{B}(A, i, dist, cnt)) \Leftrightarrow \mathbb{B}(\mathbb{C}(f, A), i, dist, cnt) \tag{4b}$$

Expanding and Shrinking Tensor: To reuse common computation results, it may be necessary to expand a tensor. For example, $\mathbb{C}(f, \mathbb{T}(1, N))$ and $\mathbb{C}(f, \mathbb{T}(0, N))$ share common results on the domain $[1, N)$. However, they are not equivalent as the first computation is applied to a slightly smaller tensor. If we can expand the first tensor to $\mathbb{T}(0, N)$, we can reduce the operations from $2N-1$ to N.

To maintain equivalence, an expanded tensor must be later shrunk to the original domain. Therefore, we introduce a shrink node, $\mathbb{S}$, which resizes the tensor along dimension i to have a new domain $[p_i, q_i)$.

Putting these together, Eq. 5 shows the rule to expand a smaller tensor of size $[p_i, q_i)$ in the i^{th} dimension into a larger tensor of size $[p_i', q_i')$, where $p_i' <= p_i$ and $q_i' >= q_i$. The shrink node returns the output tensor to the original domain, hence it is equivalent to the original tensor. Shrink nodes are only for tracking the tensor size information, and are lowered to a nop by the JIT compiler (similar to how the ϕ nodes are not lowered to instructions in SSA IR [35]). We omit shrink nodes in the paper for simplicity, as they are only needed during optimization.

$$\mathbb{T}(..., p_i, q_i, ...) \Leftrightarrow \mathbb{S}(i, p_i, q_i, \mathbb{T}(..., p_i', q_i', ...)) \quad \text{where } p_i' <= p_i, q_i' >= q_i \tag{5}$$

Exchanging Shrink and Other Nodes: A shrink node by itself is not sufficient to unlock the optimization opportunities in the tDFG. We need to define how it interacts with other tDFG nodes. Eq. 6a is a straightforward rule that two shrink nodes on different dimensions are commutable. When they operate on the same dimension, we can combine them into a single shrink node by taking the intersection, as in Eq. 6b.

$$\mathbb{S}(i, p_i, q_i, \mathbb{S}(j, p_j, q_j, A)) \Leftrightarrow \mathbb{S}(j, p_j, q_j, \mathbb{S}(i, p_i, q_i, A)) \quad \text{when } i \neq j \tag{6a}$$

$$\mathbb{S}(i, p_i, q_i, \mathbb{S}(i, p_i', q_i', A)) \Leftrightarrow \mathbb{S}(i, \max(p_i, p_i'), \min(q_i, q_i'), A) \tag{6b}$$

Similarly, shrink node and move node on different dimensions are commutable (Eq. 7a). If they are on the same dimension, we can also apply a shrink node on the moved tensor with the shifted domain $[p_i + dist, q_i + dist)$.

$$\mathbb{M}(\mathbb{S}(i, p_i, q_i, A), j, dist) \Leftrightarrow \mathbb{S}(i, p_i, q_i, \mathbb{M}(A, j, dist)) \quad \text{when } i \neq j \tag{7a}$$

$$\mathbb{M}(\mathbb{S}(i, p_i, q_i, A), i, dist) \Leftrightarrow \mathbb{S}(i, p_i + dist, q_i + dist, \mathbb{M}(A, i, dist)) \tag{7b}$$

This also applies to broadcast node and shrink node: they are commutable if on different dimension (Eq. 8a). When they are on the same dimension, we can combine them by directly broadcasting to the shrunken region.

$$\mathbb{B}(\mathbb{S}(i, p_i, q_i, A), j, dist, cnt) \Leftrightarrow \mathbb{S}(i, p_i, q_i, \mathbb{B}(A, j, dist, cnt)) \quad \text{when } i \neq j \tag{8a}$$

$$\mathbb{S}(i, p_i, q_i, \mathbb{B}(A, i, dist, cnt)) \Leftrightarrow \mathbb{B}(A, i, p_i, q_i - p_i) \tag{8b}$$

Finally, a shrink node is also commutable with the compute node (Eq. 9).

$$\mathbb{S}(i, p_i, q_i, \mathbb{C}(f, A)) \Leftrightarrow \mathbb{C}(f, \mathbb{S}(i, p_i, q_i, A)) \tag{9}$$

Optimization Example: Fig 20 shows an example of applying our rewrite rules to discover opportunities for reuse. The original tDFG first moves the input tensor `A` left and right by one before applying a constant element-wise multiply to both tensors. Since in-memory

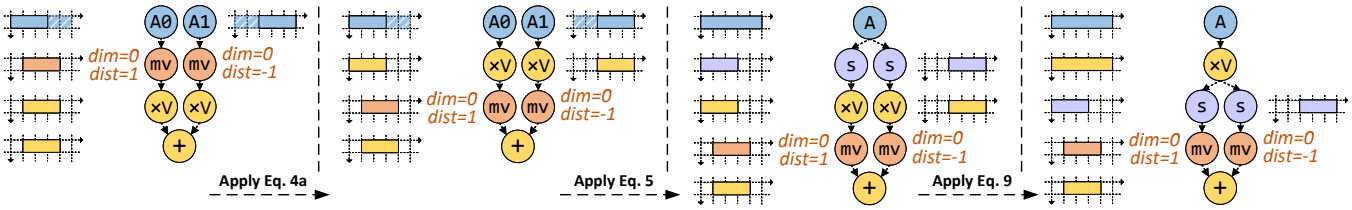

Figure 20: Example of Applying Rewrites

processing applies element-wise functions to all elements, we can save a redundant compute by first performing the computation on the entirety of tensor A before realigning the result.

We begin with the original tDFG. By rule 4a, we can commute the move and compute nodes.

$$\mathbb{C}(+, \mathbb{C}(\times V, \mathbb{M}(\mathbb{T}(0, n-2), 0, 1)), \mathbb{C}(\times V, \mathbb{M}(\mathbb{T}(2, n), 0, -1)))$$

$$\xrightarrow{\text{Eq. } 4a} \mathbb{C}(+, \mathbb{M}(\mathbb{C}(\times V, \mathbb{T}(0, n-2)), 0, 1), \mathbb{M}(\mathbb{C}(\times V, \mathbb{T}(2, n)), 0, -1))$$

We can expand the two tensor $\mathbb{T}$s to the entire domain of array A with rule 5. By commuting the shrink $\mathbb{S}$ nodes and compute $\mathbb{C}$ nodes with rule 9, we can discover a common subexpression, indicating there is an opportunity for compute reuse.

$$\xrightarrow{\text{Eq. } 5} \mathbb{C}(+, \mathbb{M}(\mathbb{C}(\times V, \mathbb{S}(0, 0, n-2, \mathbb{T}(0, n))), 0, 1), \mathbb{M}(\mathbb{C}(\times V, \mathbb{S}(0, 2, n, \mathbb{T}(0, n))), 0, -1))$$

$$\xrightarrow{\text{Eq. } 9} \mathbb{C}(+, \mathbb{M}(\mathbb{S}(0, 0, n-2, \mathbb{C}(\times V, \mathbb{T}(0, n))), 0, 1), \mathbb{M}(\mathbb{S}(0, 2, n, \mathbb{C}(\times V, \mathbb{T}(0, n))), 0, -1))$$

Fig 6 (page 5) shows a more complicated example of optimized tDFG. To see how the equivalence rules rewrite the program, first expand all the tensors to the full array, and exchange the shift nodes to the final output of the tDFG (ommitted in Fig 6). Use Eq. 3b and Eq. 3a to add v0 and v2 together. Since we are multiplying by a constant C_0 and C_1, we can use distributive rule to swap the addition and multiplication. The optimized tDFG reuses the computated results and avoids unnecessary data movements.

Equality Graphs: We leverage equality graphs (e-graphs) to efficiently search the optimal tDFG in the design space. Equality graphs represent all possible rewrites of an expression tree. Given a rewrite rule $e_1 \rightarrow e_2$ for two expressions e_1, e_2, an e-graph will apply it to all matches in its underlying expression tree. These *nondestructive* updates are performed by marking e_1 and e_2 as equivalent. Given a set of rewrite rules, all possible permutations of the original expression tree are discovered by continuously applying them. The final tDFG selection is based on architecture-informed cost metrics combining the estimated latency of move vs. compute node, the amount of moved/broadcast data, as well as the number of computations.

REFERENCES

[1] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. Compute caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[3] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *2015 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[5] Bahar Asgari, Ramyad Hadidi, Jiashen Cao, Da Eun Shim, Sung-Kyu Lim, and Hyesoon Kim. Fafnir: Accelerating sparse gathering by using efficient near-memory intelligent reduction. In *2021 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[6] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad. Bingo spatial data prefetcher. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[7] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. Cacti 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2):14, 2017.

[8] Saambhavi Baskaran, Mahmut Taylan Kandemir, and Jack Sampson. An architecture interface and offload model for low-overhead, near-data, distributed accelerators. In *2022 IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[9] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, Juan Gómez Luna, Jakub Golinowski, Marcin Copik, Lukas Kapp-Schwoerer, Salvatore Di Girolamo, Nils Blach, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. Sisa: Set-centric instruction set architecture for graph mining on processing-in-memory systems. In *2021 IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[10] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: a novel processing-in-memory architecture for neural network computation in reram-based main memory. In *2016 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[11] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. Polygraph: Exposing the value of flexibility for graph processing accelerators. In *2021 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[12] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. Towards general purpose acceleration by exploiting common data-dependence forms. In *2019 IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[13] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. Graphh: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(4):640–653, 2018.

[14] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos. The mondrian data engine. In *2017 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[15] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaaauw, and Reetuparna Das. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *2018 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[16] Daichi Fujiki, Alireza Khadem, Scott Mahlke, and Reetuparna Das. Multi-layer in-memory processing. In *2022 IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[17] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. Duality cache for data parallel acceleration. In *2019 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[18] Daichi Fujiki, Xiaowei Wang, Arun Subramaniyan, and Reetuparna Das. In-/near-memory computing. *Synthesis Lectures on Computer Architecture*, 16:1–140, 08 2021.

[19] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. Computedram: In-memory compute using off-the-shelf drams. In *2019 IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[20] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. Fracdram: Fractional values in off-the-shelf dram. In *2022 IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[21] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. A programmable, energy-minimal dataflow compiler and architecture. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 546–564. IEEE, 2022.

[22] Bob Goodwin, Michael Hopcroft, Dan Luu, Alex Clemmer, Mihaela Curmei, Sameh Elnikety, and Yuxiong He. Bitfunnel: Revisiting signatures for search. In *2017 International ACM SIGIR Conference on Research and Development in Information Retrieval (SGIR)*.

[23] Saransh Gupta, Mohsen Imani, Harveen Kaur, and Tajana Simunic Rosing. Nnpim: A processing in-memory architecture for neural network acceleration. *IEEE Transactions on Computers*, 68(9):1325–1337, 2019.

[24] Nastaran Hajinazar, Geraldo F. Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. Simdram: A framework for bit-serial simd processing using dram. In *2021 ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[25] Bagus Hanindhito, Ruihao Li, Dimitrios Gourounas, Arash Fathi, Karan Govil, Dimitar Trenev, Andreas Gerstlauer, and Lizy John. Wave-pim: Accelerating wave simulation using processing-in-memory. In *2011 International Conference on Parallel Processing (ICPP)*.

[26] Milad Hashemi, Eiman Ebrahimi, Onur Mutlu, Yale N Patt, et al. Accelerating dependent cache misses with an enhanced memory controller. In *2016 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[27] Kevin Hsieh, Eiman Ebrahim, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems. In *2016 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[28] Jiayi Huang, Ramprakash Reddy Puli, Pritam Majumder, Sungkeun Kim, Rahul Boyapati, Ki Hwan Yum, and Eun Jung Kim. Active-routing: Compute on the way for near-data processing. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[29] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. Floatpim: In-memory acceleration of deep neural network training with high precision. In *2019 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[30] Mohsen Imani, Saransh Gupta, and Tajana Rosing. Genpim: Generalized processing in-memory to accelerate data intensive applications. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[31] Mohsen Imani, Saikishan Pampana, Saransh Gupta, Minxuan Zhou, Yeseong Kim, and Tajana Rosing. Dual: Acceleration of clustering algorithms using digital-based processing in-memory. In *2020 IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[32] Supreet Jeloka, Naveen Bharathwaj Akesh, Dennis Sylvester, and David Blaauw. A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory. *IEEE Journal of Solid-State Circuits*, 2016.

[33] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. In *2016 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[34] C. Kozyrakis, J. Gebis, D. Martin, S. Williams, I. Mavroidis, S. Pope, D. Jones, D. Patterson, and K. Yelick. Vector IRAM: A Media-oriented Vector Processor with Embedded DRAM. In *12th Hot Chips Conference*, August 2000.

[35] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2004 International Symposium on Code Generation and Optimization (CGO)*.

[36] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, Seongil O, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. Hardware architecture and software stack for pim based on commercial dram technology. In *2021 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[37] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *2009 IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[38] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. Drisa: A dram-based reconfigurable in-situ accelerator. In *2017 IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[39] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *2016 ACM/EDAC/IEEE Design Automation Conference (DAC)*.

[40] Weitao Li, Pengfei Xu, Yang Zhao, Haitong Li, Yuan Xie, and Yingyan Lin. Timely: Pushing data movements and interfaces in pim accelerators towards local and in time domain. In *2020 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[41] Yinan Li and Jignesh M. Patel. Bitweaving: Fast scans for main memory data processing. In *2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[42] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao. Processing-in-memory for energy-efficient neural network training: A heterogeneous approach. In *2018 IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[43] Sihao Liu, Jian Weng, Dylan Kupsh, Atefeh Sohrabizadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Rishabh Mani, Lucheng Zhang, Jason Cong, and Tony Nowatzki. Overgen: Improving fpga usability through domain-specific overlay generation. In *2022 IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[44] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sanchez, and Nathan Beckmann. Livia: Data-centric computing throughout the memory hierarchy. In *2020 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[45] Jason Lowe-Power and et al. The gem5 simulator: Version 20.0+. *arXiv:2007.03152*, 2020.

[46] R. Nair and et. al. Active memory cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development*, 59(2/3), 2015.

[47] Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford, CA, USA, 1980. AAI8011683.

[48] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In *2005 International Conference on Term Rewriting and Applications (RTA)*.

[49] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. Stream-dataflow acceleration. In *2017 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[50] Jisung Park, Roknoddin Azizi, Geraldo F. Oliveira, Mohammad Sadrosadati, Rakesh Nadig, David Novo, Juan Gómez-Luna, Myungsuk Kim, and Onur Mutlu. Flash-cosmos: In-flash bulk bitwise operations using inherent computation capability of nand flash memory. In *2022 IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[51] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent ram. *IEEE Micro*, 17(2):34–44, 1997.

[52] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das. Scheduling techniques for gpu architectures with processing-in-memory capabilities. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*.

[53] Ashutosh Pattnaik, Xulong Tang, Onur Kayiran, Adwait Jog, Asit Mishra, Mahmut T Kandemir, Anand Sivasubramaniam, and Chita R Das. Opportunistic computing in gpu architectures. In *2019 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[54] Xiangjun Peng, Yaohua Wang, and Ming-Chang Yang. Chopper: A compiler infrastructure for programmable bit-serial simd processing using memory in dram. In *2023 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[55] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In *2017 International Conference on Neural Information Processing Systems (NIPS)*.

[56] Karthik Sangaiah, Michael Lui, Ragh Kuttappa, Baris Taskin, and Mark Hempstead. Snacknoc: Processing in the communication layer. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[57] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *2017 IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[58] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *2016 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[59] Po-An Tsai, Changping Chen, and Daniel Sanchez. Adaptive scheduling for systems with asymmetric memory hierarchies. In *2018 IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[60] Jingcheng Wang, Xiaowei Wang, Charles Eckert, Arun Subramaniyan, Reetuparna Das, David Blaauw, and Dennis Sylvester. A 28-nm compute sram with bit-serial logic/arithmetic operations for programmable in-memory vector computing. *IEEE Journal of Solid-State Circuits*, 55(1):76–86, 2020.

[61] Zhengrong Wang, Christopher Liu, and Tony Nowatzki. Infinity stream: Enabling transparent and automated in-memory computing. *IEEE Computer Architecture Letter (CAL)*, 2022.

[62] Zhengrong Wang and Tony Nowatzki. Stream-based memory access specialization for general purpose processors. In *2019 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.
[63] Zhengrong Wang, Jian Weng, Jason Lowe-Power, Jayesh Gaur, and Tony Nowatzki. Stream floating: Enabling proactive and decentralized cache optimizations. In *2021 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
[64] Zhengrong Wang, Jian Weng, Liu Sihao, and Tony Nowatzki. Near-stream computing: General and transparent near-cache acceleration. In *2022 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
[65] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki. Dsagen: Synthesizing programmable spatial accelerators. In *2020 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.
[66] J. Weng, S. Liu, Z. Wang, V. Dadu, and T. Nowatzki. A hybrid systolic-dataflow architecture for inductive matrix algorithms. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
[67] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. In *Proceedings of the ACM on Programming Languages (POPL)*, 2021.
[68] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. Spacea: Sparse matrix vector multiplication on processing-in-memory accelerator. In *2021 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
[69] Xin Xin, Youtao Zhang, and Jun Yang. Roc: Dram-based processing with reduced operation cycles. In *2019 ACM/IEEE Design Automation Conference (DAC)*.
[70] Sadegh Yazdanshenas, Kosuke Tatsumura, and Vaughn Betz. Don't forget the memory: Automatic block ram modelling, optimization, and architecture exploration. In *2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.
[71] Jingyao Zhang, Hoda Naghibijouybari, and Elaheh Sadredini. Sealer: In-sram aes for high-performance and low-overhead memory encryption. In *2022 ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*.
[72] Jingyao Zhang and Elaheh Sadredini. Inhale: Enabling high-performance and energy-efficient in-sram cryptographic hash for iot. In *2022 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.
[73] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. Graphq: Scalable pim-based graph processing. In *2019 IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

Received 2022-10-20; accepted 2023-01-19