Infinity Stream: Enabling Transparent and Automated In-Memory Computing

Zhengrong Wang, Christopher Liu, and Tony Nowatzki

Abstract—Although in-memory computing is promising to alleviate the data movement bottlenecks by parallelizing computation across memory bitlines, key challenges from its unique execution model remain unsolved: Automatically parallelizing sequential programs; Dynamically managing and aligning data in transposed layout required for bit-serial logic; Mixing in/near-memory computing. These challenges should be solved transparently to maintain portability without exposing hardware details to programmers. In this work, we introduce a novel intermediate representation – tensor dataflow graph (tDFG) – with tensor nodes representing the spatially unrolled data across bitlines, and explicit move nodes to align operands in the same bitline, which helps the compiler optimize for massive parallelism and data layout. To maintain transparency and portability, we directly embed tDFG in the ISA, which is lowered into bit-serial operations at runtime to hide the hardware details. Evaluated on cycle-accurate simulator across various data-processing workloads, our approach achieves 4.5× speedup and 52% traffic reduction over a state-of-the-art near-memory computing technique.

Index Terms—Stream-Based ISAs; Programmer-Transparent Acceleration; In-Memory Computing

As systems scale up, growing data movement bottlenecks force architects to shift from traditional core-centric computing to a memory-centric view. To reduce communication cost, near-memory computing adds specialized hardware near memory banks to decouple computation from core pipelines. In-memory computing further pushes this offloading to the extreme, providing massive parallelism (e.g. bit-serial [1]–[3] or analog [5]). Our focus is bit-serial compute in the L3 SRAM.

Although promising, in-memory computing's applicability is currently limited by challenges within its unique execution model:

- **Parallelism**: Distributing computation to bitlines requires reasoning about massive vector parallelism, which is challenging for languages/ISAs with sequential semantics.
- **Data Layout**: Bit-serial logic requires transposing data and enforcing bitline-alignment between data structures. A suitable data layout is critical to reduce data transfer traffic and realize the potential of in-memory computing.
- **Unification**: Sometimes it's more beneficial to split the computation between in/near-memory computing, e.g. in-memory reduction to quickly produce partial results, which are reduced to the final value by near-memory computing. This requires a unified abstraction across paradigms.
- **Transparency**: Programmers should be abstracted from the parallelization strategy, data layout, underlying hardware details, etc. Further, the compiled binary should be portable across input data sizes and hardware platforms.

Existing techniques have not fully addressed these, as they are either domain specific (e.g. [2] for CNN) or require program rewriting (e.g. [3]). New compiler/ISA abstractions are needed.

Our inspiration comes from previous efforts to enable transparent and general near-memory computing by encoding long-term memory access patterns (i.e. streams) and associated near-stream computation in the ISA [6]–[8]. Streams may be scheduled in-core or near-memory depending on the data locality. However, streams are inherently sequential and do not have a representation for data movement and alignment among bitlines for bit-serial logics.

To solve this problem, our insight is that in target workloads, streams usually access rectangular subregions of the multidimensional array, and corresponding computations have very simple parallelism and reuse patterns. We refer such subregions as *tensors*, and the near-stream computation and dependences now apply to the entire tensor instead of individual elements, forming a *tensor dataflow graph* (tDFG). We use tDFG as both a compiler IR and an ISA abstraction to solve the above challenges:

- **Parallelism**: tDFG effectively exposes the massive parallelism in the original sequential program, enabling the compiler to optimize for in-memory computing.
- **Data Layout**: tDFG also explicitly encodes the data movement operations to align tensor operands in the same bitline for computing, which helps the system choose a suitable data layout to align data structures and reduce overall traffic.
- **Unification**: Both tensors (parallel) and streams (sequential) can be expressed in the tDFG, enabling flexible offloading strategies. 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 the partial results from each SRAM array (tensor to stream).
- **Transparency**: As an ISA abstraction, tDFG is hidden from programmers, as it is constructed by the compiler and is neutral to hardware parameters and data input sizes for portability.

Leveraging the tDFG, we propose *infinity stream*, which transparently and flexibly enables systems to offloading to either in/near memory, fusing these two paradigms. Evaluated with LLVM and gem5 [4], it achieves 4.5× speedup and 52% traffic reduction over near-memory only [8]. Specifically, our contributions are:

- tDFG abstraction IR and ISA to enable transparent and portable in-memory computing, with detailed evaluation.
- Compiler techniques to build tDFGs from unmodified source, and optimizations for parallelism and reducing data movement.
- μarch to dynamically lower tDFGs into bit-serial ops, manage transposed data layout and parallelized execution.

- *The authors are with University of California, Los Angeles, CA 90095 USA. E-mail: {seanzw,chrisliu,tjn}@cs.ucla.edu*

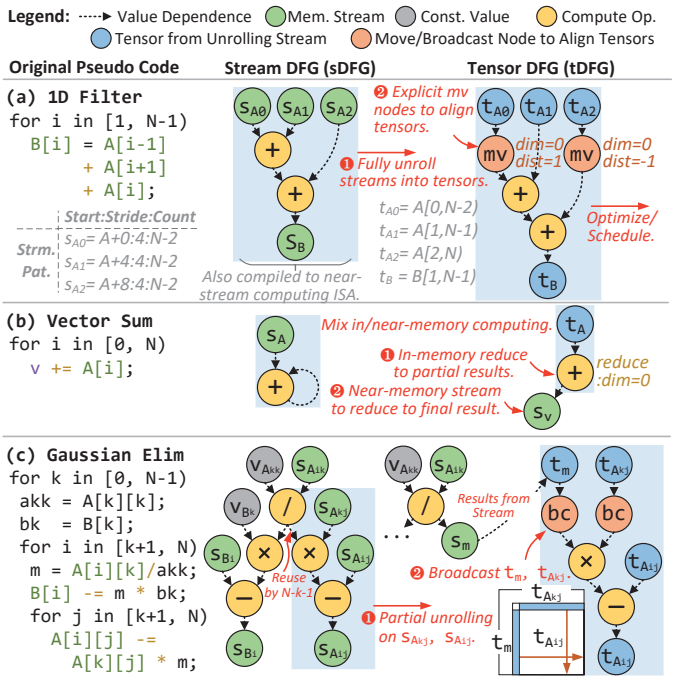

Fig. 1: Example of Compiling for Infinity Stream

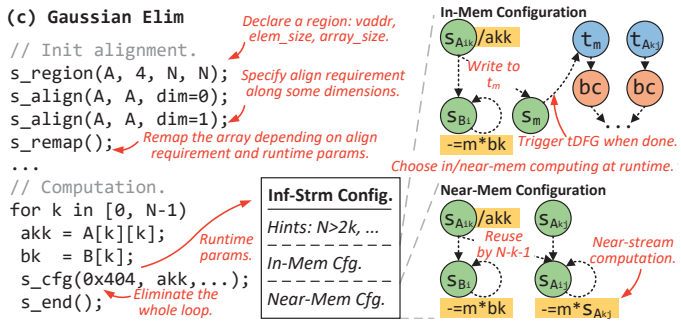

Fig. 2: Example of Compiled Infinity Stream Program

1 Infinity Stream Design

In this section, we discuss the key compiler and microarchitecture extensions of infinity stream, as well as some related works.

1.1 Compiling Infinity Stream

The compiler progressively lowers a program region through two IRs. First is the stream dataflow graph (sDFG), which embeds sequential access patterns as *streams* with associated near-stream computations. The tensor dataflow graph (tDFG) is an extension which lowers parallelizable streams to tensors for in-memory computing. The IR has a direct ending in the ISA.

Stream DFG: Fig 1 shows example programs and the extracted sDFG. In our system, each stream (and some associated instructions) can be offloaded to a core near where the data resides in L3 cache. Offload decisions depend on data locality and size (see 1.2).

Tensorization: In-memory computing essentially requires unrolling computation across all bitlines. Inspired by this observation, if the stream accesses a rectangular portion of the data structure, we can fully unroll it into a *tensor*. Unlike streams, tensors do not imply a temporal sequential semantics (i.e. elements processed one by one), but are fully expanded in space (specifically, bitlines in this work). Streams with complex patterns or loop-carried dependencies can not be unrolled into tensors (e.g. non-associative reduction).

Tensor Dataflow Graph: After unrolling streams spatially into tensors, the original computations and dependencies now apply to the entire tensor, forming a new tensor DFG (tDFG, Fig 1). Tensor operations, e.g. addition, requires operands to be exactly aligned in space, i.e. on the same bitline. Therefore, the tDFG explicitly represents shift operations to align tensors as `mv` nodes. For example, in Fig 1(a), tensor `A[0,N-2)` is moved to the right by 1 to align with `A[1,N-1)`. This is crucial to optimize and compile the data movement for in-memory computing. Also, tDFG is parameterized to maintain portability across variant input sizes.

Mixing sDFG and tDFG: Some streams/ops are not unrolled into tensors, e.g. outer-loop ops with insufficient parallelism to make in-memory computing profitable, leading to a mixed sDFG and tDFG. For example, in-memory computing can not handle the last few rounds of reduction efficiently, and the reduction in Fig 1(b) is split into two nodes: a tDFG node to perform partial in-memory reduction, and a stream node to perform final reduction on these partial results. Fig 1(c) shows an example where the sDFG communicates to the tDFG – stream s_{Bi} is not unrolled into a tensor due to low parallelism and stream s_m writes the division result (m) into a tensor t_m, which is later consumed by the tDFG.

Scheduling tDFG: After optimizing the tDFG (see below), the compiler performs register allocation and scheduling. An SRAM array with 256 wordlines only provides 8 32-bit registers so the hardware may recompute some results to reduce register pressure. Fusing multiple physical SRAM arrays into a larger virtual SRAM array with more registers is possible, but left for future work.

Infinity Stream Configuration: Fig 2 shows the compiled program of Fig 1(c). Both in/near-memory configurations are encoded in the binary to be selected at runtime. `s_cfg/end` instructions are inserted before and after the loop to set up and release the stream states. The original loop is eliminated when the entire loop body is decoupled as infinity stream region without aliasing. It also provides some hints to help the microarchitecture decide which one to execute at runtime, e.g., how large the data structure should be to make in-memory computing profitable.

Data Layout Hint: A suitable data layout (e.g. tiling sizes) is critical to reduce data movement. Since it depends on hardware details, e.g. network topology/bandwidth, we leave the decision to runtime and make tDFG neutral to the actual data layout. Nevertheless, the compiler can provide hints by analyzing the tDFG, e.g. it knows that the array would be shifted along certain dimensions (and should be tiled at those dimensions), as well as which arrays are used for computation (and should be aligned by bitlines). Fig 2 demonstrates using `s_region/align` instructions to declare a 2D array `A[N][N]`. The hint says it should be tiled to reduce shift traffic in both dimensions. Finally, a `s_remap` instruction triggers the hardware to select a data layout (see §1.2).

Optimizing tDFG: tDFG can be optimized to reduce data movement and reuse results, with two optimizations widely applicable:

- **Tensor Expansion**: It is common that two `mv` have the same distance and dimension but on slightly different tensors. In Fig 3, t_{A0}`:[0,N-2)x[0,M-2)` and t_{A3}`:[0,N-2)x[1,M-1)` are both shift to the right by 1. We can use one `mv` on the expanded tensor `[0,N-2)x[0,M-1)` instead of two shift operations.
- **Reuse Common Comp.**: We can also reuse earlier tensor computations. In Fig 3, instead of multiplying by `C0` four times, we can reuse the result by shifting it around as a tensor.

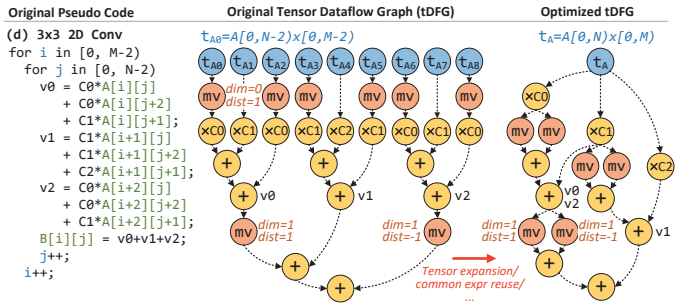

Fig. 3: Example of Optimized tDFG

Phys. Addr.	Wordlines	N_i	T_i	Transposed
0x100000-0x500000	[0, 32)	1024×1024	16×16	0

TABLE 1: Layout Override Table (LOT)

1.2 Microarchitecture Extensions

Fig 4 overviews microarchitecture extensions to support infinity stream for both in/near-memory computing. Broadly, *stream engines* execute streams, coordinate with the core for near-data execution, and make offloading decisions. *Tensor controllers* manage the data-layout through layout overide tables (LOT). *Tensor engines* executes in-memory computation on SRAM arrays.

In/Near-Memory Decision: When configuring infinity streams, the core stream engine (SE_{core} in Fig 4) evaluate the hints, e.g. the array size, and decides whether to offload as in/near-memory computing or not offload at all.

Near-Memory Computing: If near-memory computing is preferred, the SE_{core} offloads streams to the L3 stream engine (SE_{L3}) in Fig 4. The SE_{L3} issues offloaded near-stream computations to the local core within the same tile to reuse the vector units, effectively performing near-memory computing. This is similar to [8].

Transposed Data Layout: If offloaded in-memory, based on the compiler hints (via `s_region/align`), the tensor controller in Fig 4 chooses the transposed data layout. The data array is broken into multidimensional tiles, each transposed and mapped to one SRAM array (starting from the L3 bank 0, way 0, array 0).

Assume a 2D $N_0 \times N_1$ array with element size E, cache line size L and each SRAM array with B bitlines, the tile size $T_0 \times T_1$ must satisfy these constraints:

- $T_0 \times T_1 = B$: Each tile occupies all bitlines in one SRAM array.
- $T_0 \times E \bmod L = 0$: Each line is stored within one SRAM array.
- $N_i \bmod T_i = 0$: The array is fully tiled without padding.

With these constraints, transposed cache lines are guaranteed to be in a single SRAM array, and still accessible by normal core or stream requests (at the price of longer latency to transpose back). The tensor controller iterates some predefined tile sizes and pick the first valid one. If there is no valid tile size, the data structure will not be transposed and no in-memory computing is performed on it. In practice, this rarely fails for large arrays as they are often padded to cache line by programmers. Multiple data structures used by the same computation are aligned to the same bitline but occupying different wordlines, e.g. the input and output image of 2D convolution.

The layout override tables (LOTs) records the layout for each transposed data structure. Table 1 shows an example entry of LOT representing a 2D array with $N_0 = N_1 = 1024, E = 4$ tiled by 16 in both dimensions. We support transposed layout up to 3D array.

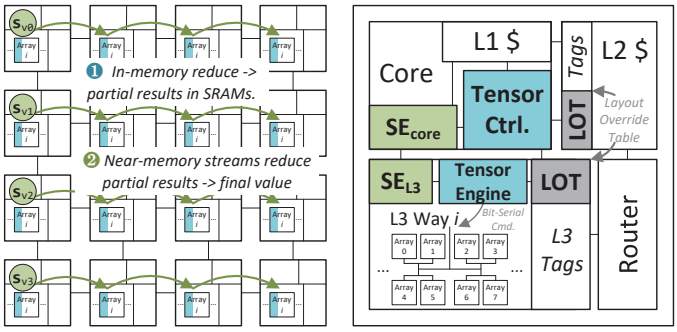

Fig. 4: Infinity Stream Microarchitecture

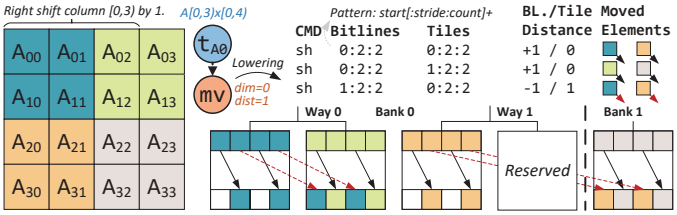

Fig. 5: Example of Tiled Data Layout and Lowering MV Node

Transpose Data: Each LOT entry contains a `transposed` bit initialized to 0, indicating whether the data within this physical address ranges are currently cached in transposed layout. Before starting the in-memory computing, the tensor controller checks this bit and creates special load streams to fetch and put the data into transposed format. Similar to normal offloaded streams, these load streams are executed in the SE_{L3} to avoid the traffic overheads between L2 and L3. After transposing, the tensor controller sets the `transposed` bit, and starts in-memory computing. We only need to transpose once for multiple computations on the same data.

Lowering tDFG: While waiting for data transposing, the tensor controller starts to lower the tDFG into bit-serial commands. This is left to runtime to maintain flexibility across variant input sizes and hardware platforms. Commands are reused if the region is executed multiple times with same parameters. Fig 5 shows a toy example how a 4x4 2D array is split into 2x2 tiles, and mapped to SRAM arrays. The `mv` node to right shift columns [0, 3) by 1 is lowered into three commands: two intra-SRAM-array shifts (with zero tile distance) to move column 0 and 2 respectively (black arrow); one inter-SRAM-array shift to move column 1 as it jumps across the tile boundary and reserved way (red arrow). Each command has a bitline and tile pattern to specify the applied bitlines and tiles. Compute and broadcast nodes are lowered similarly. The tensor controller broadcasts commands to tensor engines in L3 banks, which further breaks them into basic bit-serial operations, and skips those not applied to its local tiles.

Synchronization: To maintain correctness, the tensor controller inserts sync commands when coordination between tiles is needed. For example, a sync command is inserted between a shift and a compute command to ensure that all operands are ready. When encountering a sync command, the tensor engine reports the number of packets sent to each receiving bank to the tensor controller. After receiving acks from all tensor engines, the tensor controller checks all receiving tensor engines that the number of sent/received packets matches before broadcasting a message to clear the barrier.

System Size	8x8 Cores. Mesh NoC, 256-bit 1-cycle link.
L3 Cache	512x512 (32kB) SRAM array with 8-bit port. 8 arrays per way, 16 ways per bank, total 256MB.
DRAM	4x4 DDR4-3200MHz channels (25.6GB/s).

TABLE 2: L3 Cache Parameters

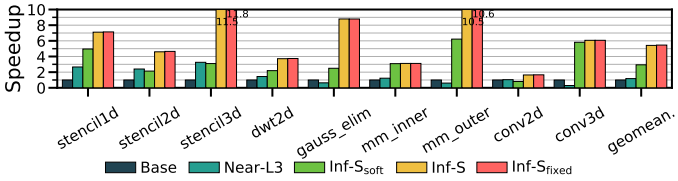

Fig. 6: Overall Speedup

1.3 Most Related Works

This work adopts the compiler and microarchitecture extensions from Near-Stream Computing [8] to support near-memory computing (e.g. sDFG, SE$_{core}$), and expands them to enable transparent and automated in-memory computing.

Duality Cache [3] is a CPU architecture with bit-serial logic logic in its L3 SRAM. Programmed with CUDA, it translates PTX emitted by NVIDIA's CUDA compiler (*nvcc*) into their in-memory ISA. While GPUs also have an abundance of threads, *nvcc* makes other architectural assumptions that are unavailable for in-memory compute. For example, without support for efficient shift operations common in many kernels, memory-traffic performance will suffer.

2 Evaluation

We evaluate infinity stream against conventional OOO cores and a state-of-the-art near-memory computing technique, and demonstrate that infinity stream successfully exploits the massive parallelism of in-memory computing while still maintaining the flexibility to incorporate near-memory computing.

Parameters and Configurations: Table 2 lists the key parameters of the L3 cache. In total, it has 4M bitlines and provides massive parallelism for in-memory computing. Arrays are connected with h-trees that support inter-array data movements. We assume the bit-serial operation latency from prior work [3]. The **Base** OOO cores uses advanced L1 and L2 prefetchers. For near-memory computing, **Near-L3** offloads streams and the associated computation to SE$_{L3}$. For infinity stream, we evaluate three configurations:

- **Inf-S$_{soft}$** invokes a runtime library to lower tDFG into bit-serial commands (about 1k cycles per command).
- **Inf-S$_{fixed}$** assumes that input and hardware parameters are known to precompile tDFG (no runtime lowering).
- **Inf-S** introduces a lightweight hardware pipeline to lower tDFG (10 cycles/cmd, mostly simple arithmetic operations).

Workloads: We evaluate 9 dense OpenMP workloads, 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. Unless stated otherwise, all workloads operate on a 2048×2048 matrix or equivalent sized input.

Performance: Fig 6 shows the overall speedup over **Base**. **Near-L3** achieves 18% speedup and 24% traffic reduction by offloading streams near L3 banks, but may hurt the performance as it is unable to capture the reuse, and it needs to send the data multiple times; e.g. for `conv3D` **Near-L3** introduces 110% extra NoC traffic. optimizing tDFG helps capture the reuse and avoid duplicated data movement and computation as in **Near-L3**. (see Fig 3).

By leveraging massive parallelism in bitlines, infinity stream achieves significant speedup (2.5× to 4.6×) over **Near-L3**. Also, **Cycle Breakdown:** Fig 7 breaks down the cycles of **Inf-S** into loading and transposing data from DRAM, lowering tDFG to commands, performing bit-serial computing and moving tensors around. Overall, in-memory computing takes 86% of total cycles, with 44%, 20%, and 20% spent in transposing, computing, and tensor moving respectively. 14% cycles are spent waiting for synchronization or near-memory streams, e.g. the reduction stream in `mm_inner`. Transposing is cheap when there is high reuse, e.g. `gauss_elim`. Dots in Fig 7 indicates the percentage of ops offloaded to bitlines – nearly all computation (99%) are performed in-memory to exploit the massive parallelism.

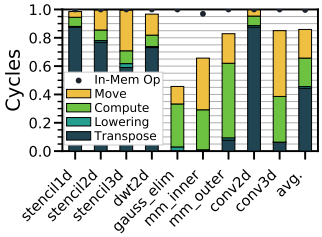

Fig. 7: Inf-S Cycle Breakdown

Overheads of Lowering tDFG: Although suffering from long latency to invoke a runtime library to lower the tDFG, **Inf-S$_{soft}$** still achieves 2.5× speedup over **Near-L3**. By fixing all parameters and precompiling tDFG into commands, **Inf-S$_{fixed}$** achieves the highest 4.6× speedup, but sacrifices portability. With the hardware lowering pipeline, **Inf-S** yields speedup close to **Inf-S$_{fixed}$** (4.5×) while maintaining the flexibility.

3 Conclusion

This work leverages tensors, a novel compiler/ISA abstraction, to enable flexible and transparent in-memory computing. By unrolling streams into tensors and explicitly embedding data movements in the tensor DFG (tDFG), the compiler can reason about exposed massive parallelism and optimize data layout and reuse. The tDFG is designed to be neutral to both data sizes and hardware details, and can be efficiently lowered to bit-serial commands at runtime in order to maintain flexibility. More importantly, conversion between streams/tensors fuses in/near-memory computing, which is crucial to continue performance scaling and energy efficiency improvements in future large-scale systems.

References

[1] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *HPCA '17*.
[2] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *ISCA '18*.
[3] D. Fujiki, S. Mahlke, and R. Das, "Duality cache for data parallel acceleration," in *ISCA '19*.
[4] J. Lowe-Power and et al., "The gem5 simulator: Version 20.0+," *arXiv:2007.03152*, 2020.
[5] P. Srivastava, M. Kang, S. K. Gonugondla, S. Lim, J. Choi, V. Adve, N. S. Kim, and N. Shanbhag, "Promise: An end-to-end design of a programmable mixed-signal accelerator for machine-learning algorithms," in *ISCA '18'*.
[6] Z. Wang and T. Nowatzki, "Stream-based memory access specialization for general purpose processors," in *ISCA '19*.
[7] Z. Wang, J. Weng, J. Lowe-Power, J. Gaur, and T. Nowatzki, "Stream floating: Enabling proactive and decentralized cache optimizations," in *HPCA '21*.
[8] Z. Wang, J. Weng, L. Sihao, and T. Nowatzki, "Near-stream computing: General and transparent near-cache acceleration," in *HPCA '22*.