Formal Method-based Data Placement for Software-Managed Heterogeneous Memory Systems

Final Report for Winter 2026 CS357S Course Project

Peijing Li
peli@stanford.edu

Katherine Mohr
kgmohr@stanford.edu

Abstract

This report presents FLAN, a formal method-based memory planning framework for software-managed heterogeneous memory systems. FLAN integrates with the MLIR compilation pipeline and formulates data placement as an Optimization Modulo Theories (OMT) problem solved by Z3 to synthesize placement schedules that are correct by construction with respect to capacity and retention constraints. Evaluations on linear algebra kernels demonstrate an average of 19.8 % decrease in energy and a 21.0 % decrease in latency compared to greedy heuristic baselines.

1 Introduction

Modern computing systems are increasingly limited by the "memory wall" [7]: the capacity and bandwidth of on-chip SRAM caches and off-chip DRAM are insufficient for data-intensive workloads such as AI/ML, especially on domain-specific accelerators. As SRAM and DRAM scaling faces physical limits [22], StRAM devices such as gain-cell embedded DRAM [8, 13]) and LtRAM devices such as PCM [6] or MRAM [21] offer denser alternatives with distinct tradeoffs: StRAM provides high density and lower access costs but requires refresh due to its limited retention; LtRAM offers non-volatility but has asymmetric access costs and limited endurance.

To exploit these new memory types effectively, we must rethink traditional memory hierarchies and data management policies. Li et al. [11] envision composing StRAM and LtRAM alongside SRAM in a software-managed scratchpad where the compiler, not hardware, controls data placement and decisions such as prefetch, eviction, migration, and refresh. As shown in Figure 1, this departs from the traditional memory hierarchy where the entire address space is transparently managed by hardware and organized into strict cache and main-memory levels. Prior work has introduced the term "cache bypass" [23] to describe this topology, where data present in one memory region (e.g., an SRAM cache) is not assumed to be a duplicate of data in another memory region (e.g., off-chip storage) when all memory regions become directly addressable by the processing elements.

We introduce FLAN, a formal method-based memory planning framework that formulates data placement in software-managed heterogeneous memory systems as an Optimization Modulo Theories (OMT) [18] problem solved by Z3 [5] that enables

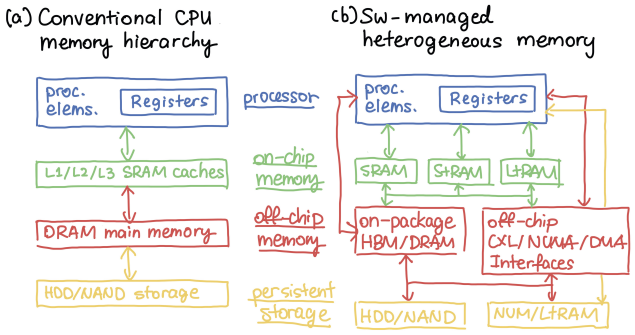

Figure 1. Comparison between the traditional hierarchical memory organization (a) and emerging heterogeneous memory systems (b).

an average of 19.8 % decrease in energy and a 21.0 % decrease in latency compared to heuristic baselines, and up to 82.9 % decrease in energy and 88.5 % decrease in latency for some configurations. FLAN's contributions are: (1) the first formalization of mapping into a heterogeneous memory system as a multi-commodity flow problem, (2) an OMT-based solver for this formulation, and (3) integration into the MLIR compilation infrastructure [9] via custom liveness analysis passes, enabling hardware-aware placement for any algorithm expressible in MLIR's `linalg` and `tensor` dialects.

2 Related Work

It is important to distinguish between two related but distinct problems, as articulated by Maas et al. [15]. The *data mapping* problem determines which level of a memory hierarchy to assign each data value to, while the *data allocation* problem chooses buffer locations within a shared, addressable memory. FLAN addresses the mapping problem and does not emit concrete memory addresses or offsets, which is the focus of native bufferization passes in MLIR.

A substantial body of prior work seeks to address data mapping as an optimization problem, often using mixed-integer linear programming (MILP) or hand-tuned heuristics for specific classes of workloads and architectures. RecShard [17] focuses on deep learning recommendation models, constructing a mixed-integer linear program for optimal embedding table sharding across memory regions. COSMA [12] combines operator scheduling, memory allocation, and tensor replacement in deep neural networks into a single ILP

formulation. CHOPT [24] formulates the data placement problem as a network flow and then uses a Minimum-Cost Maximum-Flow (MCMF) algorithm to make placement decisions. Like FLAN, it is able to handle arbitrary tiers of memory with asymmetric read and write costs; however, it cannot handle more complex memory constraints such as retention.

We hypothesize that many of the existing works relying on mixed integer linear programming (MILP) formulations for data placement are limited by the expressiveness of the linear constraint language, which necessitates complex encodings and auxiliary variables to capture non-linear relationships such as retention policies and energy-delay product objectives. In the meantime, no prior work has specifically targeted heterogeneous memory systems with retention-constrained StRAM regions, which requires additional modeling of refresh policies and overheads. Furthermore, prior work such as RANA [19] couple data placement to specific workloads (typically CNNs) and bespoke architectures, requiring redesign for each new algorithm family or hardware platform. FLAN aims to address these limitations by exploiting the expressiveness of OMT and integrating into the MLIR compilation pipeline for workload-agnostic placement.

3 Methodology Overview

Figure 2 illustrates the overall organization of the FLAN framework, consisting of three main components: (1) data flow extraction from MLIR, (2) memory system modeling, and (3) OMT-based solving, as well as a planned fourth stage that annotates MLIR `memref` types with memory-space attributes.

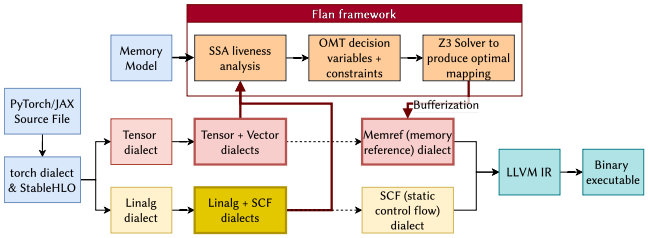

Figure 2. Overview of the FLAN framework: data flow extraction, memory modeling, and OMT-based solving.

4 Data Flow Extraction

The core representation for the data flow of a program is a set of `FlanValue` tuples, each representing a program static single assignment (SSA) value, as specified in Figure 3.

To compute these attributes, FLAN traverses the program graph, keeping track of each time an SSA value is created and read (§ 4.1), while using an analytical latency model to simulate the elapsed runtime (§ 4.2).

4.1 Capturing Data Liveness

Given some high-level Python code written in a StableHLO-compatible framework (i.e., Jax [4] or PyTorch [2]), FLAN first uses the MLIR pass infrastructure to lower this code to MLIR's "linalg-on-tensors" format [1].

`FlanValue` Abstraction Variables:

- $t_c^{(i)}$: Nanosecond when the value is defined.
- $t_k^{(i)}$: Nanosecond when the value is last used.
- $\text{size}(i)$: Size of the data value i in bytes.
- $\mathcal{T}_R^{(i)}$: A sorted list of times at which the value is read, used for determining refresh and migration needs.

Figure 3. The `FlanValue` abstraction. Each `FlanValue` represents an SSA value from the input program.

As shown in Figure 2, analyzing IR code at this level allows FLAN to guide MLIR's one-shot bufferization pass to provide hints on memory *mapping* before the concrete *allocation* decisions are made, and also allows for further optimizations such as affine transformations to be applied after the mapping decisions are made. Additionally, compared to a lower-level IR like LLVM IR, the linalg-on-tensors representation yields far more compact liveness data while still exposing analyzable low-level computations, which helps FLAN scale better to complex programs.

Once the source code has been lowered to linalg-on-tensors, FLAN walks the program control-flow graph, recording when each SSA value is created and read. For each operation, FLAN assigns a conservative wall-clock latency estimate (Section 4.2) and advances a simulated elapsed time accordingly. Bounded for-loops are handled with loop unrolling, and branches always conservatively assume the branch with higher latency will be taken. Handling of unbounded loops such as while-loops is left as future work.

4.2 From Logical Time to Elapsed Real Time

Classical liveness analysis captures which variables are alive at each *logical* point in the program. However, there is a fundamental mismatch between this logical view of time and the wall-clock time used to describe the retention of the provided memory models. To bridge this gap, we need a means of statically estimating the runtime of a list of MLIR operations.

Predicting the throughput of a set of instructions is a well-studied problem, with some prior approaches using machine learning techniques [16], and others using scheduling models combined with architecture-specific statistics about μops to simulate the runtime [1, 10, 14].

FLAN instead takes an analytical approach, using a roofline model [20] to estimate the latency of each `arith` or `linalg` operation. To estimate the runtime of each operation, FLAN computes the following conservative approximation:

[1]Here, "linalg-on-tensors" refers specifically to MLIR code consisting primarily of the `linalg`, `arith`, `scf`, and `tensor` dialects.

$$L = \max\left(\frac{F}{P}, \frac{B}{BW}\right) \tag{1}$$

- L: estimated latency
- F: floating-point operations
- P: average estimated compute throughput (FLOP/s)
- B: communicated bytes
- B: average estimated memory bandwidth (bytes/s)

Here, the number of floating-point operations F is computed as the iteration-space size multiplied by a hand-specified estimate of the floating-point operations per iteration, and the number of communicated bytes B is computed as the total sum of bytes across the operation's operands and result. The throughput P and bandwidth BW are to be specified by the user based on knowledge of the target hardware. While roofline models typically assume peak P and BW, Flan uses more conservative estimates, as underestimating latency may result in data being assigned to memories with insufficient retention time, leading to premature data loss.

The roofline model is particularly suitable for our use case, as it inherently leaves the memory bandwidth as a configurable variable, whereas other approaches often assume every memory operation will be a L1 cache hit. We assume different pieces of data can exhibit wildly different read times due to the heterogeneous nature of the memory architecture, so it is critical that the bandwidth be left as a configurable variable. Additionally, there is a circular dependency in this problem formulation: Flan needs to estimate program runtime in order to generate constraints to solve for the optimal memory mapping; meanwhile, mapping data onto memory inherently changes the runtime by impacting the latency of different data accesses. Leaving bandwidth as an explicit variable opens up potential future work to resolve this circular dependency, explained further in Sec 7.3.3.

5 Solver Design

The solver module (`FlanSolver`) takes a list of `FlanValue` instances and a `MemoryModel`, formulates the placement as a network-flow problem within the optimization modulo theorem (OMT) framework, and solves it using Z3's `Optimize` engine (νZ) [3, 5].

5.1 Input Memory Model ($\mathcal{M}$)

Flan is designed to model a heterogeneous memory system with an entirely software-managed scratchpad, so the various memory regions (e.g., DRAM, SRAM, NVM) are represented as distinct entities in our model.

Flan represents memory via two abstractions, `MemoryRegions` and `MemoryModels`, as detailed in Figure 4. A `MemoryRegion` encapsulates the physical properties of individual memory arrays. A `MemoryModel` aggregates multiple `MemoryRegions` and specifies pairwise *directed* migration costs for each ordered pair of regions, capturing the asymmetry in interconnects or other hardware-level constraints. Notably, different subpartitions of the same physical array can be modeled as separate regions, such as portions of an StRAM array with different retention times.

`MemoryRegion` Abstraction Variables

- cap(j): Total capacity of memory region j.
- ret(j): Retention time of memory region j.
- ER(j): Read energy cost of memory region j.
- EW(j): Write energy cost of memory region j.
- LR(j): Read latency cost of memory region j.
- LW(j): Write latency cost of memory region j.
- BW(j): Maximum bandwidth of memory region j.

`MemoryModel` Abstraction Variables

- M: A list of `MemoryRegions`.
- EM(s, d): Energy cost of migrating data from memory region S to memory region d.
- LM(s, d): Latency cost of migrating data from memory region S to memory region d.

Figure 4. The memory region and model abstractions.

5.2 Input Data Values ($\mathcal{V}$)

Each program data value is represented as a `FlanValue` instance, as detailed in Figure 3. We can additionally derive properties such as the lifetime of the value from these fields, which would be aligned with the retention values of memory regions to track potential refresh operation costs.

Note that we assume that the input data values are already pre-scheduled with respect to the creation and kill times of the SSA value and the read times of the data value. Future work might consider co-optimizing these schedules with the memory mapping decisions currently made by Flan.

5.3 Decision Variables and Symbolic Phases

Flan uses the following decision variables to model the placement of data values in memory regions.

$$d_k^{(i)} = \begin{cases} \tau_k^{(i)} \in [t_c^{(i)}, t_k^{(i)}] \\ \rho_k^{(i)} \in [0, |\mathcal{M}|) \end{cases} \qquad \text{where } i \in \mathcal{V}, k \in [0, K_{\max}) \tag{2}$$

Flan splits the liveness range of each data value i among all data values $\mathcal{V}$ into at most $K_{\max}$ phases, where each phase k is a contiguous time interval during which the value resides in a single memory region with the index $\rho_k^{(i)}$ within a total of $|\mathcal{M}|$ memory regions. Bounding the number of phases makes the problem more tractable, and this decision is based on the empirical assumption that most data values are not

expected to be migrated more than a few times during their lifetime.

Since $K_{\max}$ and $|\mathcal{M}|$ are constants, the total number of decision variables scales as $O(|\mathcal{V}| \cdot K_{\max})$, or linearly in the number of data values. This is a design decision to ensure that the problem does *not* scale with the number of execution cycles, making long-running programs with large lifetimes more tractable to solve.

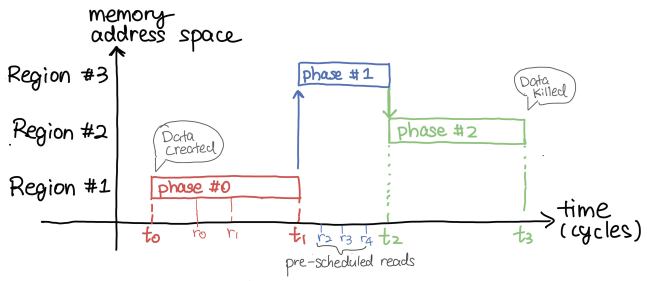

Figure 5. Symbolic transition-time encoding for a single value with three phases. Transitions between phases with different region assignments $\rho_k^{(i)} \to \rho_{k+1}^{(i)}$ incur migration costs.

5.3.1 Value Lifetime Range.

By this definition of phases, we can express the time when a data value is created as the beginning of the first phase $\tau_0^{(i)}$ and the time when a data value is killed as the end of the last phase $\tau_{K_{\max}}^{(i)}$.

$$t_c^{(i)} = \tau_0^{(i)} \quad \text{and} \quad t_k^{(i)} = \tau_{K_{\max}}^{(i)} \tag{3}$$

5.3.2 Looking Up Time Points in Phases.

One of the key challenges in the symbolic time encoding is to determine which symbolic phase k a given time point t belongs to. This can be done by iterating over all phases and checking if t is between the start and end times of the phase. We acknowledge that this is an expensive operation, and we implemented an iterative helper function to look up the phase index for a given time point.

$$T^{(i)}(t) = \begin{cases} k & \text{if } t \in [\tau_k^{(i)}, \tau_{k+1}^{(i)}] \\ \bot & \text{if } t \notin [\tau_0^{(i)}, \tau_{K_{\max}}^{(i)}] \end{cases} \tag{4}$$

Similarly, checking which memory region a data value i is in at a given time point t is nontrivial given the symbolic time encoding where the phase boundaries are not explicitly represented as decision variables.

$$\mathrm{Reg}^{(i)}(t) = \begin{cases} j & \text{if } t \in [\tau_k^{(i)}, \tau_{k+1}^{(i)}], \rho_k^{(i)} = j \\ \bot & \text{if } t \notin [\tau_0^{(i)}, \tau_{K_{\max}}^{(i)}] \end{cases} \tag{5}$$

5.4 Hard Constraints

In an OMT formulation, hard constraints are asserted on the Z3 `Optimize` instance and must be satisfied for any valid placement. All constraints are encoded as quantifier-free first-order logic formulae. The most important hard constraint is that *the placement of each data value must satisfy the capacity constraints of the memory regions*: at every key time point $t \in \mathcal{T}$ and memory region $j \in \mathcal{M}$, the total size of values that are alive at t and placed in j must not exceed the capacity of the memory region $\mathrm{cap}(j)$. We determine that there are two sets of key time points $\mathcal{T}$ that need to be evaluated for the capacity constraints: (1) the points when a data value is created or killed $t_c^{(i)}$ and $t_k^{(i)}$, and (2) the phase boundaries $\tau_k^{(i)}$ for each data value i and phase k.

$$\bigwedge_{t \in \mathcal{T}} \bigwedge_{j \in \mathcal{M}} \left(\sum_{i \in \mathcal{V}} \mathrm{If}\big(\mathrm{Reg}^{(i)}(t) = j,\ \mathrm{size}(i),\ 0\big) \le \mathrm{cap}(j) \right) \tag{6}$$

$$\mathcal{T} = \underbrace{\{t_c^{(i)}, t_k^{(i)} \mid i \in \mathcal{V}\}}_{\text{(1) creation/kill}} \cup \underbrace{\{\tau_k^{(i)} \mid i \in \mathcal{V},\ k \in \{0, \ldots, K_{\max}\}\}}_{\text{(2) phase boundaries}}$$

5.5 Cost Terms and Soft Constraints

All costs are accumulated into Z3 `IntVal` expressions and are organized into four categories, each contributing both an energy and a latency component:

- **Initial write cost:** Incurred when the data value is first written to memory in its 0th phase, a product of its size and the destination region's write cost.
- **Read cost:** Incurred at each explicit read time during the value's lifetime, a product of the size of the data value and the read cost of the memory region it resides in at that time point.
- **Migration cost:** Added whenever a data value migrates between two distinct memory regions across consecutive phases in addition to the read/write costs of the source and destination regions.
- **Refresh cost:** Applied when a data value is placed in a region with a non-zero retention time limitation. The number of refreshes is calculated based on the phase duration divided by the retention time, and each refresh is modeled as a read followed by a write within the same region.

5.6 Objective Function

The objective function for energy and latency is defined as the sum of the initial write cost, read cost, migration cost, and refresh cost.

$$E_{\text{total}} = \sum_i \left(E_{\text{init}}^{(i)} + E_{\text{read}}^{(i)} + E_{\text{mig}}^{(i)} + E_{\text{ref}}^{(i)} \right) \tag{7}$$

$$L_{\text{total}} = \sum_i \left(L_{\text{init}}^{(i)} + L_{\text{read}}^{(i)} + L_{\text{mig}}^{(i)} + L_{\text{ref}}^{(i)} \right) \tag{8}$$

FLAN's solver uses these objective functions to search for the placement schedule that minimizes one of the following:

(1) energy, (2) latency, (3) energy-latency product, (4) lexicographic (optimize energy first, then latency), or (5) Pareto (optimize energy and latency simultaneously).

6 Evaluation

We evaluated Flan on a set of simple Python JAX kernels and then compared the figures of merits (FoM) of total memory access latency and total dynamic energy of memory operations of the solver's output against heuristic baselines. The primary metric of interest is the percentage decrease in energy and latency compared to the best heuristic baseline, as this directly captures the improvement from using the solver to perform memory mapping decisions. We also varied the configurations of the memory model to understand the utility of Flan and when it becomes more beneficial to use.

6.1 Heuristic Baselines

To contextualize the solver's output, we compare against three greedy heuristic baselines inspired by Maas et al. [15]:

1. **Longest lifetime first**: greedily place the value with the longest lifetime into the lowest-cost region.
2. **Largest size first**: greedily place the value with the largest byte size.
3. **Largest area first**: greedily place the value with the largest product of size and lifetime.

The heuristics are implemented to place each value in the lowest-cost feasible region at the time of its creation, without any subsequent migrations. This means that the heuristics do not explicitly consider the potential benefits of data migration, and it may become oblivious to the different access patterns of subsequent data values after placing the first values with the highest empirical priority.

6.2 Workloads

We evaluate Flan on a set of nine Python JAX programs implementing basic linear algebra kernels, which are successively lowered into the MLIR `linalg` and `tensor` dialects to be consumed by Flan's liveness analysis passes. These are selected for two purposes: (1) to illustrate the utility of Flan on automatically processing workloads implemented in a high-level language (unlike prior work with hand-crafted execution graphs), and (2) to evaluate fundamental building blocks of more complex AI/ML workloads that will be the target of future work. The evaluation workloads as well as brief descriptions of their computational patterns are summarized below.

- `fused_ops`: Multi-phase program with fused kernel operations with early, persistent, and late buffers whose lifetimes intentionally overlap, separate, and rejoin.
- `jnp.dot`: Matrix dot product $C{=}A \cdot B$.
- `jnp.linalg.norm`: Frobenius norm of a matrix.
- `jsp.linalg.block_diag`: Assembles a block-diagonal matrix from two inputs.
- `jsp.linalg.tril`: Extracts the lower triangle part of a matrix.
- `jsp.linalg.triu`: Extracts the upper triangle part of a matrix.
- `jnp.matmul`: Single matrix multiply $C{=}AB$.
- `relu_loop`: Unrolled bounded loop of ReLU and matrix multiplication.
- `two_matmuls`: Chained matrix multiply operations.

6.3 Memory Model Configurations

We configured 34 different memory system layouts by varying the compositions, capacities, and performances of the memory regions in the `MemoryModel` input to Flan, with the following variations:

- On-chip SRAM: various capacities from 2 MB to 16 MB
- On-chip StRAM: various capacities from 8 MB to 64 MB. They have higher densities, lower access costs, but limited retention times from 64 µs to 1024 µs to simulate different gain cell RAM designs.
- Off-chip DRAM: various capacities from 128 MB to 512 MB, with high access costs and the option for remote access via CXL for more capacity and even higher access costs.
- Off-chip LtRAM: various capacities from 128 MB to 512 MB, with low read cost and high write costs to simulate various non-volatile memories.

6.4 Results

Applying Flan to the evaluation workloads and memory model configurations, we observed an average of 19.8 % decrease in the access energy and a 21.0 % decrease in the access latency compared to the best heuristic baseline; and up to 82.9 % decrease in energy and 88.5 % decrease in latency.

6.4.1 Impact of StRAM Retention Time and Capacity.

The ablation with the most significant impact on the FoM decrease was the configuration of the StRAM memory region, where varying both the capacity and retention time of the StRAM region led to significant differences in the FoM decrease, as shown in Figure 6.

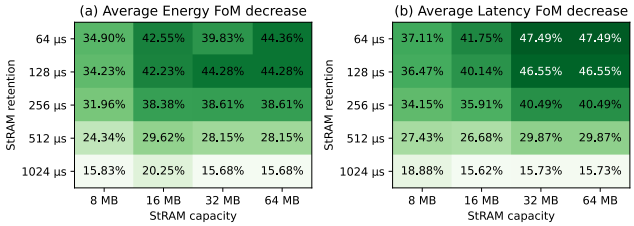

Figure 6. Average energy (a) and latency (b) FoM decrease for different StRAM retention times and capacities.

In general, a larger StRAM capacity and a smaller retention time led to more effective placements by FLAN, when the solver can discover more nuances in the refresh and migration policies to optimize for the energy and latency tradeoffs versus tight retention constraints. Conversely, when the StRAM retention time is large, the solver's optimal placement becomes more similar to the heuristic baselines that do not consider migration or refresh.

7 Discussion and Next Steps

7.1 Expressiveness of OMT

The OMT formulation provides qualitative advantages over MILP for memory placement. Retention implications and conditional refresh policies are expressed natively as quantifier-free first-order logic, avoiding Big-M reformulations. More importantly, complex data management policies (placement, migration, refresh, eviction) emerge as consequences of the optimization rather than hand-crafted heuristics requiring extensive expert knowledge.

7.2 Limitations

7.2.1 Joint scheduling. Scheduling and memory allocation are treated as separate problems. A joint formulation could yield better results but would significantly increase optimization complexity.

7.2.2 Address-level allocation. Data values are allocated to memory regions, not specific addresses. Contiguous allocation and fragmentation are not modeled; this will be addressed in future iterations via graph coloring constraints.

7.2.3 Wear leveling and endurance. The memory model currently does not capture the limited write endurance of LtRAM devices, which would require additional constraints to model wear-leveling policies and prevent excessive writes to the same region.

7.2.4 Control flow. FLAN's liveness analysis pass does not handle back-edges and data dependencies between different loop iterations without unrolling the loop. This also prevents FLAN from considering unbounded loop structures.

7.3 Next Steps

7.3.1 Improved Evaluations. FLAN is designed to compute optimal memory mappings for emerging hardware with heterogeneous memory architecture. These emerging hardwares are often built specifically to service machine learning workloads, and as such, FLAN should be evaluated over more complex machine learning models. As a next step, we plan to evaluate FLAN over more representative ML workloads. In the process, we will likely need to make changes to make FLAN more scalable.

7.3.2 End-to-End MLIR Integration. To properly integrate FLAN into the MLIR compilation flow, it should not only compute memory placement decisions, but it should also communicate these decisions to later compilation passes. To "close the loop", FLAN should include a final component which annotates bufferized `memrefs` with information on where to place them in memory.

7.3.3 Symbolic Liveness Encoding. As mentioned in Section 4.2, one advantage of the FLAN's current analytical latency model is that it inherently leaves the memory bandwidth *BW* as a configurable variable. As a next step, we could formulate the liveness constraints symbolically, leaving each operation's memory bandwidth as an additional variable to be solved for based on the memory placement decisions.

8 Conclusion

We presented FLAN, a formal method-based memory planning framework that formulates data placement in software-managed heterogeneous memory systems as an OMT problem solved by Z3. A symbolic transition-time encoding unifies placement, migration, refresh, and eviction into a single optimization, where data management policies emerge from the solver rather than hand-crafted heuristics. Integration with MLIR's `linalg` and `tensor` dialects allows FLAN to automatically extract liveness information from high-level programs and produce placement solutions that are correct by construction with respect to capacity and retention constraints. Across nine linear-algebra kernels and 34 memory configurations, FLAN achieves an average of 19.8 % and 21.0 % energy and latency reduction versus the best greedy baseline. We will continue to evaluate FLAN on more complex ML workloads, and our goal is to eventually fully integrate FLAN into the MLIR compilation flow for any arbitrary program and arbitrarily defined heterogeneous memory system.

References

[1] Andreas Abel and Jan Reineke. 2022. uiCA: accurate throughput prediction of basic blocks on recent intel microarchitectures. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 33, 14 pages. doi:10.1145/3524059.3532396

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA)

(ASPLOS '24). Association for Computing Machinery, New York, NY, USA, 929–947. doi:10.1145/3620665.3640366

[3] Nikolaj Bjørner, Phan Anh Dung, and Lars Fleckenstein. 2015. {*v*}Z - An Optimizing SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9035)*. Springer, 194–199. doi:10.1007/978-3-662-46681-0_14

[4] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/jax-ml/jax

[5] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[6] Scott W. Fong, Christopher M. Neumann, and H.-S. Philip Wong. 2017. Phase-Change Memory—Towards a Storage-Class Memory. *IEEE Transactions on Electron Devices* 64, 11 (Nov. 2017), 4374–4385. doi:10.1109/TED.2017.2746342

[7] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. 2024. AI and Memory Wall. *IEEE Micro* 44, 3 (May 2024), 33–39. doi:10.1109/MM.2024.3373763

[8] Odem Harel, Andac Yigit, Eliana Feifel, Robert Giterman, Andreas Burg, and Adam Teman. 2024. A 16-kB 65-nm GC-eDRAM Macro With Internal Bias Voltage Generation Providing Over 100 µs Retention Time. *IEEE Journal of Solid-State Circuits* (2024), 1–10. doi:10.1109/JSSC.2024.3489793

[9] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 2–14. doi:10.1109/CGO51591.2021.9370308

[10] J. Laukemann, J. Hammer, J. Hofmann, G. Hager, and G. Wellein. 2018. Automated Instruction Stream Throughput Prediction for Intel and AMD Microarchitectures. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 121–131. doi:10.1109/PMBS.2018.8641578

[11] Peijing Li, Muhammad Shahir Abdurrahman, Rachel Cleaveland, Sergey Legtchenko, Philip Levis, Ioan Stefanovici, Thierry Tambe, David Tennenhouse, Caroline Trippel, and H.-S. Philip Wong. 2025. Towards Memory Specialization: A Case for Long-Term and Short-Term RAM. In *Workshop on Disruptive Memory Systems (DIMES '25)*. Association for Computing Machinery, Seoul, Korea (South), 10. doi:10.1145/3764862.3768175

[12] Yi Li, Aarti Gupta, and Sharad Malik. 2023. *Combined Scheduling, Memory Allocation and Tensor Replacement for Minimizing Off-Chip Data Accesses of DNN Accelerators*. arXiv:2311.18246 doi:10.48550/arXiv.2311.18246

[13] Shuhan Liu, Jana Koustav, Liu Louis, Jing Wang, Leo Jen-Chieh Liu, Elia Ambrosi, Mingyuan Song, Christian Kubicka, Yiming Tan, Haitong Li, et al. 2025. Gain Cell Memory Scalability to 5-nm and Beyond. In *IEEE International Electron Devices Meeting (IEDM) (29, Vol. 5)*. IEEE, San Francisco, CA, USA.

[14] LLVM Project Developers. 2025. LLVM Machine Code Analyzer. https://llvm.org/docs/CommandGuide/llvm-mca.html

[15] Martin Maas, Ulysse Beaugnon, Arun Chauhan, and Berkin Ilbeyi. 2022. TelaMalloc: Efficient On-Chip Memory Allocation for Production Machine Learning Accelerators. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 123–137. doi:10.1145/3567955.3567961

[16] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. arXiv:1808.07412 [cs.DC] https://arxiv.org/abs/1808.07412

[17] Geet Sethi, Bilge Acun, Niket Agarwal, Christos Kozyrakis, Caroline Trippel, and Carole-Jean Wu. 2022. RecShard: statistical feature-based memory optimization for industry-scale neural recommendation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 344–358. doi:10.1145/3503222.3507777

[18] Nestan Tsiskaridze, Clark Barrett, and Cesare Tinelli. 2024. *Generalized Optimization Modulo Theories*. arXiv:2404.16122 doi:10.48550/arXiv.2404.16122

[19] Fengbin Tu, Weiwei Wu, Shouyi Yin, Leibo Liu, and Shaojun Wei. 2018. RANA: Towards Efficient Neural Acceleration with Refresh-Optimized Embedded DRAM. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 340–352. doi:10.1109/ISCA.2018.00037

[20] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. doi:10.1145/1498765.1498785

[21] H.-S. Philip Wong and Sayeef Salahuddin. 2015. Memory leads the way to better computing. *Nature Nanotechnology* 10, 3 (March 2015), 191–194. doi:10.1038/nnano.2015.29

[22] Geoffrey Yeap, S.S. Lin, H.L. Shang, H.C. Lin, Y.C. Peng, M. Wang, PW Wang, CP Lin, KF Yu, WY Lee, et al. 2024. 2nm Platform Technology Featuring Energy-Efficient Nanosheet Transistors and Interconnects Co-Optimized with 3DIC for AI, HPC and Mobile SoC Applications. In *2024 IEEE International Electron Devices Meeting (IEDM)*. 1–4. doi:10.1109/IEDM50854.2024.10873475

[23] Lei Zhang, Reza Karimi, Irfan Ahmad, and Ymir Vigfusson. 2020. Optimal Data Placement for Heterogeneous Cache, Memory, and Storage Systems. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 1 (May 2020), 06:1–06:27. doi:10.1145/3379472

[24] Lei Zhang, Reza Karimi, Irfan Ahmad, and Ymir Vigfusson. 2020. Optimal Data Placement for Heterogeneous Cache, Memory, and Storage Systems. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 1, Article 06 (May 2020), 27 pages. doi:10.1145/3379472