

Reducing Address Arithmetic Overheads: New Compiler Techniques for Programmable Dataflow-based AI Accelerators

Prasanth Chatarasi
IBM Research
Yorktown Heights, USA
prasanth@ibm.com

Chris Bowler
IBM
Toronto, Canada
cbowler@ca.ibm.com

Masoud Ataei Jaliseh
IBM
Toronto, Canada
Masoud.Ataei.Jaliseh@ibm.com

Viji Srinivasan
IBM Research
Yorktown Heights, USA
viji@us.ibm.com

Nicole Khoun
IBM
Toronto, Canada
nicolean@ca.ibm.com

Alex Gatea
IBM
Toronto, Canada
alexgatea@ibm.com

Alberto Mannari
IBM Research
Zurich, Switzerland
ERT@zurich.ibm.com

Swagath Venkataramani
IBM Research
Yorktown Heights, USA
swagath.venkataramani@ibm.com

Wei Wang*
IBM Research
Yorktown Heights, USA
wangweicomcn163@gmail.com

Shubham Jain
IBM Research
Yorktown Heights, USA
shubham.jain35@ibm.com

Bardia Mahjour
IBM
Toronto, Canada
bmahjour@ca.ibm.com

Abstract

Dataflow accelerators have emerged as a dominant architecture for modern AI workloads, combining massively parallel compute fabrics with hierarchical on-chip memories to achieve high performance and energy efficiency. Many such systems adopt a decoupled access-execute model, where programmable load/store engines orchestrate data movement between memory and compute units. While this design improves utilization, it imposes efficient address generation as a first-class performance constraint for the compiler. In these units, address arithmetic is typically realized as scalar operations interleaved with memory operations, forming dependence chains along the critical data supply path. Simple in-order pipelines exacerbate the impact of the latency of these computations and impact overall performance.

In this work, we propose a compiler approach to reduce address arithmetic overheads in programmable load/store units. Our approach consists of two complementary techniques: (1) *Address arithmetic absorption*, a local transformation that incorporates scalar address updates into composite memory operations by leveraging their pre-incr and post-update fields, and (2) *Address arithmetic hoisting*, a global transformation that restructures loop-carried address computations to simplify updates across iterations. We demonstrate the effectiveness of our approach in the context of IBM Spyre

accelerator and its compiler. Our results over a range of AI workloads spanning transformer, convolution-style models along with object-detection models demonstrate a geometric mean speedup improvement of 1.65 $\times$ in execution cycles over a strong baseline with state-of-the-art optimizations.

ACM Reference Format:

Prasanth Chatarasi, Nicole Khoun, Wei Wang, Chris Bowler, Alex Gatea, Shubham Jain, Masoud Ataei Jaliseh, Alberto Mannari, Bardia Mahjour, Viji Srinivasan, and Swagath Venkataramani. 2026. Reducing Address Arithmetic Overheads: New Compiler Techniques for Programmable Dataflow-based AI Accelerators. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '26)*, October 19–22, 2026, Chicago, IL, USA. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3838684.3846868>

1 Introduction

Artificial intelligence (AI) workloads continue to drive unprecedented demands on modern computing systems, necessitating specialized architectures to sustain performance and efficiency. Hardware acceleration has therefore become central to contemporary system design, with both industry and academia proposing a wide spectrum of accelerator architectures. Industrial designs such as IBM's Spyre [1, 5], NVIDIA's Deep Learning Accelerator (DLA) [29], Meta's MTIA [14], Google's TPU [23], AMD Versal ACAP [17], and SambaNova's systems [32] exemplify this trend.

Dataflow architectures have emerged as a dominant design paradigm for AI accelerators, enabling high compute utilization and energy-efficient execution. These architectures integrate massively parallel compute fabrics, such as SIMD units or systolic arrays, with hierarchical on-chip memory systems comprising multiple levels of scratchpad memories and vector register files [14, 29, 32, 34, 38]. To sustain the throughput demands of dense compute arrays, they employ wide, high-bandwidth datapaths that balance data supply

*The author was affiliated with IBM while this work was conducted.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

PACT '26, Chicago, IL, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2915-7/2026/10

<https://doi.org/10.1145/3838684.3846868>

with computational intensity. Many dataflow accelerators adopt a *decoupled access-execute* paradigm, wherein memory access and computation are handled by distinct programmable units. In such designs, specialized and simple in-order load/store engines are responsible for transferring data between on-chip scratchpad memories and compute units, in both directions. While this separation enables fine-grained control over data movement and improves compute utilization, it also shifts significant complexity to the compiler. In particular, the compiler must generate efficient load/store programs that orchestrate address generation, memory access, and data transfer to rate match the compute requirements.

A key challenge in programmable load/store engines arises from *address generation*. Unlike conventional processors, where out-of-order execution and speculation help hide address computation overheads, these units operate under constraints, including in-order execution and absence of branch prediction units. As a result, address arithmetic, typically expressed as scalar operations interleaved with memory operations, becomes a significant source of inefficiency. Since these engines operate in-order, each scalar addition must complete before the subsequent memory operations, serializing address computation with data transfer. Consequently, any delay in data transfer by the load/store engines directly stalls the compute units—in our evaluation, address arithmetic accounts for up to 40% of execution cycles of the load/store units (section 5)—leading to degraded performance.

Prior compiler efforts have addressed scalar computations through classical optimizations such as constant propagation, dead code elimination, CSE, and PRE [2, 3, 6, 7, 12, 18]. However, in load/store units, address arithmetic often manifests as linear chains of dependent scalar updates interleaved with memory operations, where each computation is live and contributes to subsequent address formation (an example in fig. 3(b)). These operations are neither redundant nor dead, and are therefore beyond the reach of classical scalar optimizations.

Many architectures, including general-purpose ISAs such as PowerPC [30] and ARM [4], as well as accelerators such as IBM Spyre [5], support composite memory operations with pre-increment and post-update semantics, such as auto-increment and base+offset addressing modes in load/store instructions. Peephole optimizations [2, 28] and instruction combining in compilers such as LLVM and GCC [16, 25] exploit these modes by fusing a scalar address update with an adjacent load/store into a single instruction. However, these techniques are inherently limited in two fundamental ways. First, they rely on syntactic matching over small windows of instructions, making them blind to address progression patterns that span multiple operations. Second, naively fusing scalar computations into pre-increment or post-update fields is often semantically incorrect under the constraints these fields impose (section 2.2, section 3), or inapplicable in the presence of multi-transfer operations such as bursts.

This paper makes the following contributions for reducing *address arithmetic overheads* in load/store units.

- We introduce *address arithmetic absorption*, a local, intra-loop transformation that collapses linear chains of composite memory operations interleaved with scalar additions into a

single leading scalar addition followed by composite operations by adjusting their pre-increment fields and preserving correct effective addresses (section 4.2).

- We develop *address arithmetic hoisting*, a global transformation that eliminates the residual per-iteration scalar address update remaining after absorption by simplifying across iterations and moving it out of the loop body (section 4.3).
- We present a unified compilation approach that applies absorption and hoisting in a bottom-up traversal of the loop hierarchy. Both algorithms are linear-time and add less than 1% to total compilation time in practice. We implement these techniques as an MLIR-based compiler pass deployed in production as part of the IBM Spyre compiler toolchain [37].
- Evaluation on diverse deep learning workloads, including transformers [13], CNN’s [19], and object detection models [27], demonstrates a geometric mean speedup of 1.65× over a strong baseline with state-of-the-art optimizations.

2 Background

In this section, we provide a concise overview of key concepts necessary for understanding the remainder of this paper.

2.1 Programmable Dataflow Accelerators

Dataflow architectures [10, 11, 15, 17, 20, 21, 24, 29, 31, 34], rooted in the decoupled access-execute paradigm [35], have emerged as a compelling foundation for modern AI accelerators. Unlike conventional CPUs and GPUs, which rely on cache hierarchies and a unified instruction stream to manage both memory access and computation, dataflow architectures explicitly orchestrate data movement through compiler-managed scratchpad memories. In these systems, computation and data movement are performed by multiple independent, specialized programmable units that communicate via distributed scratchpad memories and FIFO-based channels. Each unit executes its own instruction stream and is responsible for a distinct role in the dataflow pipeline, such as computation, data transfer, or specialized functional operations.

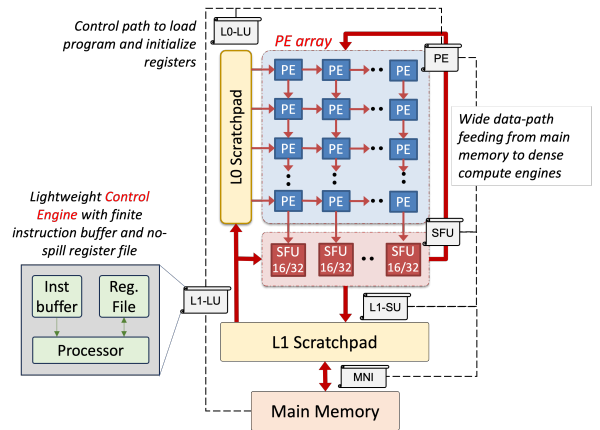


Figure 1: IBM Spyre single-core dataflow architecture (reproduced from [8, 9, 38]).

Figure 1 illustrates the baseline core architecture of the IBM Spyre accelerator [1, 5, 38], which follows a decoupled access–execute design. In this architecture, programmable compute engines, such as the PE (Processing Engine) and SFU (Special Function Unit), operate on data residing in a hierarchy of on-chip scratchpad memories, denoted as L0 and L1. Data movement across these memory levels and compute units is orchestrated by dedicated programmable load and store engines. Of particular relevance are the load engines that supply data to the compute pipeline. For instance, the L1–LU unit transfers data from the L1 scratchpad to L0, while the L0–LU unit streams data from L0 to the PE rows via FIFO channels. Complementarily, store engines (e.g., L1–SU) collect results from compute units and write them back into the memory hierarchy.

These load/store engines are lightweight programmable cores, each maintaining its own instruction stream and program counter. Beyond data movement, they are responsible for address generation, orchestration of memory accesses, and communication with other units via FIFO-based channels. To achieve high energy efficiency, these engines adopt simplified microarchitectures characterized by in-order execution, limited address arithmetic capabilities (e.g., only addition and subtraction), and the omission of features commonly found in general-purpose processors, such as branch prediction.

2.2 Composite Memory Operations

Composite memory operations are first-class primitives in programmable load/store units that unify address generation, memory access, and data movement within a single instruction. Unlike regular load/store instructions, which typically require separate scalar operations to compute addresses, composite memory operations embed address update semantics directly into the instruction, enabling more efficient execution. Conceptually, the address updates in these operations are analogous to indirect addressing modes in conventional ISAs such as x86 [22] and PowerPC [30], where base addresses can be combined with offsets (pre-increment) and/or updated automatically (post-update). In programmable dataflow accelerators, this idea is generalized further by tightly integrating memory access and address updates with data transfer across units. For example, in architectures such as IBM Spyre [5], composite memory operations simultaneously compute effective addresses, access data from scratchpad memories, stream the data to or from other programmable units, and then update the base address.

Composite memory operations differ from GPU bulk-copy instructions such as NVIDIA’s TMA, which are offloaded to a separate, asynchronous hardware engine — the issuing thread hands off a descriptor and continues other work while the engine executes the transfer independently. Composite memory operations, by contrast, execute directly on the issuing load/store unit itself, using explicit pre-incr/post-update fields as part of the instruction’s own addressing semantics.

Benefits. Composite memory operations avoid additional instruction fetch, decode, and issue overheads for pre-increment and post updates. At the micro-architectural level, address updates are performed as part of the instruction pipeline and can proceed in parallel with memory access and data transfer. This enables the base address to be updated without introducing additional pipeline dependencies or waiting for memory operations to complete. As a

result, these operations reduce serialization in the data supply path and enable more efficient streaming of memory accesses, which is critical for sustaining throughput of connected compute engines.

Representation. We model composite memory operations using a canonical five-operand form: `composite_op(base, pre_incr, post_update, unit, size)`, where the first operand (base) specifies the base address, the second operand (pre_incr) denotes the offset applied to the base address prior to the memory access, and the third operand (post_update) specifies the offset to add to the base address after the operation. The fourth operand (unit) identifies the destination (for loads) or source (for stores) programmable unit, and the fifth operand (size) denotes the number of bytes transferred. The instruction produces a new address as its result, which can be used in subsequent operations. An example of such composite operations can be seen in lines L05, L08, L11, and L14 in the motivating example shown at fig. 3(b).

Pre-increment and Post-update Constraints. Across ISAs, composite memory instructions restrict the post-update value to either equal the pre-increment or be zero. PowerPC’s update-form loads (`lwzu r3, 4(r4) [30]`) write back $EA = r4 + 4$ to the base register—tying the writeback to the pre-increment—while non-update forms (`lwz`) leave the base unchanged. ARM’s pre-index with writeback (`LDR Rd, [Rn, #imm]! [4]`) and plain offset form follow the same split. IBM Spyre also adopts a similar design. The restriction is not a hardware limitation but an enabling choice: because the writeback value is always either the effective address or the original base, thus address generation unit can select between these two already-computed values without requiring a separate adder or additional control logic—allowing the base update to be folded into the load/store with no pipeline delay. When a different post-update is needed, the programmer/compiler simply issues a scalar addition after the memory operation. The flexibility with the ISA support combines the update with the access for the common case, or express it separately when the stride differs, with no penalty either way. *Thus, the single-instruction form (common case) is therefore a performance optimization, not a fundamental limitation and maximizing its use is precisely the goal of our approach.*

2.3 Time-wrapped Composite Operations

Some composite memory operations encode multiple time-sequential memory transfers within a single instruction, and we refer to these as *time-wrapped* operations. The most common form of time-wrapping is *burst* mode, in which a single instruction performs B iterations of address generation, memory access, and data transfer, repeatedly advancing the address using the pre-incr and post-update fields. The post-update field is essential to this semantics: at each of the B steps it advances the result address by post-update value, so that successive transfers within the burst touch distinct locations; without it, every transfer in the burst would redundantly access the same address.

Lines 3–25 of fig. 3(b) can equivalently be expressed as a single time-wrapped composite operation. Whereas the original code issues four separate composite load operations along with address scalar arithmetic operations, the time-wrapped version requires only two composite load operations within the j loop: since `%u1` specifies a burst of 3 ($B = 3$), it expands to $1 + 3 = 4$ loads at runtime,

```

1 %a_next = for %j = 0 to 32 iter_args (%a_addr = %a) {
2   %u0 = composite_load_and_send(%a_addr, 0, 0, PE, 4)
3   %u1 = composite_load_and_send(%u0, 8, 8, PE, 4, ←
      burst = 3) // default burst is 1
4   %u2 = scalar_add %u1, 40
5   yield %u2
6 }

```

Figure 2: Time-wrapped composite operation equivalent to lines 3–25 of fig. 3(b).

exactly matching the four loads of the original. At each of the B iterations, the address is advanced by the post-update value, exactly as the explicit scalar arithmetic did in the original code; the two forms are therefore semantically equivalent. The burst argument thus encodes the time dimension of the composite operation along with this address manipulation, compactly capturing what would otherwise require unrolling the burst explicitly.

2.4 Loop Carried Variables Representation

Address arithmetic is often expressed through loop-carried variables, particularly after lowering and applying optimizations such as scalar replacement [36] and strength reduction [2]. We represent these variables similar to using MLIR’s structured control-flow constructs (e.g., Affine and SCF dialects [26]), where loop iterator arguments (`iter_args`) are initialized at loop entry, updated via `yield` operations at the end of each iteration, and returned as results of the `for` operation. In our motivating example (fig. 3(b)), the j -loop carries `%a_addr`, initialized with `%a` at line L03 and updated through `yield %u7`, while the k -loop similarly propagates `%b_addr` from its initial value `%b` at line L19. Conceptually, this mechanism is analogous to ϕ -nodes in LLVM IR, but is embedded directly within structured loop constructs.

3 Motivating Example

As a motivating and running example throughout the paper, we consider an illustrative program executed on the load unit (L0–LU) shown in fig. 3(a). This unit is responsible for fetching data from the L0 scratchpad and supplying it to the (PE), which performs core computations such as convolutions and batch matrix multiplications. The example captures a representative execution pattern, where address generation, memory accesses, and data movement are orchestrated within load/store programs alongside structured control flow. The program operates on a 4-D tensor A and a 3-D tensor B with int8 precision, located at base addresses X and Y , respectively (lines 7–8). The outer loop over i (line 9) defines the overall access pattern for both tensors. Within this loop, the j -loop (lines 14–21) traverses tensor A along its second dimension with a stride of two, while performing a group of four transfers along the third dimension (lines 17–20). Each transfer loads four contiguous bytes from the innermost dimension and forwards them to the PE using the `load_and_send_4` operation. In contrast, the k -loop (lines 23–26) performs a linear traversal over tensor B , where each transfer loads eight contiguous bytes from the innermost dimension using the `load_and_send_8` operation.

The corresponding lowered and optimized IR is shown in fig. 3(b). High-level affine accesses from the original program (lines 17–20 and 25 in fig. 3(a)) are translated into explicit address computations

using loop-carried variables (lines 2–3, and 19 in fig. 3(b)). Affine multiplications (e.g., $2*j$ at line 17) are eliminated via strength reduction [2], scalar evolution [33], and realized as loop-carried accumulations, which then further get optimized by loop-invariant code motion (LICM). These transformations are essential in accelerator settings where scalar multiplications and divisions are either expensive (for performance) or unsupported (for functionality).

Successive accesses (lines 17–20 in fig. 3(a)) are lowered into a sequence of composite memory operations interleaved with scalar increments (lines 5–14 in fig. 3(b)), where each address is derived from its predecessor using `scalar_add`. These increments are introduced through re-materialization strategies, such as live range reduction [9], which shorten the lifetime of intermediate values by recomputing them at their points of use (e.g., `scalar_add` at lines 6, 9, 12, and 16) with near-by definitions to mitigate register pressure. Consequently, address progression is transformed as a chain of scalar updates interleaved with composite memory operations, rather than relying on long-lived temporaries. As an example, the live range reduction technique realizes the address $A[i][2*(j+1)][0][0]$ from $A[i][2*j][3][0]$ with an addition of 40 (line 16), and this is correct because j strides by 2, one skipped group of four 8-byte sub-rows contributes $4 \times 8 = 32$ bytes, plus 8 bytes past the last loaded element, giving $32 + 8 = 40$. Even without re-materialization, such scalar increments are inherently present in the lowered IR, and they would appear as address computations directly derived from loop induction variables.

At this stage, composite memory operations (lines 5, 8, 11, and 14) are emitted with neutral addressing semantics, with both pre-increment and post-update fields set to zero. Focusing on the j -loop, each composite memory operation is followed by a scalar arithmetic instruction, resulting in suboptimal performance. Since the load unit can issue data transfers to the PE only after its base address is ready, these arithmetic instructions lie directly on the critical path of the data supply pipeline from the L0 scratchpad to the PE. This introduces additional latency and stall the compute units awaiting input data. Assuming a uniform cost model where each IR operation consumes one cycle, the explicit address arithmetic accounts for nearly 50% of the total execution cycles of the load program. This behavior exposes a key performance inefficiency in decoupled access-execute accelerators: *A large number of scalar address computations surrounding composite memory operations can significantly limit the utilization of the compute units they serve.*

A natural question is whether existing compiler techniques can address this inefficiency. Many architectures expose pre-increment and post-update fields in their memory instructions precisely to absorb scalar address updates, and compilers such as LLVM and GCC exploit these modes through peephole optimizations and instruction combining. We examine three such approaches below and show that each is either semantically incorrect, constrained by the field restrictions, or limited in scope — motivating the need for a more principled solution.

Possible approach 1 (Fusing into pre-incr). A naive approach is to eliminate scalar arithmetic by directly fusing it into the *pre-increment* field (second operand) of composite memory operations. For instance, the computation `%u1 = %u0 + 8` at line L06 could be absorbed into the pre-increment operand of the subsequent `composite_load_and_send` at line L08, rewriting the operation to

```

1 // (a) Running example: input L0LU program (psuedo C)
2 // 4-D Tensor A starts at L0 address X, and
3 // 3-D Tensor B starts at L0 address Y
4 // Layout of A (outer to inner) : [100][64][4][8]
5 // Layout of B (outer to inner) : [100][256][8]
6
7 l0addr_t baseA = X;
8 l0addr_t baseB = Y;
9 for (int i = 0; i < 100; ++i) {
10
11     // Traverse A along its second dim in strides of 2,
12     // while issuing a burst of four transfers
13     // corresponding to the third dimension in the body.
14     for (int j = 0; j < 32; ++j) {
15
16         // Each transfer loads 4 elements and send to PE
17         load_and_send_4(&A[i][2*j][0][0:3], PE);
18         load_and_send_4(&A[i][2*j][1][0:3], PE);
19         load_and_send_4(&A[i][2*j][2][0:3], PE);
20         load_and_send_4(&A[i][2*j][3][0:3], PE);
21     }
22
23     // Each transfer loads 8 contiguous elements from B.
24     for (int k = 0; k < 256; ++k) {
25         load_and_send_8(&B[i][k][0:7], PE);
26     }
27 }

```

```

1 // (b) Corresponding lowered (psuedo IR) from (a)
2 %res:2 = for %i = 0 to 100 iter_args(%a=%X, %b=%Y){
3     %a_next = for %j = 0 to 32 iter_args(%a_addr = %a) {
4         // A[i][2*j][0][0:3]
5         %u0 = composite_load_and_send(%a_addr, 0, 0, PE, 4)
6         %u1 = scalar_add %u0, 8
7         // A[i][2*j][1][0:3]
8         %u2 = composite_load_and_send(%u1, 0, 0, PE, 4)
9         %u3 = scalar_add %u2, 8
10        // A[i][2*j][2][0:3]
11        %u4 = composite_load_and_send(%u3, 0, 0, PE, 4)
12        %u5 = scalar_add %u4, 8
13        // A[i][2*j][3][0:3]
14        %u6 = composite_load_and_send(%u5, 0, 0, PE, 4)
15        // advance to A[i][2*(j+1)][0][0]
16        %u7 = scalar_add %u6, 40
17        yield %u7
18    }
19    %b_next = for %k = 0 to 256 iter_args(%b_addr = %b) {
20        // B[i][k][0:7]
21        %v0 = composite_load_and_send(%b_addr, 0, 0, PE, 8)
22        // advance to B[i][k+1][0]
23        %v1 = scalar_add %v0, 8
24        yield %v1
25    }
26    yield %a_next, %b_next
27 }

```

Figure 3: (a) Motivating and running example showing a typical load-unit (L0LU) program with explicit scalar address arithmetic and composite memory operations along with imperfectly nested loop, and (b) its corresponding lowered and optimized IR which introduces composite memory operations and explicit scalar arithmetic operations.

use %u0 as the base address with an adjusted pre-increment (e.g., 8) instead of materializing %u1 explicitly. While this transformation preserves the effective address of the memory access at line L08, it corrupts the result value produced by the operation. Composite memory operations have dual semantics: the memory access uses the effective address $EA = base + pre_incr$, but the result value is computed as $base + post_update$ — not from the pre-incremented address. In the original program, the result %u2 equals %u1 + 0 (post-update is zero), correctly reflecting the address passed in as the base. After absorption, the base becomes %u0 instead of %u1, so the result becomes %u0 + 0 — an incorrect value that causes all subsequent address computations to observe the wrong address, even though the memory access itself was correct. This mismatch makes direct fusion into the pre-increment field semantically incorrect.

Possible approach 2 (Fusing into post-update). Another seemingly natural approach is to absorb scalar address updates into the *post-update* fields of preceding composite memory operations. For example, the same computation $\%u1 = \%u0 + 8$ at line L06 could be folded into the post-update operand of the previous memory operation at line L05. Conceptually, this would change the post-update value of the instruction from 0 to 8, thereby eliminating the explicit scalar addition. From a purely semantic standpoint, this transformation is valid because the updated base address returned by the operation would match the value produced by the original addition. However, such transformations are often constrained by the values taken by post-update field of composite memory operations, i.e., either 0 or the same value as the pre-increment field (more details in section 2.2). Hence, the value 8 cannot be absorbed into the post-update operand of the operation at line L05 because it is neither zero nor same as the pre-increment value. Therefore,

while absorbing scalar additions into post-update fields may appear attractive, it is infeasible in practice due to the constraints on post-update fields to form a single instruction.

Possible approach 3 (Fusing into both pre-incr and post-update). Extending on the above approaches, another potential solution is to eliminate scalar arithmetic by fusing it into both the *pre-increment* (second operand) and *post-update* (third operand) fields of the subsequent dependent composite memory operations. While this strategy may partially reduce scalar arithmetic instructions — for example, by absorbing the scalar_add operations at lines L06 into the composite memory operation at line L08 — it is limited in scope. In particular, this approach fails to eliminate address arithmetic at lines L16 and L23. These remaining scalar computations cannot be captured by local fusion since there are no subsequent composite memory operations. Moreover, this fusion is only valid when the pre-increment and post-update fields of the target composite operation already have the same value before the fusion. Furthermore, this approach is inapplicable to time-wrapped composite operations such as burst mode, where the pre-increment and post-update fields govern multiple time-sequential transfers—fusing an arbitrary scalar offset into these fields would corrupt the address sequence across all transfers within the burst. We include this approach (referred to as IC) for comparison in our evaluation.

Our approach. We introduce two complementary compiler optimizations to reduce scalar address arithmetic in programmable load/store units: *Address Arithmetic Absorption* (section 4.2) and *Address Arithmetic Hoisting* (section 4.3). Together, these transformations remove all explicit scalar address computations within the i, j- and k-loops, as shown in fig. 5.

4 Our Proposed Approach

Our overall approach performs a systematic transformation of the input IR (section 4.1) by traversing the loop hierarchy in the reverse breadth-first (RBF) order. This traversal enables a bottom-up optimization strategy in which transformations are applied to inner loops before propagating to enclosing loops. For each loop node encountered, the approach first applies address arithmetic absorption (section 4.2) within the loop body to subsume explicit address arithmetic into composite memory operations, followed by address arithmetic hoisting (section 4.3) at the loop level to further reduce address arithmetic operations.

4.1 Input IR Programs

Our approach operates on an SSA-based intermediate representation with structured control-flow semantics (i.e., single-entry, single-exit). The IR comprises composite memory operations with pre-increment and post-update semantics, explicit scalar arithmetic, and loop constructs. Programs are organized as a hierarchy of for loops, where loop-carried state is explicitly represented using `iter_args` and `yield` operations, following the structured control-flow abstractions of MLIR Affine/SCF dialects (sections 2.2 and 2.4).

While the input IR can include general scalar arithmetic (e.g., multiplication and division), we assume that standard pre-processing passes normalize such expressions into sequences of additions and subtractions via strength reduction and related optimizations. After strength reduction, affine memory accesses — which multiply a loop induction variable by a constant stride — are converted into loop-carried additions with a constant increment, so some of the scalar additions in the normalized IR have one constant operand.

Additionally, upstream re-materialization passes such as live-range reduction [9] — a prior technique applied earlier in the compiler pipeline — often restructure input programs’ address computations into chains of scalar updates interleaved with composite memory operations by shortening value lifetimes to reduce register pressure. Although not required for correctness, such transformations significantly increase the opportunities for our optimizations, as demonstrated in section 5.

4.2 Address Arithmetic Absorption (AAA)

Address arithmetic absorption is a local transformation applied within the body of a loop that eliminates explicit scalar address arithmetic with the following key insight: *a linear chain of composite memory operations interleaved with scalar additions represents a sequence of pointer updates that can be restructured: the chain can be collapsed into a single leading scalar addition followed by a sequence of composite operations whose pre-incr fields are adjusted to preserve the correct effective address at each step, with post-update fields set to zero.* Unlike peephole instruction combining, which fuses only a single adjacent scalar-memory pair, absorption reasons over the *entire address chain jointly*. This global view allows it to collapse arbitrarily long chains of interleaved scalar additions and composite operations.

Linear chain. As an example, the body of the `j`-loop in fig. 3(b) forms a linear chain of address updates, `%a_addr → %u0 → %u1 → %u2 → %u3 → %u4 → %u5 → %u6 → %u7 → yield`, where each `composite_load_and_send` propagates the base address forward,

and each `scalar_add` advances it by a fixed offset (8 at lines L06, L09, and L12, and 40 at line L16). Similarly, the `k`-loop forms a shorter chain, `%b_addr → %v0 → %v1 → yield`. In both cases, each intermediate value has exactly one use, forming a single-def/single-use spine that connects the loop-carried input to the yield. This structural property enables systematic rewriting of the chain for absorption. The algorithm for address arithmetic absorption over a given loop L is presented in algorithm 1.

Algorithm 1: Address Arithmetic Absorption over loop L

```

//  $\mathcal{B}$ : Set of (chain, total-offset) pairs
//  $\mathcal{V}$ : visited ops
1  $\mathcal{B} \leftarrow \emptyset$ ;  $\mathcal{V} \leftarrow \emptyset$ ;
2 foreach  $op \in \text{body}(L)$  in bottom-up order do
3   if  $op \in \mathcal{V}$  then continue;
4    $\mathcal{V} \leftarrow \mathcal{V} \cup \{op\}$ ;
   //  $\Delta$ : merging increment;  $\sigma$ : chain
5    $\Delta \leftarrow 0$ ;  $\sigma \leftarrow \langle \rangle$ ;  $c \leftarrow op$ ;
6   while  $c$  has exactly one use  $\wedge c \in L$  do
7      $\mathcal{V} \leftarrow \mathcal{V} \cup \{c\}$ ;
8     if  $c \equiv \text{scalar\_add}(x, k) \wedge k \text{ constant}$  then
9        $\Delta \leftarrow \Delta + k$ ;  $\sigma \leftarrow \langle c, \perp \rangle$ ;  $c \leftarrow \text{Def}(x)$ ;
10    else if  $c$  is a composite memory op with constant
        pre-incr  $\delta$  and post-update  $\rho$  then
11       $\delta' \leftarrow \delta - \rho - \Delta$ ;  $\Delta \leftarrow \rho + \Delta$ ;  $\sigma \leftarrow \langle c, \delta' \rangle$ ;
12       $c \leftarrow \text{def}(\text{addr}(c))$ ;
13    else break;
14  if  $|\sigma| > 1$  then  $\mathcal{B} \leftarrow \mathcal{B} \cup \{(\sigma, \Delta)\}$ ;
15 foreach  $(\sigma, \Delta) \in \mathcal{B}$  do
16   let  $v_{\text{in}}$  be the value feeding the last entry of  $\sigma$  (i.e., the
        chain root, closest to the loop-carried input);
17   insert  $v'_{\text{in}} \leftarrow \text{scalar\_add}(v_{\text{in}}, \Delta)$  before the chain root;
18   foreach  $\langle e, \delta' \rangle \in \sigma$  do
19     if  $e$  is scalar_add then
20       replace all uses of  $e$  with the value feeding  $e$  in
        the chain and erase  $e$ ;
21   else
22      $e.\text{pre-incr} \leftarrow \delta'$ ;  $e.\text{post-update} \leftarrow 0$ ;

```

Chain identification (Algorithm 1, lines 1–14). The algorithm initializes an empty set $\mathcal{B}$ of (chain, total-offset) pairs (line 1) and scans the body of L bottom-up (line 2). For each operation op encountered, it initializes a running merging increment $\Delta \leftarrow 0$ and an empty chain σ (line 5), then walks backward through the use-def chain via c . Δ tracks the *total address offset accumulated so far in the backward walk* — the sum of all scalar additions and post-updates already absorbed — which will be placed as a single leading `scalar_add` at the chain root. At each step, if c is a `scalar_add` by constant k (line 8), k is folded into Δ , c is recorded in σ for deletion, and the walk continues to the defining operation of x (line 9). If c is a composite op with pre-incr δ and post-update ρ (line 10), the adjusted pre-incr is set to $\delta' = \delta - \rho - \Delta$, and Δ is updated to $\rho + \Delta$ (since the post-update ρ also advances the address and must be accounted for). The operation is appended to σ with its adjusted

pre-incr δ' , and the walk continues to the defining operation of *addr* (line 12). The walk also terminates if *c* has more than one use, belongs to a different region (outside the loop), or is of an unrecognized form (line 13). A completed chain, together with its final Δ , is added to $\mathcal{B}$ if it contains more than one entry. Single-entry chains (a lone `scalar_add` feeding `yield`) are excluded from absorption and are instead handled by address arithmetic hoisting (section 4.3).

Application to the running example—chain identification.

Scanning the j-loop body of fig. 3(b) bottom-up, the algorithm first encounters `%u7 = scalar_add (%u6, 40)` as a candidate op. Tracing backward: Δ accumulates to 40 after processing `%u7`; at `%u6 = composite_load_and_send(%u5, 0, 0, ...)`, $\delta' = 0 - 0 - 40 = -40$ and Δ remains 40 (since $\rho = 0$, $\Delta \leftarrow \rho + \Delta = 40$). Continuing through `%u5 = scalar_add (%u4, 8)`, $\Delta = 48$; at `%u4`, $\delta' = -48$, $\Delta = 48$; through `%u3 = scalar_add (%u2, 8)`, $\Delta = 56$; at `%u2`, $\delta' = -56$, $\Delta = 56$; through `%u1 = scalar_add (%u0, 8)`, $\Delta = 64$; at `%u0`, $\delta' = -64$, $\Delta = 64$. The walk stops at `%a_addr` since it is a loop-carried argument and the defining operation is not within the loop body. Hence, the resulting chain σ has eight entries with final $\Delta = 64$ and adjusted pre-incr values $-64, -56, -48, -40$ for `%u0, %u2, %u4, %u6` respectively. Scanning the k-loop body yields a two-entry chain $\sigma = \langle \%v1, \%v0 \rangle$ with $\Delta = 8$ and $\delta' = -8$ for `%v0`.

Chain transformation (Algorithm 1, lines 15–22). For each chain $(\sigma, \Delta) \in \mathcal{B}$, a single `scalar_add(vin,  $\Delta$ )` is inserted before the chain root (line 17), encoding the total accumulated offset as a single leading address update. Each `scalar_add` in σ is deleted and its uses forwarded to the next operation in the chain (lines 19–20), eliminating the intermediate SSA values. Each composite memory operation receives its computed δ' as its new pre-incr and its post-update is zeroed (line 22), since *addr* now flows unchanged through the chain carrying only the leading offset.

Application to the running example—chain transformation. Applied to the j-loop, the four `scalar_add` operations `%u1, %u3, %u5, %u7` are eliminated and a single `scalar_add(%a_addr, 64)` is inserted at the head of the chain; the four composite ops `%u0, %u2, %u4, %u6` receive pre-incr values $-64, -56, -48, -40$ respectively with post-update set to 0. For the k-loop, `%v1` is eliminated,

a leading `scalar_add(%b_addr, 8)` is inserted, and `%v0` receives pre-incr -8 . The resulting IR is shown in fig. 4(a).

Correctness. We show that after transformation, the effective address of each composite op is preserved. Define the *running address* at a program point as the value that flows into the next op in the chain. In the original IR, the running address entering the chain root is v_{in} ; each `scalar_add` by *k* advances it by *k*, and each composite op with post-update ρ advances it by ρ . In the transformed IR, the running address entering the chain root is $v_{in} + \Delta_{final}$; all `scalar_add`s are removed and all post-updates are zero, so the running address remains $v_{in} + \Delta_{final}$ throughout. For each composite op, let *S* denote the total advance in the original IR from v_{in} to just before this op (sum of preceding `scalar_add` constants and post-updates). Then $EA_{orig} = v_{in} + S + \delta$, and $EA_{new} = (v_{in} + \Delta_{final}) + \delta'$. Setting $\delta' = \delta - \rho - \Delta_{before}$ and using the fact that the backward walk maintains $\Delta_{final} - \Delta_{before} = S + \rho$ (since Δ accumulates the offsets of ops processed after this one in the walk, while *S* accumulates those before it), we get $EA_{new} = v_{in} + \Delta_{final} + \delta - \rho - \Delta_{before} = v_{in} + S + \delta = EA_{orig}$, preserving correctness.

4.3 Address Arithmetic Hoisting (AAH)

After applying absorption to a loop body, each linear chain begins with a `scalar_add` operation. The results of these operations form *derived induction variables* (derived IVs), expressed as a `scalar_add` of the loop-carried address argument *arg_i* with a loop-invariant constant *k* (e.g., Line 04 in fig. 4(a)). This update advances the base pointer in each iteration and naively incorporating this offset into subsequent composite memory operations may violate constraints on the post-update field, which must either be zero or pre-incr.

Key idea. The per-iteration offset *k* can be subsumed into the address progression already performed by the composite memory operations — specifically, into the post-update field of the first composite op m_1 , which already advances the address each iteration. Concretely, we set both the pre-increment and post-update of m_1 to *k*, so m_1 takes over the role of the scalar add, and then distribute

```

1 // (a) IR after Address Arithmetic Absorption
2 // on j, k-loops
3 %res:2 = for %i = 0 to 100 iter_args(%a = %X, %b = %Y){
4   %a_next = for %j = 0 to 32 iter_args(%a_addr = %a) {
5     %u0 = scalar_add %a_addr, 64
6     %u1 = composite_load_and_send(%u0, -64, 0, PE, 4)
7     %u2 = composite_load_and_send(%u1, -56, 0, PE, 4)
8     %u3 = composite_load_and_send(%u2, -48, 0, PE, 4)
9     %u4 = composite_load_and_send(%u3, -40, 0, PE, 4)
10    yield %u4
11  }
12
13 %b_next = for %k = 0 to 256 iter_args(%b_addr = %b) {
14   %v0 = scalar_add %b_addr, 8
15   %v1 = composite_load_and_send(%v0, -8, 0, PE, 8)
16   yield %v1
17 }
18 yield %a_next, %b_next
19 }

```

```

1 // (b) After Address Arithmetic Hoisting on j,k-loops
2 %res:2 = for %i = 0 to 100 iter_args(%a = %X, %b = %Y){
3   %a_0 = scalar_add %a, -64
4   %a_1 = for %j = 0 to 32 iter_args(%a_addr = %a_0) {
5     %u1 = composite_load_and_send(%a_addr, 64, 64, PE, 4)
6     %u2 = composite_load_and_send(%u1, 8, 0, PE, 4)
7     %u3 = composite_load_and_send(%u2, 16, 0, PE, 4)
8     %u4 = composite_load_and_send(%u3, 24, 0, PE, 4)
9     yield %u4
10  }
11  %a_2 = scalar_add %a_1 + 64
12
13 %b_0 = scalar_add %b, -8
14 %b_1 = for %k = 0 to 256 iter_args(%b_addr = %b_0) {
15   %v1 = composite_load_and_send(%b_addr, 8, 8, PE, 8)
16   yield %v1
17 }
18 %b_2 = scalar_add %b_1 + 8
19 yield %a_2, %b_2
20 }

```

Figure 4: (a) IR after address arithmetic absorption applied to the running example in fig. 3 and (b) IR after address arithmetic hoisting (Linear Chain) applied to the j- and k-loops.

the prior pre-increment of m_1 across other composite memory operation in the chain. The algorithm for address arithmetic hoisting is presented in algorithm 2.

Algorithm 2: Address Arithmetic Hoisting

```

1 foreach iter_arg  $arg_i$  of input loop node  $L$  do
2    $\mathcal{D} \leftarrow \emptyset$ ;
3   foreach user  $u$  of  $arg_i$  do
4     if  $u \neq \text{scalar\_add}(arg_i, k)$  then  $\mathcal{D} \leftarrow \emptyset$ ; break;
5     else  $\mathcal{D} \leftarrow \mathcal{D} \cup \{u\}$ ;
6   foreach derived iv  $d \in \mathcal{D}$ ;  $d \equiv \text{scalar\_add}(arg_i, k)$  do
7     if  $\forall v \in \text{users}(d): v \equiv \text{scalar\_add}(d, k_2)$  or ( $v$  is
      composite op with  $\delta = \rho$ ) (Elimination) then
8       foreach  $v \in \text{users}(d): v$  is composite op do
9          $v.\delta \leftarrow v.\delta + k; v.\rho \leftarrow v.\rho + k$ ;
10      foreach  $v \in \text{users}(d): v \equiv \text{scalar\_add}(d, k_2)$ 
11        do  $v.c \leftarrow k_2 + k$ ;
12      replace uses of  $d$  with  $arg_i$ ; erase  $d$ ;
13      break;
14   else if  $d$  forms a linear chain exclusively of
      composite ops  $m_1 \rightarrow \dots \rightarrow m_p \rightarrow \text{yield}[arg_i]$ 
      with  $\rho = 0$  (Linear Chain) then
15     let  $\delta_1 \leftarrow m_1.\delta; m_1.\rho \leftarrow k; m_1.\delta \leftarrow k$ ;
16     foreach  $j > 1$  do  $m_j.\delta \leftarrow m_j.\delta - \delta_1$ ;
17     insert  $init' \leftarrow \text{scalar\_add}(init_i, \delta_1)$  before  $L$ ;
18      $init_i \leftarrow init'$ ; replace uses of  $d$  with  $arg_i$ ; erase  $d$ ;
19     insert  $\text{scalar\_add}(result_i, -\delta_1)$  after  $L$ ;
20     replace uses of  $result_i$  with it;
21     break;
22   else if  $e \equiv \text{scalar\_add}(\_, c)$  feeds  $\text{yield}[arg_i]$ 
      (Generic) then
23      $e.c \leftarrow e.c + k$ ;
24     insert  $init' \leftarrow \text{scalar\_add}(init_i, k)$  before  $L$ ;
25      $init_i \leftarrow init'$ ; replace uses of  $d$  with  $arg_i$ ; erase  $d$ ;
26     insert  $\text{scalar\_add}(result_i, -k)$  after  $L$ ; replace
      uses of  $result_i$  with it;
27     simplify  $e$  and update users of  $e, arg_i$ ;
28     break;

```

Candidate collection (Algorithm 2, lines 1–5). For each loop-carried argument arg_i , the algorithm collects derived induction variables into $\mathcal{D}$: every user of arg_i must be a `scalar_add` with a constant operand k (lines 3–5). If any user fails this test, collection is abandoned for this arg_i since no hoisting is possible.

Method 1—Direct Elimination (lines 7–11). If every user of derived IV d is either a `scalar_add( $d, k_2$ )` or a composite op with $\delta = \rho$, then k can be absorbed directly: both δ and ρ of each composite user are incremented by k (preserving the $\delta = \rho$ invariant), and each `scalar_add` user’s constant is updated to $k_2 + k$. The derived IV d is erased and replaced by arg_i . Applied to the j-loop in fig. 4(a), $\%u0$ is the derived IV with $k = 64$. Its sole user $\%u1$ is a composite op with $\delta = -64$ and $\rho = 0$, so $\delta \neq \rho$; hence next method is tried.

Method 2—Linear Chain (lines 12–18). If d has exactly one use, arg_i has exactly one use, and the uses of d form a chain exclusively of composite ops $m_1 \rightarrow \dots \rightarrow m_p \rightarrow \text{yield}[arg_i]$ with $\rho = 0$ (outcome of absorption technique in section 4.2), the derived IV is hoisted by encoding k into m_1 : set $m_1.\delta \leftarrow k$ and $m_1.\rho \leftarrow k$, making m_1 advance the base pointer by k each iteration and eliminating the scalar add. Since this shifts the effective starting address of the chain by δ_1 (the original pre-incr of m_1), three compensating adjustments are made: (1) insert `scalar_add( $init_i, \delta_1$ )` before L as the new initial value, (2) subtract δ_1 from the pre-incr of each remaining op $m_2, \dots, m_p$, and (3) insert `scalar_add( $result_i, -\delta_1$ )` after L so the enclosing loop sees an unchanged value. Then d is erased and its uses replaced by arg_i . Applied to the j-loop: $k = 64$, $\delta_1 = -64$; after hoisting $m_1.\delta = m_1.\rho = 64$; the pre-incrs of m_2, m_3, m_4 become $-56 + 64, -48 + 64, -40 + 64$, i.e., 8, 16, 24; `scalar_add( $\%, -64$ )` is inserted before the j-loop; and a compensating `scalar_add` of $-(-64) = 64$ is inserted after it. The resulting IR after applying the linear chain method on the j- and k-loops is shown in fig. 4(b).

Correctness. The transformation replaces the initial value with $init_i + \delta_1$, shifting the loop-carried base arg_i up by δ_1 each iteration. To preserve each effective address, δ_1 must be redistributed across the chain. For m_1 : $EA_1^{\text{new}} = (arg_i + \delta_1) + k = (d + \delta_1) = EA_1^{\text{orig}}$ (since $d = arg_i + k$). For m_j ($j > 1$): after m_1 ’s post-update $\rho_1 = k$, the result of m_1 is $(arg_i + \delta_1) + k = d + \delta_1$; since all subsequent ops have $\rho = 0$, this value flows unchanged as the base to every m_j . With the adjusted pre-incr $\delta'_j = \delta_j - \delta_1$, we get $EA_j^{\text{new}} = (d + \delta_1) + (\delta_j - \delta_1) = d + \delta_j = EA_j^{\text{orig}}$.

```

1 // (c) IR after Address Arithmetic Hoisting on i-loop
2 %a_0 = scalar_add %X, -64
3 %b_0 = scalar_add %Y, -8
4 %res:2 = for %i = 0 to 100 iter_args(%a=%a_0, %b=%b_0){
5   %a_1 = for %j = 0 to 32 iter_args(%a_addr = %a) {
6     %u1 = composite_load_and_send(%a_addr, 64, 64, PE, 4)
7     %u2 = composite_load_and_send(%u1, 8, 0, PE, 4)
8     %u3 = composite_load_and_send(%u2, 16, 0, PE, 4)
9     %u4 = composite_load_and_send(%u3, 24, 0, PE, 4)
10    yield %u4
11  }
12
13 %b_1 = for %k = 0 to 256 iter_args(%b_addr = %b) {
14   %v1 = composite_load_and_send(%b_addr, 8, 8, PE, 8)
15   yield %v1
16 }
17 yield %a_1, %b_1
18 }

```

Figure 5: IR after address arithmetic hoisting (Generic) applied to the i-loop. All scalar address arithmetic (total dynamic count: 38,400) has been eliminated from loop bodies.

Method 3—Generic (lines 19–26). If a `scalar_add( $\_, c$ )` feeds $\text{yield}[arg_i]$, hoisting proceeds by absorbing k into that yielded update: set $e.c \leftarrow c + k$, insert `scalar_add( $init_i, k$ )` before L to pre-adjust the initial value, insert `scalar_add( $result_i, -k$ )` after L to restore the value seen by the enclosing loop, and erase d . **Correctness:** the loop-carried value now starts k higher (via the init), so $d = arg_i + k$ is equivalent to arg_i itself — the derived IV is redundant and can be removed. The yielded update absorbs the extra k , keeping the per-iteration progression unchanged. Applied

to the i-loop in fig. 4(b): `%a_0` is the derived IV with $k = -64$; $e \equiv \text{scalar_add}(\%a_1, 64)$ feeds `yield[%a]`, so $e.c$ updates to $64 + (-64) = 0$ (the add collapses to identity), `scalar_add(%X, -64)` is inserted before the i-loop, and a compensating `scalar_add` of $+64$ after it (omitted since the result has no uses). The same applies to `%b_0` with $k = -8$. The final transformed program, shown in fig. 5, removes all explicit scalar address arithmetic inside the loops while exploiting composite instructions respecting constraints on the post-update. The savings amount to $100 \times 32 \times 4 + 100 \times 256 \times 1 = 38,400$ scalar ops (one per composite op per loop iteration), each costing one cycle. A detailed step-by-step IR trace of the full approach applied to the running example from fig. 3(b) is shown in the supplementary material.

4.4 Time Wrapped Composite Operations

Some composite memory operations encode *multiple* time-sequential transfers within a single IR operation — a *burst* of B transfers where the effective address advances by post-update ρ value at each time step (described in section 2.3). Bursts cannot appear as chain members in absorption because their pre-incr and post-update fields are shared across all B time steps: directly fusing a chain offset into them would uniformly shift every transfer’s address, corrupting the intra-burst address sequence.

We handle this by *logically* expanding a burst into B single-transfer chain members during chain construction, assigning the t -th expanded operation ($t = 0, \dots, B-1$) a pre-increment based on its existing pre-incr, post-update, burst instance, and finally the merging increment accumulated just before this burst in the backward chain walk. This gives each time step an independent offset that accounts for both its intra-burst position ($t \cdot \rho + \delta$) and the surrounding chain context (Δ_{before}), allowing the standard absorption machinery to assign correct per-step pre-increments without corrupting the intra-burst address sequence. At materialization, the expanded single-transfer operations are re-compressed back into burst form wherever their adjusted pre-increments form a uniform arithmetic progression — i.e., $\delta'_t = \delta'_0 + t \cdot \rho'$ for some constant ρ' . This re-compression keeps the program footprint small, avoiding the code size blowup that naive expansion would incur, while still benefiting from the chain-level address arithmetic elimination.

5 Evaluation

In this section, we evaluate the effectiveness of our approach on representative deep learning (DL) workloads. Our techniques are implemented as a backend optimization pass in the Spyre compiler toolchain [37], targeting the IBM Spyre accelerator [38], a dataflow architecture with programmable load/store units supporting composite memory operations.

Workloads. We evaluate inference workloads from three representative DL model families: ResNet-50 [19] (Image classification), SSD [27] (object detection), and BERT Large [13] (LLM encoders). All experiments use INT8 precision with a batch size of one.

Hardware. Our evaluation targets a single-core configuration of the IBM Spyre accelerator [5] (Figure 1). A key component is the L0-LU (Level-0 Load Unit), which transfers data from the L0 scratchpad to the processing element (PE) array and is actively

engaged in operations such as convolutions and batch matrix multiplications. Since the PE array depends on a continuous supply of operands, the efficiency of the L0-LU is critical for sustaining high PE utilization. The data transfer schedule and access patterns executed by this unit are fully determined by the compiler.

Baseline, Variants, and Metric. The Baseline is the production code path from IBM’s Spyre compiler [8, 9, 37], built on open-source MLIR, and already incorporates a mature set of optimizations including aggressive scalar simplification, loop-invariant code motion, strength reduction, common sub-expression elimination, dead-code elimination, and partial redundancy elimination, layered with proprietary architecture-specific transformations, representing the best code the compiler can currently produce for this hardware. All variants (IC, AAA, AAH, AAA+AAH) share these identical preceding passes; these techniques are applied as an additional pass on top of the baseline. We evaluate three variants of our approach: AAA (absorption only), AAH (hoisting only), and AAA+AAH (the full combined approach). For comparison, we also implement a peephole-style instruction combining (IC) optimization that locally fuses scalar address arithmetic into the pre-increment and post-update fields of adjacent composite memory operations, analogous to Possible Approach 3 in section 3. We report *speedup* of the L0-LU computed relative to baseline execution cycles as our primary metric.

5.1 Layer-Wise Comparison

Figure 6 presents per-layer-group speedups for all three workloads. Layers with similar behaviors are grouped on the x-axis (for space brevity); the geometric mean across all groups is shown as the rightmost bar. We describe the key observations below.

Observation 1: AAA+AAH eliminates nearly all scalar address arithmetic from L0-LU programs. Across all three workloads, the combined AAA+AAH approach removes more than 90% of scalar arithmetic operations from the generated L0-LU programs: 99.3% for ResNet-50, 100% for BERT Large, and 93.7% for SSD (seen in fig. 8). The 100% elimination in BERT Large reflects the homogeneous structure of transformer MatMul kernels, where all scalar address arithmetic is fully reflected into the pre-increment and post-update fields of composite memory operations (similar to our running example). The residual in SSD (6.3%) corresponds to address computations that fall outside the single-def/single-use chain structure required by absorption, such as addresses used by multiple consumers.

Observation 2: AAA and AAH are complementary; their combination is necessary for maximum benefit. Across all three workloads, AAA+AAH consistently and substantially outperforms either technique applied in isolation. AAA alone restructures intra-loop address chains—subsuming scalar arithmetic into composite memory operations—leaving exactly one residual scalar update at the head of each chain. AAH alone has limited opportunity without this prior normalization, since it requires the loop-carried update to appear as a clean derived induction variable feeding a linear chain of composite ops; consequently, AAH-only speedups remain close to $1.1\times$ across all workloads and groups. To quantify the extent to which AAA enables AAH, we measured the fraction of AAH’s transformation opportunities that are exposed only after AAA has run, as opposed to those already present in the input IR. Across

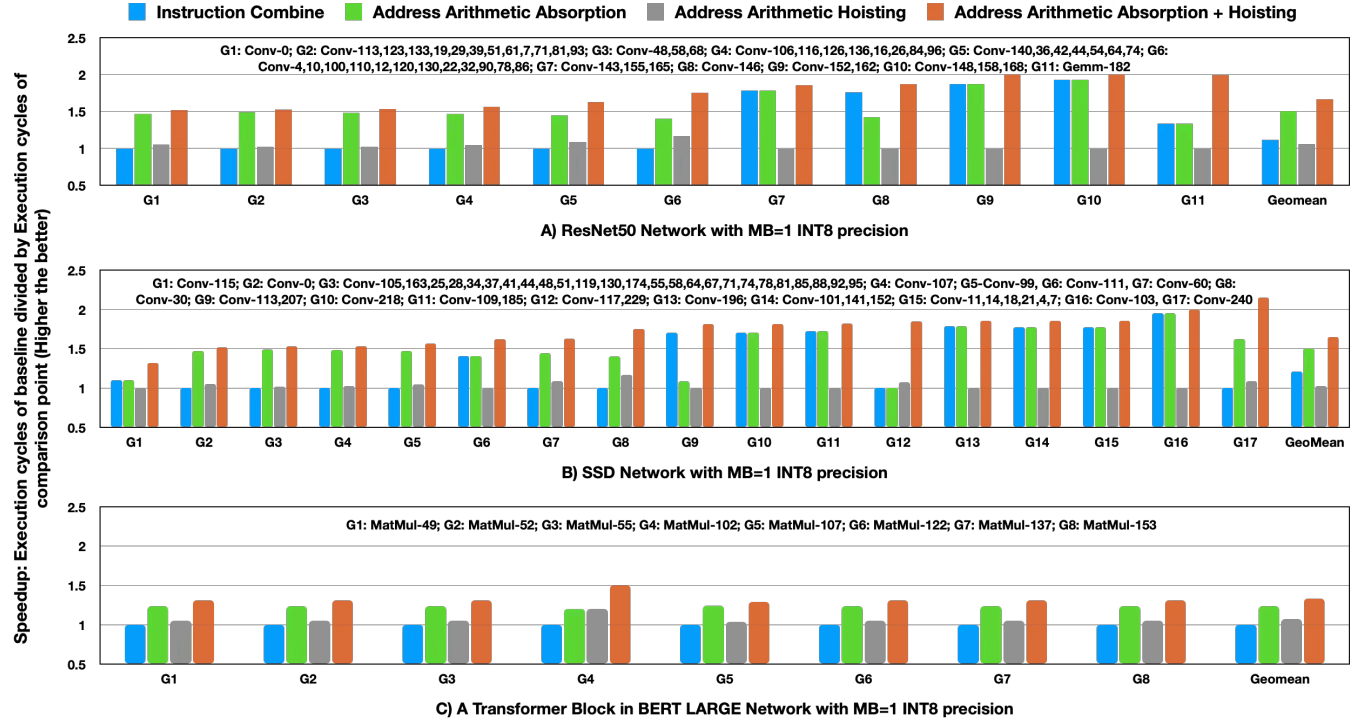


Figure 6: Layer-group speedup for ResNet-50, SSD, and a BERT Large transformer block across the baseline and four compiler configurations: Instruction Combine (IC), Address Arithmetic Absorption (AAA), Address Arithmetic Hoisting (AAH), and the combined AAA+AAH approach.

our evaluated workloads, more than 75% of AAH’s opportunities emerge exclusively after AAA has collapsed the chain of composite memory operations and adjusted their pre-increment fields — underscoring that AAA is not merely a standalone optimization, but a critical enabler that unlocks the majority of AAH’s impact. The combined approach achieves geometric mean speedups of $1.65\times$ (ResNet-50), $1.66\times$ (SSD), and $1.33\times$ (BERT Large) over the baseline, compared to lower gains from either variant alone.

Observation 3: AAA outperforms IC due to burst-awareness and relaxed fusion preconditions. In several groups, G1–G6 in ResNet-50 and G2–G5 in SSD, AAA-only substantially outperforms IC. These groups correspond to convolutional layers whose access patterns involve composite memory operations with burst semantics, where a single IR operation encodes multiple time-sequential memory transfers. IC is inapplicable in two ways in these groups: first, directly fusing a scalar offset into the pre-increment field of a burst operation would incorrectly apply the offset to every transfer within the burst, corrupting the address sequence; second, several non-burst composite ops in these groups have pre-incr and post-update values that already differ ($\delta \neq \rho$), violating the precondition that IC requires before fusing an adjacent scalar offset. AAA handles both cases: burst operations are handled via logical expansion (section 4.4), and non-burst ops with $\delta \neq \rho$ are handled by the chain-wide absorption that adjusts pre-incr fields globally rather than relying on a local $\delta = \rho$ precondition.

In a small number of groups (e.g., G8 in ResNet-50 and G9 in SSD), IC modestly outperforms AAA-only. These correspond to innermost loops with very short chains—typically a single scalar

add immediately preceding one composite op with same pre-incr and post-update values—where IC’s local fusion eliminates the scalar op directly. AAA, by design, leaves this residual update in place as the normalized chain head for AAH to hoist out of the loop. The full AAA+AAH approach recovers and exceeds IC’s gain in every such case, confirming that the residual is not wasted but deferred to exploiting together with AAH.

Observation 4: Speedup correlates with loop depth and iteration count. The magnitude of speedup from AAA+AAH varies across layer groups within each workload. For ResNet-50, speedups range from $1.5\times$ – $1.6\times$ in G1–G4 (early shallower-nested layers) up to $\approx 2\times$ in G7–G10 (deeper convolutional layers with larger kernels such as Conv-143,155,165). For SSD, the highest gains appear in G7–G17, with peaks approaching $2\times$ or above in groups covering the larger convolutions (e.g., Conv-117,229 in G12 and Conv-103 in G16), which feature deeper loop nests and higher per-iteration address arithmetic. For BERT Large, gains are more uniform across G1–G8 ($1.3\times$ – $1.5\times$, with G4 peaking at $\approx 1.5\times$), reflecting transformer attention and feed-forward MatMul blocks.

5.2 Impact of Live Range Reduction Technique

Figure 7 isolates the contribution of live range reduction (LRR) [9], a re-materialization technique that restructures address computations into explicit chains of scalar updates interleaved with composite memory operations that our techniques exploit. As discussed in section 4.1, LRR is not required for correctness of our optimizations, but it directly increases the density of absorption and hoisting opportunities by producing the single-def/single-use chains.

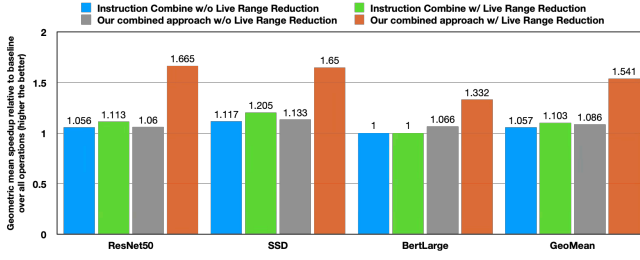


Figure 7: Geometric mean speedup over the baseline for IC and our combined AAA+AAH approach, each evaluated with and without live range reduction (LRR).

The results confirm this relationship sharply. Without LRR, our combined approach achieves modest geometric mean speedups of 1.06 $\times$ (ResNet-50), 1.13 $\times$ (SSD), and 1.06 $\times$ (BERT Large), improvements that are real but limited, since only the address chains already present after standard lowering are available for transformation. With LRR enabled, speedups increase dramatically to 1.66 $\times$, 1.65 $\times$, and 1.33 $\times$ respectively, a gain of 1.57 $\times$ –1.59 $\times$ relative to the without-LRR configuration. This demonstrates that LRR and our address arithmetic optimizations are strongly synergistic: LRR exposes the chain structure, and our techniques then eliminate it.

In contrast, IC benefits only minimally from LRR. Without LRR, IC achieves 1 $\times$ –1.11 $\times$ geometric mean speedup; with LRR, this improves to only 1 $\times$ –1.20 $\times$. The limited interaction reflects IC’s fundamental scope: it fuses only locally adjacent scalar-op/memory-op pairs, so restructuring value lifetimes to form longer chains provides no additional fusion targets. The ratio of AAA+AAH+LRR to IC+LRR geometric mean speedups ranges from 1.33 $\times$ to 1.49 $\times$ across workloads, quantifying the advantage of our global, chain-aware, loop-level approach over purely local instruction combining.

Summary. The results from section 5.1, section 5.2 collectively highlight a broader trend: typical AI inference workloads — dominated by convolutions, matrix multiplications, and attention mechanisms — operate on tensors with affine access patterns, where strength reduction, loop-invariant code motion, and live range reduction transform high-level multi-dimensional accesses into structured chains of scalar updates interleaved with composite memory operations. This regularity is precisely the structure that AAA and AAH are designed to exploit. The near-complete elimination of scalar address arithmetic across diverse model families confirms that encoding address progression directly into composite memory operations is not merely a micro-optimization but a critical compiler responsibility in decoupled access–execute accelerators, where every cycle of address arithmetic on the critical data supply path directly translates to compute unit stalls.

5.3 End-to-End Layer Speedups

Throughout section 5.1, we report speedup of the L0-LU as our primary metric because it is the major data-supply engine to the PE compute array: any delay in L0-LU directly stalls compute. Compute-bound operations such as convolutions and batch matrix multiplications are scheduled in two phases — a *block-loading* phase, in which weights are staged into the PE array, followed by a *streaming-computation* phase fed continuously by L0-LU. Since block-loading overhead is small relative to the streaming-computation

phase, L0-LU efficiency dominates the per-layer critical path in the common case, motivating its use as our primary evaluation metric.

To validate this assumption directly, we additionally measure *end-to-end* per-layer speedups, shown alongside the corresponding scalar-arithmetic reduction in fig. 8. Geometric-mean end-to-end speedups are 1.54 $\times$, 1.47 $\times$, and 1.30 $\times$ for ResNet-50, SSD, and BERT Large, respectively, compared to L0-LU-level speedups of 1.65 $\times$, 1.66 $\times$, and 1.33 $\times$ in fig. 6. End-to-end gains thus retain 88–98% of the corresponding L0-LU-level gains across all three workloads, confirming that L0-LU lies on, or very near, the critical path for the large majority of layers, rather than hidden by compute overlap.

For a finer-grained view of where L0-LU is — or is not — the bottleneck, fig. 8 reports per-group end-to-end speedups alongside the scalar-arithmetic reduction. The specific layers comprising each group are listed in fig. 6. The overwhelming majority of groups closely track their corresponding L0-LU speedups, with a small number of exceptions — most notably the final group of ResNet-50, and groups G1, G12, and G17 of SSD — falling closer to 1.0–1.2 $\times$. These groups correspond to layers with small tensors or low arithmetic intensity, where the streaming-computation phase is short enough that L0-LU is no longer the dominant contributor to the critical path, and other factors (e.g., block-loading overhead or compute latency) become comparatively more significant.

5.4 Compilation Overheads

Both AAA and AAH are designed with linear-time complexity. AAA performs a single bottom-up traversal of each loop body, visiting every operation at most once across all chain walks due to the visited-ops guard, yielding $O(n)$ per loop node. AAH performs one scan per *iter_arg* with a single-use chain walk of bounded length, giving $O(p \cdot n)$ per loop node where p is the number of *iter_args*—effectively $O(n)$ in practice since p is bounded by the number of distinct address streams, which is small and constant in the load/store programs we target. Applied bottom-up across the full loop hierarchy, the total pass complexity is $O(N)$ in the number of operations in the program unit.

We measure compilation overhead on a standard development machine (Apple M3 Max with 64GB Memory) using single-threaded compilation of the full ResNet-50, SSD, and BERT Large workloads. Across all three workloads, enabling our combined AAA+AAH pass adds less than 1% to total single-threaded compilation time relative to the baseline compiler. This negligible overhead is a direct consequence of the linear-time algorithms: the pass touches each operation a constant number of times and introduces no iterative fixed-point computation or exponential search. The optimization is therefore practical to enable unconditionally in production compiler pipelines without any compile-time budget concerns.

5.5 Applicability to Other Architectures

Our approach relies on the presence of composite memory operations with pre-increment and post-update fields, where the post-update is constrained to be either zero or equal to the pre-increment. As discussed in section 2, this pattern is not unique to IBM Spyre — it is a common property of update-form addressing found across general-purpose ISAs, including PowerPC (lwzu,

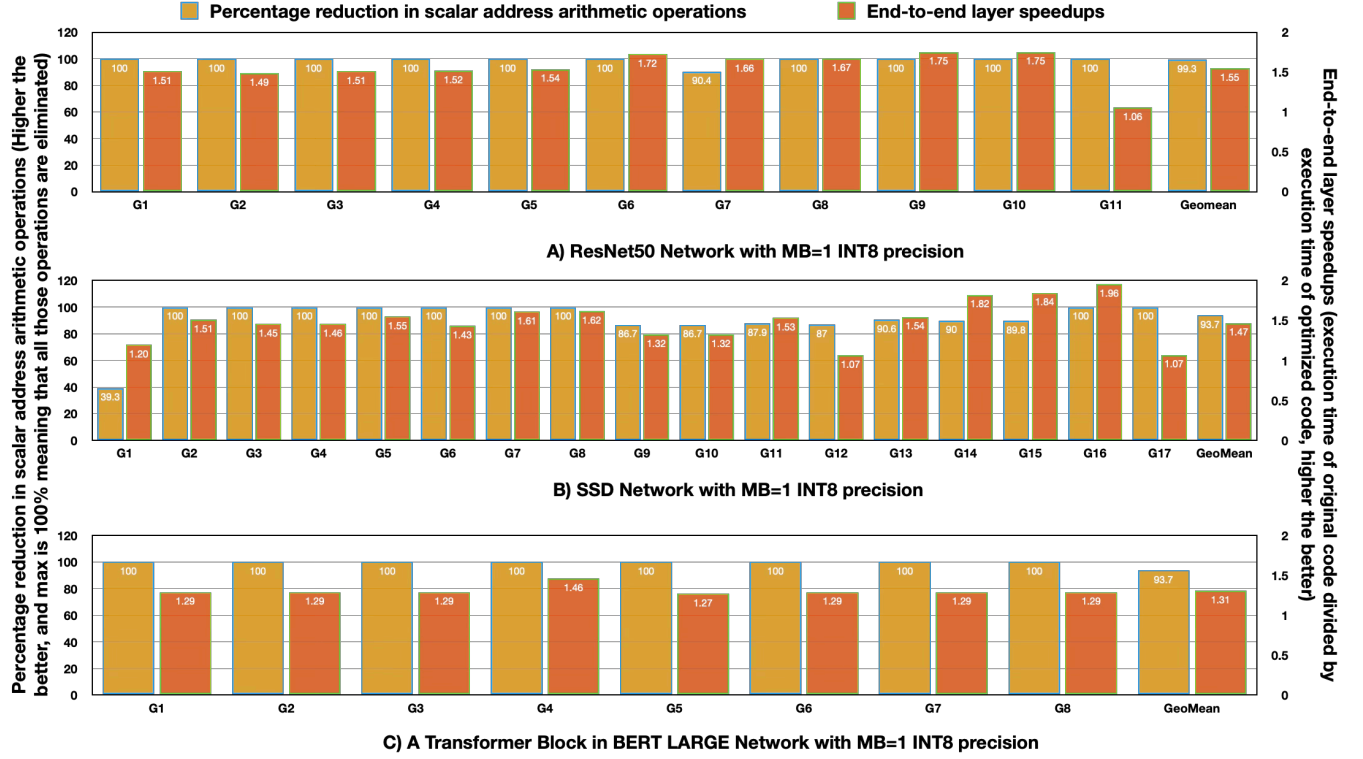


Figure 8: Per-layer-group percentage reduction in scalar address arithmetic operations (left axis) and corresponding end-to-end layer speedups (right axis) achieved by AAA+AAH relative to the baseline, for (A) ResNet-50, (B) SSD, and (C) a BERT Large transformer block. Layers with similar access patterns are grouped on the x-axis; the rightmost bar in each subplot shows the geometric mean across all groups. The specific layers comprising each group are listed in fig. 6.

stwu), ARM pre/post-index with writeback, and others. Any architecture exposing such instructions, and whose compiler IR makes the pre-increment and post-update fields explicitly manipulable, is a direct candidate for our optimizations.

In general-purpose programs, the effectiveness of our approach may be more limited. Our chain construction conservatively terminates at any operation it cannot classify, such as conditionals, function calls, multi-use values, or operations outside the loop, so address chains spanning complex control flow will not be optimized. Furthermore, the density of optimization opportunities depends on whether an upstream pass such as live range reduction constructs the single-def/single-use chains that absorption requires; without such a pass, fewer chains are exposed and gains will be modest, as confirmed by our without-LRR results (section 5.2). Functionally, the transformation is always correct and safe to apply — it simply produces no change when no eligible chains exist. The performance benefit is therefore architecture- and workload-dependent, and is highest in structured, loop-heavy programs found in ML workloads dominated by address arithmetic, such as the load/store programs in decoupled accelerators that motivated this work.

Although some of these accelerators expose coarse-grained DMA engines, scalar instructions remain essential for completing the address arithmetic required by the diverse tensor traversals the scheduler generates — coarser-grained instructions alone cannot fully capture this arithmetic. Consequently, our approach applies directly whenever a given hardware’s coarser-grained instructions

can be represented as composite memory operations (including time-wrapped variants such as bursts, discussed in section 2.3); where this is not possible, our analysis and transformation conservatively skip that region of code (e.g., line 13 of AAA) rather than risk an incorrect transformation. Notably, our evaluation shows that converting coarser-grained movement instructions into finer-grained ones significantly improves performance by exposing additional opportunities for AAA and AAH (Observation 3).

6 Conclusions & Future Work

Address generation is a critical performance concern for compilers targeting decoupled access-execute dataflow accelerators: We presented two complementary compiler techniques—Address Arithmetic Absorption (AAA) and Address Arithmetic Hoisting (AAH)—for addressing the performance concern. Evaluated on ResNet-50, SSD, and BERT Large on the IBM Spyre accelerator, the combined AAA+AAH approach eliminates more than 90% of scalar address arithmetic from generated L0-LU programs—99.3% for ResNet-50, 100% for BERT Large, and 93.7% for SSD—and achieves geometric mean speedups of 1.65×, 1.66×, and 1.66× respectively over a strong baseline, substantially outperforming a peephole instruction-combining baseline across all workloads and layer groups.

References

- [1] 2024. New IBM Processor Innovations To Accelerate AI on Next-Generation IBM Z Mainframe Systems. (2024). <https://newsroom.ibm.com/ai-on-z>

- [2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- [3] Bowen Alpern, Mark N Wegman, and F Kenneth Zadeck. 1988. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1–11.
- [4] ARM Limited. 2013. *ARM Architecture Reference Manual: ARMv8, for ARMv8-A Architecture Profile*. ARM Limited. <https://developer.arm.com/documentation/ddi0487>.
- [5] Chris Berry. 2024. IBM Telum® II processor and IBM Spyre Accelerator chip for AI. (2024). https://hc2024.hotchips.org/assets/program/conference/day1/04_HC2024.IBMCBerry.final.pdf
- [6] Preston Briggs and Keith D. Cooper. 1994. Effective Partial Redundancy Elimination. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI '94)*, Barbara G. Ryder and Mary Lou Soffa (Eds.). Association for Computing Machinery, New York, NY, USA, 159–170. doi:10.1145/178243.178257
- [7] Preston Briggs, Keith D Cooper, and L Taylor Simpson. 1997. Value numbering. *Software: Practice and Experience* 27, 6 (1997), 701–724.
- [8] Prasanth Chatarasi, Alex Gatea, Bardia Mahjour, Jintao Zhang, Alberto Mannari, Chris Bowler, Shubham Jain, Masoud Ateei Jaliseh, Nicole Khoun, Kamlesh Kumar, Viji Srinivasan, and Swagath Venkataramani. 2026. Eliminating Redundancy: Ultra-compact Code Generation for Programmable Dataflow Accelerators. In *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 44–56. doi:10.1109/CGO68049.2026.11395195
- [9] Prasanth Chatarasi, Alex Gatea, Wei Wang, Chris Bowler, Shubham Jain, Masoud Ateei Jaliseh, Nicole Khoun, Alberto Mannari, Bardia Mahjour, Viji Srinivasan, and Swagath Venkataramani. 2026. Enabling Spill-Free Compilation via Affine-Based Live Range Reduction Optimization. In *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–13. doi:10.1109/CGO68049.2026.11395221
- [10] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *International Symposium on Computer Architecture (ISCA)*.
- [11] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308.
- [12] Keith D Cooper, Kathryn S McKinley, and Linda Torczon. 1998. Compiler-Based Code-Improvement Techniques.
- [13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL] <https://arxiv.org/abs/1810.04805>
- [14] Amin Firoozshahian, Joel Coburn, Roman Levenstein, Rakesh Nattoji, Ashwin Kamath, Olivia Wu, Gurdeepak Grewal, Harish Aepala, Bhasker Jakka, Bob Dreyer, Adam Hutchin, Utku Diril, Krishnakumar Nair, Ehsan K. Aredestani, Martin Schatz, Yuchen Hao, Rakesh Komuravelli, Kunming Ho, Sameer Abu Asal, Joe Shajrawi, Kevin Quinn, Nagesh Sreedhara, Pankaj Kansal, Willie Wei, Dheepak Jayaraman, Linda Cheng, Pritam Chopda, Eric Wang, Ajay Bikumandla, Arun Karthik Sengottuvel, Krishna Thottempudi, Ashwin Narasimha, Brian Dodds, Cao Gao, Jiyuan Zhang, Mohammed Al-Sanabani, Ana Zehtabioskuie, Jordan Fix, Hangchen Yu, Richard Li, Kaustubh Gondkar, Jack Montgomery, Mike Tsai, Saritha Dwarakapuram, Sanjay Desai, Nili Avidan, Poorvaja Ramani, Karthik Narayanan, Ajit Mathews, Sethu Gopal, Maxim Naumov, Vijay Rao, Krishna Noru, Hari Krishna Reddy, Prahlad Venkatapuram, and Alexis Bjorlin. 2023. MTIA: First Generation Silicon Targeting Meta's Recommendation Systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (Orlando, FL, USA) (ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 80, 13 pages. doi:10.1145/3579371.3589348
- [15] B. Fleischer, S. Shukla, M. Ziegler, J. Silberman, J. Oh, V. Srinivasan, J. Choi, S. Mueller, A. Agrawal, T. Babinsky, N. Cao, C. Chen, P. Chuang, T. Fox, G. Grinstead, M. Guillorn, H. Haynie, M. Klaiber, D. Lee, S. Lo, G. Maier, M. Scheuermann, S. Venkataramani, C. Vezirtzis, N. Wang, F. Yee, C. Zhou, P. Lu, B. Curran, L. Chang, and K. Gopalakrishnan. 2018. A Scalable Multi-TeraOPS Deep Learning Processor Core for AI Trainina and Inference. In *2018 IEEE Symposium on VLSI Circuits*. 35–36. doi:10.1109/VLSI.2018.8502276
- [16] Free Software Foundation. [n. d.]. GCC Internals Manual. <https://gcc.gnu.org/onlinedocs/gccint/>. Accessed: 2026.
- [17] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. 2019. Xilinx Adaptive Compute Acceleration Platform: Versal™ Architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (Seaside, CA, USA) (FPGA '19)*. ACM, New York, NY, USA, 84–93. doi:10.1145/3289602.3293906
- [18] LI HAOKUN. 2024. Survey About Value Numbering Optimization. In *2024 21st International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)*. 1–4. doi:10.1109/ICCWAMTIP64812.2024.10873757
- [19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV] <https://arxiv.org/abs/1512.03385>
- [20] K Hegde, R Agrawal, Y Yao, and CW Fletcher. 2018. Morph: Flexible Acceleration for 3D CNN-Based Video Understanding. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 933–946.
- [21] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. Extensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 319–333.
- [22] Intel Corporation. 2023. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation. <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html> Instruction Set Reference, including MOV instruction.
- [23] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. 2021. Ten Lessons From Three Generations Shaped Google's TPUv4i: Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. doi:10.1109/ISCA52012.2021.00010
- [24] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (Williamsburg, VA, USA) (ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 461–475. doi:10.1145/3173162.3173176
- [25] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 75–86.
- [26] 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 *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308
- [27] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. *SSD: Single Shot MultiBox Detector*. Springer International Publishing, 21–37. doi:10.1007/978-3-319-46448-0_2
- [28] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.
- [29] NVIDIA. 2018. NVIDIA Deep Learning Accelerator (NVDLA). <https://nvidia.org>.
- [30] OpenPOWER Foundation. 2022. *Power ISA Version 3.1B*. OpenPOWER Foundation. <https://openpowerfoundation.org/specifications/isa/> Includes load/store update instructions such as lwzu.
- [31] Angshuman Parashar, Prasanth Chatarasi, and Po-An Tsai. 2021. Hardware abstractions for targeting EDDO Architectures with the Polyhedral Model. In *11th International Workshop on Polyhedral Compilation Techniques (Vitalur Event) (IMPACT 2021)*.
- [32] Raghu Prabhakar, Sumti Jairath, and Jinuk Luke Shin. 2022. SambaNova SN10 RDU: A 7nm Dataflow Architecture to Accelerate Software 2.0. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. 350–352. doi:10.1109/ISSCC42614.2022.9731612
- [33] LLVM Project. [n. d.]. Scalar Evolution Analysis — LLVM Documentation. <https://llvm.org/docs/ScalarEvolution.html>. Accessed: 2024-05-13.
- [34] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. 2019. Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 14–27.
- [35] James E Smith. 1982. Decoupled access/execute computer architectures. *ACM SIGARCH Computer Architecture News* 10, 3 (1982), 112–119.
- [36] Rishi Surendran, Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. 2014. Iteration Scalar Replacement Using Array SSA Form. In *Compiler Construction*, Albert Cohen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 40–60.
- [37] Swagath Venkataramani, Jungwook Choi, Vijayalakshmi Srinivasan, Wei Wang, Jintao Zhang, Marcel Schaal, Mauricio J. Serrano, Kazuaki Ishizaki, Hiroshi Inoue, Eri Ogawa, Moriyoshi Ohara, Leland Chang, and Kailash Gopalakrishnan. 2019. DeepTools: Compiler and Execution Runtime Extensions for RaPiD AI Accelerator. *IEEE Micro* 39, 5 (2019), 102–111. doi:10.1109/MM.2019.2931584
- [38] Swagath Venkataramani, Vijayalakshmi Srinivasan, Wei Wang, Sanchari Sen, Jintao Zhang, Ankur Agrawal, Monodeep Kar, Shubham Jain, Alberto Mannari, Hoang Tran, Yulong Li, Eri Ogawa, Kazuaki Ishizaki, Hiroshi Inoue, Marcel Schaal, Mauricio Serrano, Jungwook Choi, Xiao Sun, Naigang Wang, Chia-Yu Chen, Allison Allain, James Bonano, Nianzheng Cao, Robert Casatuta, Matthew Cohen, Bruce Fleischer, Michael Guillorn, Howard Haynie, Jinwook Jung, Mingyu Kang, Kyu-hyoun Kim, Siyu Koswatta, Saekyu Lee, Martin Lutz, Silvia Mueller, Jinwook Oh, Ashish Ranjan, Zhibin Ren, Scot Rider, Kerstin Schelm, Michael Scheuermann, Joel Silberman, Jie Yang, Vidhi Zalani, Xin Zhang, Ching Zhou, Matt Ziegler, Vinay Shah, Moriyoshi Ohara, Pong-Fei Lu, Brian Curran, Sunil Shukla, Leland Chang, and Kailash Gopalakrishnan. 2021. RaPiD: AI Accelerator for Ultra-low Precision Training and Inference. In *2021 ACM/IEEE 48th Annual International Symposium*

Prasanth Chatarasi, Nicole Khoun, Wei Wang, Chris Bowler, Alex Gatea, Shubham Jain, Masoud Ataei Jaliseh, Alberto Mannari, Bardia Mahjour, Viji Srinivasan, and Swagath Venkataramani

on Computer Architecture (ISCA). 153–166. doi:10.1109/ISCA52012.2021.00021