Ember: A Compiler for Embedding Operations on Decoupled Access-Execute Architectures

Marco Siracusa[1], Olivia Hsu[2,3], Victor Soria–Pardos[1], Joshua Randall[4], Arnaud Grasset[5], Eric Biscondi[5], Doug Joseph[4], Randy Allen[1], Fredrik Kjolstad[2], Miquel Moretó Planas[6,1], Adrià Armejach[6,1]
[1]Barcelona Supercomputing Center, Barcelona, Spain
[2]Stanford University, Stanford, USA
[3]Carnegie Mellon University, Pittsburgh, USA
[4]Arm, Austin, USA
[5]Arm, Biot, France
[6]Universitat Politècnica de Catalunya, Barcelona, Spain

***Abstract*—Decoupled Access-Execute (DAE) architectures separate memory accesses from computation in two specialized units. This design is becoming increasingly popular among hyperscalers to accelerate irregular embedding lookups in recommendation models. In this paper, we first broaden the scope by demonstrating the benefits of DAE architectures across a wider range of irregular embedding operations in several machine learning models. Then, we propose the Ember compiler to automatically compile all of these embedding operations to DAE architectures. Conversely from other DAE compilers, Ember features multiple intermediate representations specifically designed for different optimization levels. In this way, Ember can implement all optimizations to match the performance of hand-written code, unlocking the full potential of DAE architectures at scale.**

***Index Terms*—Recommendation models, Large language models, Graph learning models, Tensor compilers, Hardware–software co-design, Machine learning accelerators**

I. Introduction

Machine learning models represent categorical features—such as users, products, and words—using dense embedding vectors [1]. Due to the vast number of categories, interactions between these features are inherently sparse [2], [3], [4]. A prime example is recommendation models, which must look up the embedding vectors of the products a user has interacted with, often loading hundreds of vectors scattered among millions [5]. Overall, these irregular lookups are a critical bottleneck in traditional architectures such as CPUs and GPUs (Section II).

Decoupled Access-Execute (DAE) architectures address this challenge by separating memory access from computation into two specialized units [6], [7], [8], [9], [10]. Recent designs such as the Google TPU [11] and Meta MTIA [12] demonstrate that DAE can substantially accelerate irregular embedding lookups in recommendation models. Yet, the broader potential of DAE for general embedding operations, and the compiler support required to harness it, remains mostly unexplored (Section IX). This paper closes that gap.

Our **first contribution** is to empirically demonstrate the potential of DAE architectures beyond recommendation models. In Section III we demonstrate that DAE architectures can outperform CPUs and GPUs on a large set of embedding operations in multiple machine learning models.

Our **second contribution** is a DAE programming model for general embedding operations. In DAE architectures, control flow and data flow are handled by the access program and (de)serialized to the execute program through queues. Not all workloads map to such a programming model. In Section IV we demonstrate that, by interpreting general embedding operations as tensor algebra, they can map to DAE architectures.

The **third and main contribution** of this paper is Ember, a DAE tensor compiler to automatically generate high-performance DAE code directly from PyTorch [13] embedding operations. Manually writing DAE code for all combinations of embedding operations, algorithmic variants, and input formats is unfeasible and, therefore, requires an automated solution. As explained in Section V, a DAE compiler needs to automatically decouple memory accesses from computation and generate *optimized* code for the access unit and execute unit. However, optimizing across access and execute code is challenging. On one hand, such global optimizations require the control flow and data flow of the input program, which are lost when memory accesses are separated from computation and serialized through queues. On the other hand, optimizing DAE code before separation requires an Intermediate Representation (IR) that knows which operations will run on the access unit and compute unit, and how data will be (de)serialized across units. However, no existing IR can naturally represent this.

Hence, Ember generates optimized DAE code by lowering embedding operations through multiple *novel* DAE IRs, each designed for different DAE optimization levels. In this way, Ember supports the necessary optimizations to match the performance of hand-optimized code, enabling the full potential of DAE architectures at no programmability cost.

The goal of this paper is to motivate Ember and discuss its main design points. We refer to Ember's documentation and codebase for further implementation details[1].

II. Embedding Ops on Traditional Architectures

This section reviews embedding operations in modern machine learning models and highlights their performance limitations on conventional CPUs and GPUs.

[1]Available at: https://github.com/bsc-loca/ember

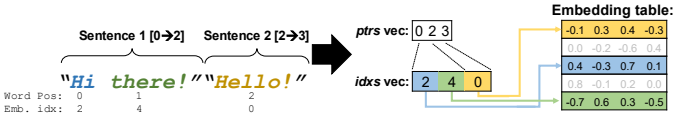
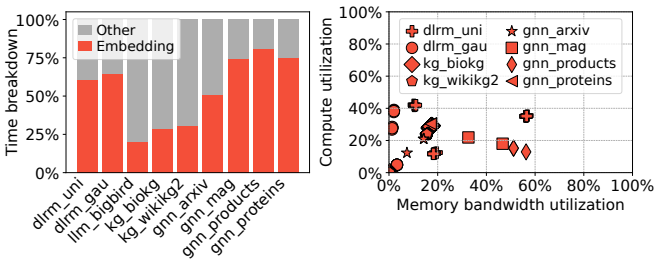


Fig. 1: Example of word embedding for a batch of two sentences [1]. Each token is represented with an ID, which is used to look up its embedding vector in the embedding table. The `idxs` vector stores the IDs of all tokens of all sentences in a batch whereas the `ptrs` vector stores the position each sentence begins at. Overall, feature embedding requires scattered memory lookups to fetch embedding vectors from embedding tables.



(a) Runtime breakdown (b) Resource Utilization

Fig. 2: Several machine learning models heavily rely on embedding operations that do not perform efficiently even on modern Nvidia H100 GPUs [18]. All experiments use highly-optimized models from the literature (Appendix A).

A. Embedding Operations

Modern machine learning models process an increasing number of *categorical features* [14]—representation variables that can take on a distinct set of values and are not comparable. Examples include product categories, words, and actors in a movie. Since these categories lack numerical meaning, they cannot be directly processed by deep neural networks [5]. Machine learning models encode a numerical representation of these categories into dense *embedding vectors* [1], which are stored in large *embedding tables*. Figure 1 illustrates how categorical features are processed during inference. Incoming categories are tokenized into IDs, which are used to *lookup* into the embedding table. Embedding lookup is generally fused with downstream computation in an optimized *embedding operation*. Table I lists the main embedding operations in common machine learning models, which we describe below.

1) Deep-Learning Recommendation Models (DLRMs): DLRMs recommend content to users by processing a large set of categorical features [5], [14]. During inference, the embedding vectors corresponding to the active categories in a feature (e.g. movie cast) need to be looked up and aggregated with, for instance, an element sum. The `nn.EmbeddingBag` (EB) is a PyTorch implementation of such a function [13].

2) Sparse Attention Mechanisms: Large Language Models (LLMs) embed text as vector representations [15]. Transformers such as Google's BigBird [3] sparsify the attention mechanism, restricting comparisons to a selected subset of input tokens to reduce memory footprint and accelerate processing on long sequences. While this technique reduces inference latency, it introduces an additional step to gather blocks of embeddings, which is generally not fused with downstream deep neural networks [3] and involves embedding lookups with no compute.

3) Graph Machine Learning Models: Graph machine learning models such as Graph Neural Networks (GNNs) [4], Message Passing (MP) models [16], and Knowledge Graphs (KGs) [17] embed node or edge features of a graph into vectors. For example, GNNs chain neural layers and graph convolutions, which gather neighbor embeddings via an SpMM-like operation shown in Table I.

B. Challenges on Traditional Architectures

Overall, embedding operations involve looking up embedding vectors from scattered memory locations and aggregating them with elementwise operations. Figure 2 shows how these critical operations result in low operational intensities and low system utilization, even on state-of-the-art machine learning accelerator architectures like GPUs.

GPUs, specifically, are designed for massively parallel and regular computation where (1) memory accesses can be coalesced and (2) there is enough computation to hide memory latency [19]. However, (1) embedding vectors are only hundreds of elements long, limiting coalescing opportunities, and (2) elementwise operations do not provide enough compute to hide memory latency, resulting in low GPU utilization [20].

CPUs are also not optimized for embedding lookups [21]. Irregular lookups cannot be efficiently cached nor prefetched, causing long memory accesses that clog the pipeline of off-the-shelf CPU cores, stalling the whole execution [2], [14], [22]. Hence, we explore alternative designs in the next sections.

III. The Potential of DAE Architectures

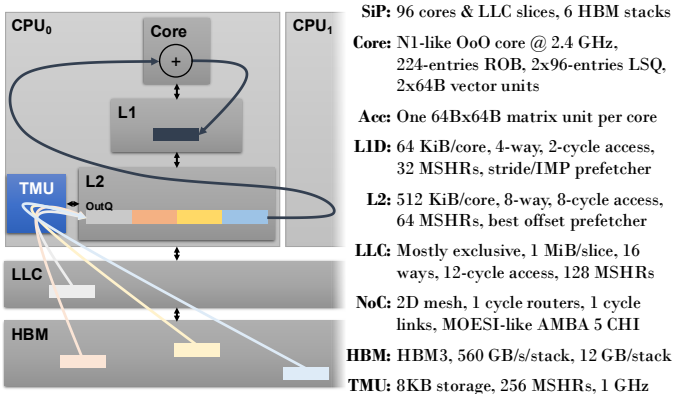
Machine learning accelerators such as the Google TPU [11] and Meta MTIA [12] speed up recommendation models by offloading embedding lookups to dedicated access units. This separation of memory access from computation is known as a Decoupled Access-Execute (DAE) architecture. In this section, we broaden the scope and demonstrate the potential of DAE architectures across all the models introduced earlier.

We evaluate the broad performance potential of DAE architectures through the system in Figure 3, a multicore DAE processor. The DAE processor is a traditional high-performance processor that integrates one access unit per core—the Tensor Marshaling Unit (TMU) [7]—to offload critical tensor traversal and embedding lookups from the core. As discussed in Section IV, the core can program the TMU to traverse sparse tensors, load embedding vectors, and send them to the core for computation. While similar systems have demonstrated a large potential in scientific computation [10], graph analytics [8], and other fields [9], no prior work has studied their impact on general embedding operations.

We conduct all experiments using full-system gem5 simulations [23] for performance evaluation and McPAT [24] for power analysis, with further experimental methodology details

TABLE I: Embedding operations in main machine learning models.

EmbeddingBag (EB) for Deep-Learning Recommendation Models (DLRMs) (Section II-A1)	Sparse attention (SpAttn) for Large Language Models (LLMs) (Section II-A2)	Sparse Matrix-Matrix Multiplication (SpMM) for Graph Neural Networks (GNNs) (Section II-A3)	Message Passing (MP) models (Section II-A3)	Knowledge Graphs (KGs) (Section II-A3)
∀ segment in batch ∀ category in segment **lookup** vector **accumulate** vector	∀ block in Q input ∀ block in K input ∀ token in K block **lookup** K vector ∀ token in Q block **copy** K vector	∀ vertex in graph ∀ neighbor of vertex **lookup** vector **rescale** vector **accumulate** vector	∀ vertex in graph **lookup** vertex vector ∀ neighbor of vertex **lookup** neighbor vector **multiply** vertex-neighbor vectors **scale** and **accumulate** vector **multiply** and **accumulate** vector	∀ sample in batch **lookup** vector **compute** h-r-l norm



(a) (b)

Fig. 3: A DAE processor. Each traditional core offloads embedding lookups to an access unit like the TMU [7].

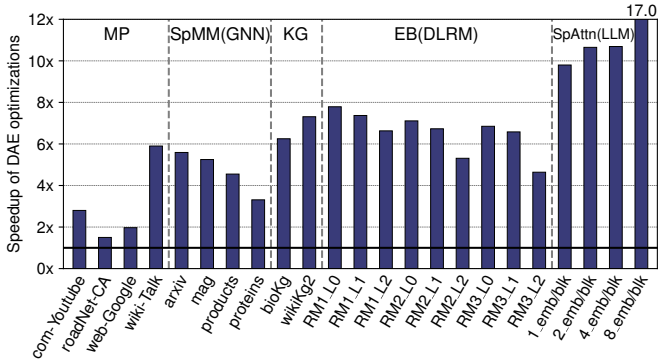
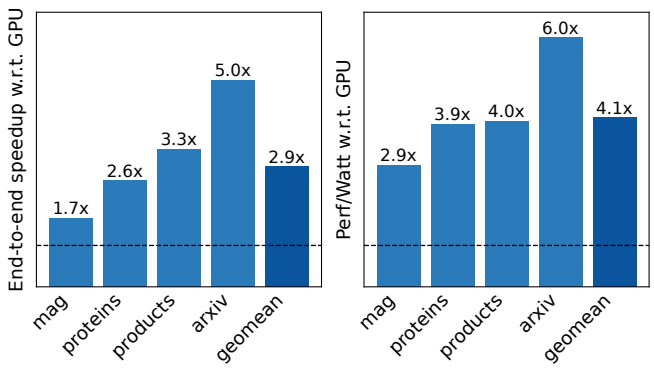


Fig. 4: Performance of a DAE multicore processor (CPU+TMU) vs. an off-the-shelf multicore processor (CPU-only). Offloading lookups to a specialized access unit (the TMU) improves performance of embedding operations by 5.8× on average. Appendix A details the tested algorithms and inputs.

provided in Appendix A. Figure 4 shows the performance of the processor in Figure 3 when tensor traversal and embedding lookups are offloaded to the TMU (i.e. DAE computation) compared to when executed directly in the core (i.e. traditional computation). In the end, offloading tensor traversals and embedding lookups to the TMU improves performance by an average of 5.8× across the embedding operations presented in



(a) Inference latency (runtime) (b) Efficiency

Fig. 5: The DAE processor outperforms the Nvidia H100 GPU on OGB GNNs both in inference latency (a) and perf/watt (b).

Table I. In this way, as shown in Figure 5, the DAE processor also outperforms GPUs on end-to-end models by 2.9× in performance and 4.1× in performance per watt.

Overall, decoupling embedding lookup and computation into two distinct units allows for independent specialization and optimization. The TMU implements traversal and load logic in dataflow hardware, enabling higher memory throughput and efficiency than that of traditional compute cores [7]. The remainder of this paper discusses the programmability challenges introduced by this decoupling, and how the Ember compiler addresses them.

IV. DAE Abstraction for Embedding Operations

This section introduces our DAE programming model, which we formalize as the Decoupled Lookup-Compute (DLC) IR. The DLC IR provides the necessary components to program embedding operations on DAE architectures. As we discuss in the next section, the DLC IR also serves as Ember's low-level IR, enabling a clean separation between target-agnostic optimizations and target-specific code generation.

Figure 6 shows our abstraction for DAE architectures like the TMU–CPU system and others [8], [9], [10]. This abstraction consists of (1) an access unit specialized for embedding lookups, (2) an execute unit for computation, and (3) data and control queues that stream values and commands from the access unit to the execute unit.

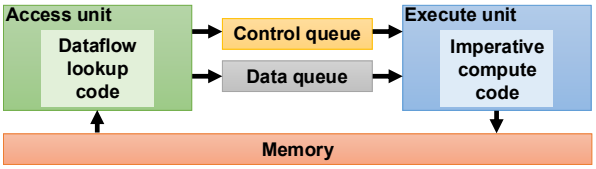


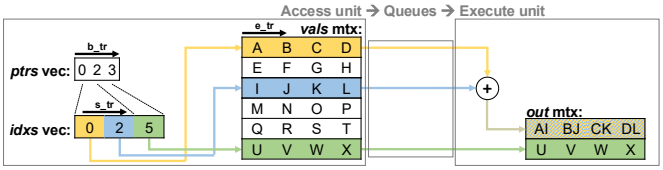
Fig. 6: DAE architectural abstraction.

To map embedding operations to such a DAE abstraction, we observe that general embedding operations are expressible as variants of sparse-dense tensor multiplications. For instance, the example EB function in Figure 7a can be interpreted as a sparse-dense matrix multiplication (SpMM) $Z_{i,j} = A_{i,k}B_{k,j}$, with an ikj loop schedule, and a sparse matrix stored in a Compressed Sparse Row (CSR) format [25]. For the example in Figure 1, dimension i of the sparse matrix A contains the sentences in a batch whereas dimension k contains the possible words in a sentence. $A_{pq} = 1$ if the sentence p contains the word q, and 0 otherwise. The CSR format stores the indices of the non-zero elements and the pointers where each sentence begins. Multiplying the sparse matrix A by the dense matrix B accumulates all embedding vectors of the words in each sentence, which is the imperative code for EB in Figure 7b. Non-zero values beyond one rescale the embedding vector, which is needed in GNNs (Section II-A3). MP models are a Sampled-Dense-Dense Matrix Multiplication (SDDMM) fused with an SpMM kernel [16]. KGs are EB functions that use semirings with just one non-zero per row and do not require segment pointers or lengths. Sparse attention mechanisms (SpAttn) are similar to KGs but have a blocked tensor format and no computation.

Sparse-dense tensor multiplications are linear combinations of one sparse and any number of dense tensors. This yields a linear iteration space that can be *traversed* with a hierarchy of for-loops without additional conditional branches. Computation—typically elementwise reductions [16]—occurs only at the beginning, within iterations, or at the end of each loop [26]. Since output tensors differ from input tensors, read-after-write conflicts do not arise.

Because of these properties, sparse-dense tensor operations can be expressed as a dataflow chain of access and compute blocks [27]. We can map lookup blocks to the access unit and compute blocks to the execute unit, and stream data and commands through the queues [7]. The DLC IR represents lookup blocks with streaming dataflow code, as it maps well to spatial architectures for memory-intensive workloads [27], [7], [8], [28], [29], [30], [31], [32]. Compute blocks, instead, are represented with imperative code as it can implement all the compute variants introduced above, and other general-purpose computation. Figure 7c and 7e show the DLC IR for the EB function, which we discuss in the reminder of this section.

The DLC lookup code loads embedding elements through memory streams. The IR uses index streams to maintain the iteration space for these memory streams, which are generated by traversal operators and can be transformed with integer



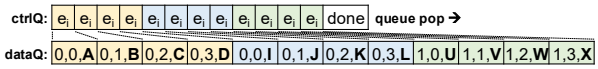
(a) Functional example of the EB function.

```
1 void eb(idxs: mref<? x idx>,
2         ptrs: mref<? x idx>,
3         vals: mref<? x ? x f32>,
4         out: mref<? x ? x f32>){
5  for(int b = 0; b < num_batches; b++){ // Batch trav. (b_tr)
6    for(int p = ptrs[b]; p < ptrs[b+1]; p++){ // Segment trav. (s_tr)
7      i = idxs[p]; // Load embedding index
8      for(int e = 0; e < emb_len; e++){ // Embedding vec trav. (e_tr)
9        val = vals[i,e]; // Load embedding element
10       out[b,e] += val; }}}} // Reduce embedding elements
```

(b) Traditional imperative code, which also acts as input to Ember.

```
1  tu b_tr=loop_tr(0, num_batches, 1) // iterate thru batch
2  str beg_ptr=b_tr.mem_str(ptrs, b_tr.ite) // load segment beg ptr
3  str end_pos=b_tr.alu_str('+', b_tr.ite, 1) // compute next ptr pos
4  str end_ptr=b_tr.mem_str(ptrs, end_pos) // load segment end ptr
5
6  tu s_tr=loop_tr(beg_ptr, end_ptr, 1) // iterate thru segment
7  str emb_idx=s_tr.mem_str(s_tr, idxs, s_tr.ite) // load emb idx
8  str emb_beg=s_tr.alu_str(s_tr, '*', emb_idx, emb_len) // emb addr
9
10 tu e_tr=loop_tr(0, emb_len, 1) // iterate thru emb vec
11 str emb_pos=e_tr.alu_str('+', emb_beg, e_tr.ite) // element addr
12 str emb_val=e_tr.mem_str(vals, emb_pos) // load emb element
13
14 enq_op(b_tr.ite, e_tr, ITE) // enqueue batch position
15 enq_op(e_tr.ite, e_tr, ITE) // enqueue emb el position
16 enq_op(emb_val, e_tr, ITE) // enqueue emb el value
17 callback(e_tr, ITE) // trigger call on every emb iter
```

(c) DLC dataflow lookup code.



(d) DAE queue content for our EB example.

```
1 while(ctrlQ.deq() == e_i){ // while some element
2   b = dataQ.deq<1 x i32>(); // read embedding position
3   e = dataQ.deq<1 x i32>(); // read element position
4   v = dataQ.deq<1 x f32>(); // read element value
5   add(&out[b,e],v); // increment and store
```

(e) DLC imperative compute code.

Fig. 7: DAE abstraction for the EmbeddingBag (EB) operation.

ALU streams. Formally:

- **`loop_tr(lb,ub,stride)`**: traverses the iteration space `i=lb; i<ub;i+=stride`, where `lb` and `ub` are streams or immediate values and `stride` is an immediate value. The stream `loop_tr.0` contains the induction variable.
- **`mem_str(base,idx)`**: loads into a stream the values of the `base` memory locations indexed by the `idx` stream.
- **`alu_str(op,op1,op2)`**: computes an integer binary operation `op` $\in \{+, -, \times, \div\}$ on operands `op1` and `op2`, which can either be streams or immediate values.

Concretely, we show these streams in the lookup code for the EB example in Figure 7c. Lines 1–12 in Figure 7c are the DLC representation of lines 5–9 in Figure 7b, as emitted by the Ember compiler.

Compute code consists of callbacks that the access unit triggers while traversing and loading tensor operands. In Figure 7b, we can wrap the compute operation in line 10 into a callback that the access unit can trigger at each iteration

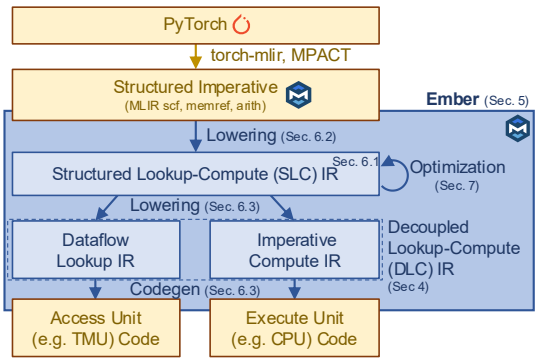


Fig. 8: Overview of Ember. Yellow represents compiler inputs and outputs and blue represents new contributions.

Any CPU Statement	$STMT$	*Memory Reference*	$mref$
CPU variable	var	*Stream Variable*	s
Unsigned Integer	uint		

For-Loop	FOR	::=	for $(HEAD)$ { $SDEC^*$ $BODY$ }
For Header	$HEAD$	::=	$s = r$ to r step uint
Stream Declaration	$SDEC$	::=	$s = se$
Loop Body	$BODY$	::=	$CALL \mid CALL?\ FOR\ CALL?$
Callback	$CALL$	::=	callback() { $TVAL^*\ STMT^+$ }
Value Conversion	$TVAL$	::=	var = to_val(s)
Loop Range	r	::=	$s \mid var \mid$ uint
Stream Expression	se	::=	$ls \mid as \mid cs \mid bs$
Load Stream	ls	::=	load_str($mref[indices^*]$, $uint$)
Indices	$indices$	::=	$s \mid var \mid$ uint
ALU Stream	as	::=	alu_str(sop, s_1, s_2)
Stream Ops	sop	::=	$+ \mid - \mid * \mid / \mid \%$	...

Fig. 9: The Structured Lookup-Compute (SLC) IR.

of its parent loop. To trigger these callbacks, the access unit enqueues its corresponding token (the inner-loop iteration token (e_i)) in the control queue and its operands (result coordinates b and e and value v) in the data queue. Lines 14–17 in Figure 7c and lines 2–4 in Figure 7e show how these tokens and data are enqueued and dequeued from the queues, and the exact queue content is shown in Figure 7d. Figure 11d demonstrates a more complex example with multiple callbacks. Formally, we can program the access unit to marshal control tokens and operands at three traversal locations—at each iteration (e_i), the beginning (e_b), and the end (e_e) of a loop—through the following operations:

- **`enq_op(s_id,tu_id,event)`**: enqueues into the data queue (dataQ) the contents of the `s_id` stream every time a traversal `event` $\in \{$`beg`, `ite`, `end`$\}$ occurs in `tu_id`.
- **`callback(tu_id,event)`**: enqueues into the control queue (ctrlQ) the corresponding control token every time a traversal `event` $\in \{$`beg`, `ite`, `end`$\}$ occurs in `tu_id`.

whereas the execute unit reads tokens and operands with:

- **`deq()`**: dequeues a control token from the data queue.
- **`deq<ty>()`**: dequeues a value with `ty` type from dataQ.

Appendix B discusses how our DLC IR is sufficient to express general embedding operations on various DAE architectures from the literature beyond the TMU.

Ultimately, although separating embedding lookups from computation substantially improves performance and efficiency, it also introduces significant complexity into the DAE programming model, hindering the widespread adoption of DAE architectures. In the remainder of this paper, we demonstrate how Ember addresses this challenge by automatically targeting the DLC IR from normal imperative code and compiling down to DAE hardware code.

V. Ember Overview

Figure 8 shows our MLIR [33] implementation of Ember. Ember compiles structured representations of embedding operations into optimized DLC code, which is then used for DAE code generation. Appendix C discusses the advantages of lowering from structured IRs and additional considerations when lowering from other IRs. The Structured Control Flow (SCF) Dialect is perhaps the most popular structured IR in MLIR, and resembles the imperative code in Figure 7b. Ember generates SCF from PyTorch embedding operations or general tensor algebra expressions with tools like torch-mlir [34] or MPACT [35], respectively.

To generate optimized DAE code from SCF, Ember needs to decouple memory accesses from computation and generate optimized code for the access unit and execute unit. However, the main challenge of this process is that DAE code cannot be optimized in the DLC IR nor in SCF. Global optimizations across access and execute code need the control flow and data flow of the entire input program, which is already lost and decoupled in the DLC IR. For instance, it would be impractical to identify what stream generates the `v` value in the DLC IR in line 5 in Figure 11d, which is an essential analysis for several optimization discussed in Section VII. Instead, while SCF has knowledge of both control flow and data flow, it does not differentiate between access and execute code, nor identify how data is (de)serialized between them. Therefore, global optimizations would need to be integrated in the code separation and/or generation logic, which is impractical for a large set of complex optimizations, like the ones in Section VII, which should execute iteratively [33].

For this reason, we designed the Structured Lookup-Compute (SLC) IR (Section VI-A). The SLC IR is a natural extension of the SCF IR for DAE code. The key feature of the SLC IR is to represent DAE code while preserving control flow and data flow of the input program, enabling iterative DAE optimizations across access and execute code. In this way, Ember decides what SCF operations to map to the two decoupled units and generates the SLC IR (Section VI-B). Then, it performs global optimizations (Section VII) on the SLC IR and lowers it to the DLC IR for separate optimizations of access and execute code (Section VI-C). Access and execute codes are then lowered to target-specific IRs such as the `tmu` and `llvm` dialects, which are then used for code generation (Section VI-D). Each step is presented in the following sections.

VI. SCF Decoupling and Lowering

This section introduces the SLC IR and how Ember lowers SCF and DLC code to and from it, respectively.

A. The SLC IR

Figure 9 shows the SLC grammar, and Figure 10b shows the SLC IR for the EB function example. The main goal of the SLC IR is to abstract the queue (de)serialization logic in the DLC IR, enabling global DAE optimizations.

Similar to the DLC IR, the SLC IR defines access code (i.e. memory access and index arithmetic) through streams and compute code through callbacks. However, unlike the DLC IR, the SLC IR places these streams and callbacks within structured SLC imperative for-loops.

In this way, the values of the streams can be directly accessed from the callbacks through stream-to-value conversion operations, preserving the data flow of the input code. The SLC IR allows Ember to simultaneously optimize lookup and compute code through global analyses and transformations, including dominance analysis, vectorization, and code motion across different compute regions as well as across access and execute code. We later describe these analyses and optimizations that the SLC IR enables in more detail in Section VII.

B. Lowering the SCF IR to the SLC IR

Figure 10 shows a lowering example from SCF to SLC. To generate the SLC IR, Ember recursively traverses the loop hierarchy of the SCF code looking for loops to offload to the access unit. We define an *offloading candidate* as a loop that (1) has iteration bounds which are either static or computed by another offloading candidate and (2) loads from at least one read-only memory location which has not been read from a parent loop. Condition (1) is necessary as access units cannot generally read data from the execute units. Condition (2) excludes workspace loops [36], [37] (i.e. loops that work on partial results and temporaries). As partial results are likely cached, workspace loops do not benefit from memory acceleration. All loops in Figure 10a are offloading candidates. For a more complex example, only the first two loops from the MP pseudocode in Table I are offloading candidates. The last two lines (loops) in MP are workspace loops, as they multiply and accumulate the temporary vector with the vertex vector, which have already been read.

As embedding operations are variants of sparse-dense tensor multiplications, their loop hierarchy can only have at most one offloading candidate per level and at most one workspace loop per level [26], [36]. The decoupling algorithm, therefore, recursively traverses and selects one offloading candidate per level, leaving the other loops for software execution. The SCF offloading candidates are lowered to SLC for-loops.

Then, Ember places callbacks in the SLC loops. Operations to be offloaded (i.e., read-only load operations and index arithmetic) are transformed into streams and moved before their corresponding callback. Operations that should not be offloaded to the access unit are moved inside their corresponding callback, meaning they will be executed in software. For instance, Ember transforms and moves the read-only load operation in line 11 in Figure 10a before the callback, whereas Ember moves the accumulation operations in lines 12 and 13 inside the callback.

```
void eb(idxs: mref<? x idx>,
        ptrs: mref<? x idx>,
        vals: mref<? x ? x f32>,
        out: mref<? x ? x f32>){
 for(idx b=0; b<n_batches; b++){
  idx beg=ptrs[b];
  idx end=ptrs[b+1];
  for(idx p=beg; p<end; p++){
   idx i=idxs[ptr];
   for(idx e=0; e<emb_len; e++){
    f32 val=vals[i,e];
    f32 acc=out[b,e];
    out[b,e]=acc+val; }}}}
```

(a) SCF IR

```
1  void eb(idxs: mref<? x idx>,
2          ptrs: mref<? x idx>,
3          vals: mref<? x ? x f32>,
4          out: mref<? x ? x f32>){
5  slc.for(str b from 0 to n_batches){
6   str beg=slc.mem_str(ptrs[b]);
7   str end=slc.mem_str(ptrs[b+1]);
8   slc.for(str p from beg to end){
9    str i=slc.mem_str(idxs[p]);
10   slc.for(str es from 0 to emb_len){
11    str vals=slc.mem_str(vals[i,e]);
12    slc.callback{
13     idx bv=slc.to_val(b);
14     idx ev=slc.to_val(e);
15     f32 val=slc.to_val(vals);
16     f32 acc=out[bv,ev];
17     out[bv,ev]=acc+val; }}}}}
```

(b) SLC IR

Fig. 10: Lowering from SCF to our SLC IR. Ember converts SCF for-loops into SLC for-loops and SCF memory accesses into SLC streams. Ember wraps computation into SLC callbacks that can access SLC streams with stream-to-value operations.

Finally, stream-to-value operations are added in a callback for each operation reading from a stream.

C. Lowering SLC IR to DLC IR

After performing global optimizations, the SLC IR is lowered to the low-level DLC IR. Ember generates the DLC IR by traversing the SLC IR from outer to inner SLC for-loops. SLC for-loops and streams are lowered to DLC traversal operators and streams. Callbacks, instead, are moved into a while-loop in the compute code. Multiple callbacks are chained into an if-then-else construct that cases the token IDs from the control queue (e.g. Figure 11d). Ember emits enqueue instructions for control tokens based on the location of the callback in the SLC IR and queueing instructions for operands according to the SLC stream-to-value operations.

D. Code Generation

After lowering to the DLC IR, Ember performs separate target-specific optimizations of access and execute code before generating machine code. Ember lowers DLC access code to TMU code by mapping logical streams to physical streams, and it lowers DLC compute code to CPU code by transforming dequeue instructions into load operations and emitting queue handling code.

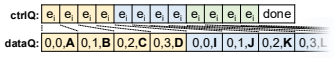
VII. Ember Global Optimizations

This section first describes three key DAE optimizations enabled by the SLC IR. Figures 11 and 12 show how these optimizations transform the SLC IR, compute code, and queues. We also discuss model-specific optimizations and Ember–TMU HW–SW co-design in this section.

A. Vectorization

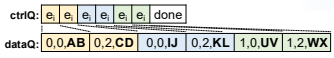
Vectorization is one of the most impactful DAE optimizations [7]. As shown in Figure 11b, for each token, vectorization dequeues a vector of *vector length (vlen)* elements, improving

```
while(ctrlQ.deq() == e_i){
  b = dataQ.deq<1 x idx>();
  e = dataQ.deq<1 x idx>();
  v = dataQ.deq<1 x f32>();
  acc(&out[b,e],v); }
```



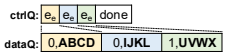
(a) Unoptimized code.

```
while(ctrlQ.deq() == e_i){
  b = dataQ.deq<1 x idx>();
  e = dataQ.deq<1 x idx>();
  v = dataQ.deq<vlen x f32>();
  v_acc(&out[b,e], v);
```



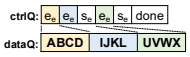
(b) Vectorized code.

```
while(ctrlQ.deq() == e_e){
  b = dataQ.deq<1 x idx>()
  for(e=0; e<emb_len; e+=vlen){
    v = dataQ.deq<vlen x f32>()
    v_acc(&out[b,e],v) }
```



(c) Buffered code.

```
out_ptr = out
while(tkn = ctrlQ.deq() != done){
  if(tkn == e_e){
    for(e=0; e<emb_len; e+=vlen){
      v = dataQ.deq<vlen x f32>()
      v_acc(out_ptr+e, v) }}
  else if(tkn == s_e){
    out_ptr += emb_len }
```



(d) Queue aligned code.

Fig. 11: Impact of SLC optimizations on the DLC EB compute code and output queues.

marshaling and compute efficiency. While auto-vectorizing general code is challenging [38], the SLC IR provides an effective representation to vectorize loading, marshaling, and computing of embedding operations.

As shown in Figure 12b, Ember vectorizes code by converting SLC operations into their vectorized SLCV duals. Conversely from the SLC for-loop, the SLCV for-loop (1) adds a *vector length* attribute, (2) instantiates vectorized induction variables, and (3) introduces the concept of `mask` to handle loop boundaries which are not multiple of the vector length. Masks are used by SLCV streams to perform vectorized index computation and data loading.

We define a *vectorization scheme* as the set of for-loops within a parent loop p and the inner loop i, with $p \geq i$, that we intend to vectorize. A for-loop can be vectorized if and only if all of its callbacks can be vectorized. A vectorization scheme is *legal* if and only if all of its for-loops can be vectorized.

Vector extensions such as Arm SVE [39] provide instructions to vectorize most callbacks in embedding operations, making the space of legal vectorization schemes large. However, prior work has demonstrated that the most efficient vectorization scheme for sparse-dense tensor multiplication is inner-loop vectorization, assuming the dense tensor is in row-major order and has rows which are larger than the vector length [7]. Embedding operations generally satisfy these assumptions. Therefore, like the MLIR sparsifier [40], Ember only attempts inner-loop vectorization.

If the inner for-loop is legal, Ember first vectorizes its access code, then its execute code. As a first step, Ember vectorizes the inner for-loop and its streams. As the stream-to-value operations in the callbacks expect stream types, the algorithm adds a

```
void eb(idxs: mref<? x idx>, offs: mref<? x idx>,
        vals: mref<? x ? x f32>, out: mref<? x ? x f32>){
 //Access: Iterate over segments in a batch
 slc.for(str s_b from 0 to num_batches){
   str s_beg = slc.mem_str(offs[s_b]);
   str s_end = slc.mem_str(offs[s_b+1]);
   //Access: Iterate over embeddings in a segment
   slc.for(str s_ptr from s_beg to s_end){
     str s_idx = slc.mem_str(idxs[s_ptr]);
     //Access: Iterate over embedding vector elements
     slc.for(str s_e from 0 to emb_len){
       str s_val = slc.mem_str(vals[s_idx,s_e]);
       //Execute: Reduce embedding vectors
       slc.callback{
         idx b = slc.to_val(s_b);
         idx e = slc.to_val(s_e);
         f32 val = slc.to_val(s_val);
         f32 acc = out[b,e];
         out[b,e] = acc + val; }}}}}
```

(a) Unoptimized code.

```
void eb(idxs: mref<? x idx>, offs: mref<? x idx>,
        vals: mref<? x ? x f32>, out: mref<? x ? x f32>){
 //Access: Iterate over segments in a batch
 slc.for(str s_b from 0 to num_batches){
   str s_beg = slc.mem_str(offs[s_b]);
   str s_end = slc.mem_str(offs[s_b+1]);
   //Access: Iterate over embeddings in a segment
   slc.for(str s_ptr from s_beg to s_end){
     str s_idx = slc.mem_str(idxs[s_ptr]);
     //Access: Iterate over embedding vector elements
     slcv.for<vlen>((str s_e, str msk) from 0 to emb_len){
       str s_val = slcv.mem_str<vlen>(vals[s_idx,s_e], msk);
       //Execute: Reduce embedding vectors
       slcv.callback{
         idx b = slc.to_val(s_b);
         idx e = slcv.to_val(s_e)[0];
         vec<vlen x f32> val = slcv.to_val<vlen>(s_val);
         vec<vlen x f32> acc = vload<vlen>(out[b,e]);
         vstore<vlen>(acc + val, out[b,e], acc); }}}}}
```

(b) Vectorized code.

```
void eb(idxs: mref<? x idx>, offs: mref<? x idx>,
        vals: mref<? x ? x f32>, out: mref<? x ? x f32>){
 //Access: Iterate over segments in a batch
 slc.for(str s_b from 0 to num_batches){
   str s_beg = slc.mem_str(offs[s_b]);
   str s_end = slc.mem_str(offs[s_b+1]);
   //Access: Iterate over embeddings in a segment
   slc.for(str s_ptr from s_beg to s_end){
     str s_idx = slc.mem_str(idxs[s_ptr]);
     str<vec<vlen x f32>> buf = slcv.buf_str(); // Buffer str
     //Access: Iterate over embedding vector elements
     slcv.for<vlen>((str s_e, str msk) from 0 to emb_len){
       str s_val = slcv.mem_str<vlen>(vals[s_idx,s_e], msk);
       slc.enq(buf, s_val); } // Enqueue into the buffer
     //Execute: Reduce embedding vectors
     slcv.callback{ // Callback moved at the end of inner loop
       idx b = slc.to_val(s_b);
       vec<vlen x f32> buf_vec = slc.to_val(buf); // Get buffer
       for(idx e = 0; e < emb_len; e++){ // Iterate buffer
         vec<vlen x f32> val = buf_vec[e]; // Get buffered element
         vec<vlen x f32> acc = vload<vlen>(out[b,e]);
         vstore<vlen>(acc + val, out[b,e], acc); }}}}}
```

(c) Buffered code.

```
void eb(idxs: mref<? x idx>, offs: mref<? x idx>,
        vals: mref<? x ? x f32>, out: mref<? x ? x f32>){
 //Access: Iterate over segments in a batch
 slc.for(str s_b from 0 to num_batches){
   str s_beg = slc.mem_str(offs[s_b]);
   str s_end = slc.mem_str(offs[s_b+1]);
   //Access: Iterate over embeddings in a segment
   slc.for(str s_ptr from s_beg to s_end){
     str s_idx = slc.mem_str(idxs[s_ptr]);
     str<vec<vlen x f32>> buf = slcv.buf_str();
     //Access: Iterate over embedding vector elements
     slcv.for<vlen>((str s_e, str msk) from 0 to emb_len)(idx i=0){
       str s_val = slcv.mem_str<vlen>(vals[s_idx,s_e], msk);
       slc.enq(buf, s_val); }
     //Execute: Reduce embedding vectors
     slcv.callback{
       vec<vlen x f32> buf_vec = slc.to_val(buf);
       for(idx e = 0; e < emb_len; e++){
         vec<vlen x f32> val = buf_vec[e];
         vec<vlen x f32> acc = vload<vlen>(out[i,e]);
         vstore<vlen>(acc + val, out[i,e], acc); }}}
   slc.callback{ i++; }}} // end of indices in segment loop
```

(d) Queue aligned code.

Fig. 12: Progressive optimization of the EB SLC IR. Gray signifies unchanged code during progressive optimization.

temporary cast operation for source materialization. Then, during callback vectorization, Ember recursively vectorizes the uses of these cast operations, producing full SLCV code.

We decided to implement our custom vectorization pass on SLC code rather than vectorizing in SCF before lowering because scalar SCF code is simpler to decouple than vector SCF code that contains complex instructions, like mask manipulation. Moreover, it allows for advanced optimizations. For instance, an access unit like the TMU can deal with out-of-bound loads in two ways, where the benefit depends on context. The first option, similarly, sends to the CPU a mask along with the vector operand to indicate the valid vector elements. Although the mask can be directly fed into Arm SVE instructions [39], mask overheads might impact performance of tight loops. As a second option, the TMU can send zero-padded vector operands, which is the best choice when embedding vectors are multiples of the architectural vector length.

B. Embedding Buffering

The vectorized code in Figure 11b, besides (de)serializing vector operands, also needs to (de)serialize their positions within the embedding vector (variable `e`, line 16 in Figure 11b). However, since the size of the embedding vector is fixed, we can generate these positions in the compute unit to reduce queue utilization.

Ember implements this optimization by (de)serializing embedding vectors as compound types. As shown in Figure 11c, the access unit enqueues into the control queue one e_e (embedding-vector end) token for each embedding vector and, in the data queue, the position of the output embedding vector and all of its values. As the length of the embedding vector (`emb_len`) is constant, once the core reads an e_e token, it dequeues `emb_len` elements with a vectorized for-loop. This optimization greatly improves marshaling and compute efficiency.

The SLC IR greatly simplifies this optimization. With the SLC IR, Ember identifies optimization opportunities by looking into iteration callbacks for stream-to-value operations whose input is a SCL(V) loop with constant bounds. Then, as shown in Figure 12c, Ember initializes a *buffer* stream of vector type (line 10) before the inner SLCV for-loop, where such loop can enqueue the loaded embedding elements (line 14). Finally, Ember (1) moves the inner-loop body callback (lines 14–19 in Figure 12b) right after the loop (lines 16–22 in Figure 12c), (2) adds a stream-to-value operation for the buffer (line 18 in Figure 12c), and (3) adds a loop to iterate through it (lines 19–22 in Figure 12c).

Implementing the same optimization in the DLC IR would require complex analyses to reconstruct the dependency chain across the queue that Ember needs to search for optimization opportunities. Implementing the same optimization in the SCF IR is complex since the code is not in DAE form, and Ember would need to integrate such optimization in the decoupling logic, substantially complicating the compiler design. Moreover, since the SCF IR cannot represent DAE code, it would need to lower directly to the DLC IR.

C. Queue Alignment

Vector loads are more efficient when aligned to cache lines [41], [42]. However, as shown in Figure 11c, the access unit enqueues in the data queue both scalar and vector operands. Hence, the vector loads that the access unit performs from the queue are misaligned by scalar operands. Ember implements two optimizations to align queue accesses according to the code characteristics.

For simpler functions like the EB in Figure 12c, where all segment IDs are just loop induction variables, Ember stores a reference of these indices in the core and increments them after the loop. This increment is triggered by an end token of the embedding segment, s_e in Figure 11d.

As shown in Figure 12d, Ember implements such optimizations by looking into iteration callbacks for stream-to-value operations that only read the induction variable of their own SLC(V) loop (e.g. line 17 in Figure 12c). Then Ember adds a new variable in the SLC loop (`i` variable in line 12 in Figure 12d), replaces all uses of the stream-to-value operations with that variable (line 20-21 in Figure 12d), and increments it in the end callback of its child loop (line 22 in Figure 12d).

However, for more complex models like MP, certain scalars like rescaling values cannot be simplified. In this case, Ember preserves alignment by padding scalars to vectors while generating the DLC IR. As a further optimization, instead of sending segment IDs, Ember offloads to the access unit full index calculation of partial and output results and directly sends addresses to the core, reducing pressure on the core's ALUs.

Similar to embedding buffering, implementing queue alignment in SCF is not possible and implementing it in the DLC IR requires complex analyses to reconstruct the data flow of, control flow, and queue (de)serialization of the input program. Queue alignment requires control flow information to understand where to add the increment callback. It also requires queue (de)serialization to understand which alignment strategy to select.

D. Model-Specific Optimizations and HW–SW Co-Design

While the optimizations we presented so far are for general embedding operations, we also extended Ember to automatically implement some model-specific optimizations. This demonstrates the generality of the SLC IR and our compilation framework, and how Ember lends itself to HW–SW co-design.

For instance, block-sparse attention mechanisms exhibit (1) large structured reuse within each block, (2) low reuse throughout blocks, and (3) no computation. Therefore, we extended the SLC IR, DLC IR, and TMU ISA with *store* streams to write directly into memory without passing through the core, and we extended load streams to (1) select what cache level to read from and (2) whether to issue (non-)temporal requests. As demonstrated in Section VIII-B, this greatly improves core and cache utilization. Since the SLC IR preserves code structure, Ember can automatically (1) identify callbacks with no computation, (2) move the store operations into the

TABLE II: Evaluated code and reference. Ember's input is torch-mlir [34], output is TMU-CPU machine code.

Name	MLIR dialects	Description
emb-opt0	slc, scf	unoptimized Ember DAE code
emb-opt1	slcv, scf, vector	emb-opt0+vectorization
emb-opt2	slcv, scf, vector	emb-opt1+bufferization
emb-opt3	slcv, scf, vector	emb-opt2+queue alignment
ref-dae	tmu, scf, vector	hand-optimized TMU-CPU code

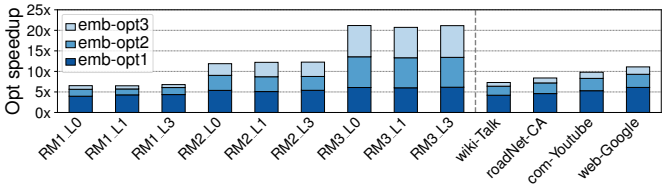


Fig. 13: Performance speedup of Ember optimizations on MP models and EB function for various DLRM models and input locality. All optimizations combined (emb-opt3) improve performance by 6.6×–21×.

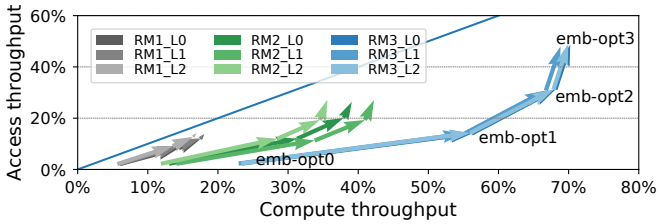


Fig. 14: Impact of Ember optimizations on access (TMU) and compute (CPU) throughput. By optimizing both, Ember achieves highest performance (top-right).

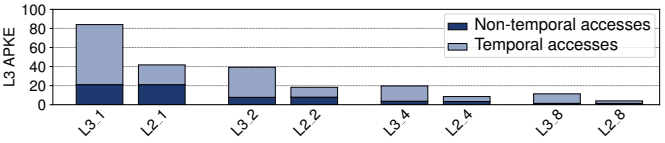


Fig. 15: L3 Accesses Per Kilo Element (APKE) of the BigBird gather function with different block sizes (1, 2, 4, and 8) and TMU configurations. Temporal accesses load indexes and non-temporal accesses load embedding vectors. Reading from L2 substantially helps filtering LLC accesses.

access unit, and (3) simplify the callback. Ember performs additional optimizations, which we introduce in Appendix D.

VIII. Evaluation

This section demonstrates how Ember enables full DAE potential at scale. Our evaluations first demonstrate the benefit of the SLC IR by performing an ablation study on the general embedding optimizations (vectorization, bufferization, and queue alignment) introduced in Section VII. Then, we show the generality of the SLC IR by showing the impact of the model-specific optimizations for LLMs introduced in Section VII-D. Finally, we demonstrate that Ember optimizations match the performance of handwritten code specifically optimized for our target TMU–CPU system. To the best of our knowledge, this is the first paper to explore DAE optimizations for embedding operations. As such, we compare Ember to handwritten SCF+VEC MLIR code that we manually decoupled and iteratively optimized through performance analyses with gem5. The experimental setting in this section follows the DAE study in Section III. Table II summarizes the code utilized in the experiments. This evaluation focuses on the TMU as it is the only DAE solution that supports both vectorization and multiple callbacks, but we believe similar considerations apply to the other DAE designs the DLC IR can abstract [8], [9], [10] (see Appendix B).

A. Impact of General Optimizations

Figure 13 shows the performance impact of general optimizations such as vectorization (emb-opt1), bufferization (emb-opt2), and queue alignment (emb-opt3) over unoptimized Ember-generated code (emb-opt0) for the EB function and more compute-intensive MP models.

For the EB function, we evaluated the three DLRMs in Table IV in Appendix A, each running representative synthetic inputs [5] with low (L0), medium (L1), and high (L2) locality. Overall, all optimizations combined (emb-opt3) achieve 6.6×, 12.1×, and 21× better performance over unoptimized code (emb-opt0) for RM1, RM2, and RM3, respectively. Vectorization is consistently the most impactful optimization with a 5.13× speedup and only 17% deviation, whereas other optimizations deliver widely different performance improvements.

To better understand these results, Figure 14 shows how these optimizations impact the throughput at which the compute unit reads and the access unit writes into the L2 queue. Compute optimizations move upward whereas memory optimizations move rightward. The blue line indicates where the compute-unit throughput equals the access-unit throughput. Only optimizing compute code cannot move above the blue line as the access unit would not be able to marshal enough data to process. This would only improve throughput up to 8× (RM3, emb-opt0) before the access unit starts to be the bottleneck (blue line). However, because of the SLC IR, Ember can perform global optimizations on both access and compute code and move both rightward and upward in the plot, improving performance by up to 21× (RM3, emb-opt3). For RM1, the most control-intensive model (shorter loops), vectorization already saturates throughput, leaving other optimizations little room for improvement. For RM2 and RM3, by reducing coordinate overhead, bufferization helps to move closer to the blue line. For RM3, the model with largest loops, queue alignment removes index overhead, pushing performance closer to the limit.

As shown in Figure 13, MP models have similar trends, and the optimization impact depends on their compute intensity.

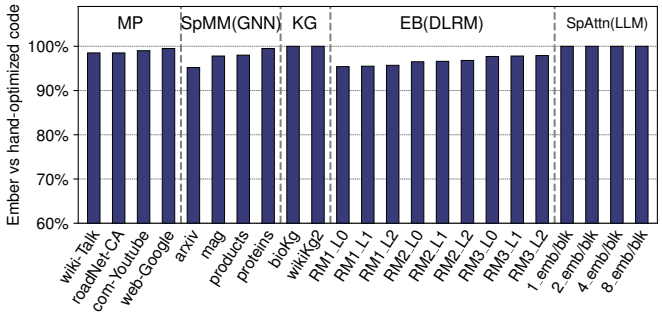


Fig. 16: DAE code automatically generated and optimized by Ember (`emb-opt3`) achieves a geomean 99% performance compared to hand-optimized code (`ref-dae`) across the different model classes in Table I.

B. Impact of Model-Specific Optimizations

As discussed in Section VII-D, Ember lends itself to model-specific optimizations. Figure 15 shows that, in block-sparse attention mechanisms (Section II-A2), loading highly-reused embedding blocks from L2 rather than LLC filters 67%–74% of the embedding reads and 50%–65% of the overall accesses, which include non-temporal loads for indexes, with larger reductions on larger block sizes with more intrinsic reuse. By directly writing data with the store streams instead of going through the core, Ember enables efficient gather operations with low resource utilization.

C. Comparison with Hand-Optimized Code

Figure 16 shows the performance of DAE code automatically generated and optimized by Ember (`emb-opt3`) compared to hand-optimized code (`ref-dae`) for all model classes in Table I. Besides all optimizations discussed in Section VII, hand-optimized code also includes low-level, CPU-specific optimizations to improve callback invocations. These CPU-specific optimizations include, for instance, (1) reordering the if-cases of multi-callback code (e.g. Figure 11d) according to their taken frequency or (2) setting the values of control tokens to be directly used in compute code (e.g. to increment variables).

Overall, these CPU-specific optimizations primarily affect multi-callback code such as MP, EB, and SpMM, yielding performance improvements of up to 5%, with an average geometric mean improvement of 1%. The limited impact of these low-level optimizations arises because, as discussed in Section VIII-A, the optimizations introduced in Section VII already push the architecture close to its limits, leaving little room for further gains. Nevertheless, because these low-level optimizations are highly CPU-specific, we chose not to integrate them into Ember, which is designed to provide a more general solution for a larger class of architectures.

Ultimately, we believe that the optimizations presented in Section VII are sufficient to fully unlock the potential of general DAE architectures at no programmability cost.

IX. Related Work

a) ***Hardware Support for Embedding Operations****:* While Meta MTIAs [12] execute embedding lookups on dedicated general purpose cores, Google TPUs [11] integrate SparseCores specifically designed to accelerate embedding operations with a dataflow architecture. The lack of publicly available details prevents any meaningful analysis of their performance or programmability, and their applicability beyond recommendation models. Moreover, the latest Nvidia GPUs [18] incorporate specialized access units, Tensor Memory Accelerators (TMAs). However, TMAs are specifically designed to fetch dense tensor tiles and cannot accelerate embedding lookups. Finally, several DAE architectures have been proposed to accelerate irregular accesses in several domains [7], [8], [9], [10], but they have not been applied to embedding operations.

b) ***Machine Learning Compilers****:* Machine learning compilers generate machine code from a semantic representation of a machine learning model [43], [13], [44], [45], [46], [47]. In general, like in the Meta MTIA ecosystem [12], these models are decomposed and lowered to handwritten kernels specifically optimized for the underlying architectures. While this solution generally achieves best performance, it does not scale with the large number of embedding operations required for different models, their complexity, and the large combination of input formats, algorithmic variants (e.g. different reduction operators), and algorithmic optimizations (e.g. operator fusion).

c) ***Tensor Compilers****:* Tensor compilers solve such issues by lowering machine learning models expressed as tensor operations through different IRs until code generation. For instance, through MLIR, one can lower PyTorch code through IRs like Linalg, SCF, and LLVM, performing optimizations at different abstraction levels. In general, high-level IRs (e.g. Linalg [33], index notations [26], [40], or einsums [43]) describe what to compute, but not how to compute it, which is represented in lower-level IRs. Although these tensor IRs allow for efficient algorithmic optimizations like operation fusion [43], [13], [44], [45], [46], [47], such an abstract representation is unsuitable for general architectural optimizations, including for DAE architectures. Therefore, most of the architectural optimizations in a tensor compiler are performed at lower-level IRs (e.g. structured IRs like the SCF IR [33], and then unstructured IRs like the LLVM IR [48]), which are generated after high-level algorithmic optimizations. While tensor compilers have been widely used to lower to CPUs [26], [40], [49], [50], [51], GPUs [52], [53], [54], FPGAs [54], dataflow accelerators [27], [55], [56], [57], and distributed systems [58], tensor DAE compilation is mostly unexplored. Lopoukhine et al. [59] recently demonstrated the potential of a multi-level flow to compile ISA extensions such as streaming registers. However, this technique is not directly applicable to a complex DAE system, like a TMU-CPU core, which completely offloads both indexing and load operations to the access unit.

d) ***Low-Level DAE Compilers****:* Prior work on DAE compilation [60], which predates tensor compilers, investigates

generating DAE code from the LLVM IR [48]. However, since the LLVM IR is unstructured, it represents loops with conditional branches. This substantially limits code analyses complex loop hierarchies like embedding operations [61], [59]. For instance, in such low-level representation, it would be unpractical to identify loop offloading candidates (Section VI-B). High-level synthesis tools overcome such challenges through pragmas [62], [63] which, however, require user intervention and cannot be used in an automated flow like Ember. For affine loops, instead, Jimborean et al. [64] overcome such challenges with polyhedral analysis which, however, cannot be applied to embedding operations. Moreover, conversely from the SLC IR, the LLVM IR is not designed to represent DAE code, and cannot define separate access and compute code nor data (de)serialization. Hence, as discussed in Section V, VI-A, and VII, this does not allow for iterative optimizations, and would require to integrate optimizations in the lowering and/or code generation algorithm, which is unpractical for complex optimizations like the ones in Section VII.

X. Conclusions

In conclusion, we demonstrated that DAE architectures outperform CPUs and GPUs in a large set of irregular embedding operations. Then, we designed the Ember compiler to integrate DAE architectures in common machine-learning frameworks. Compared to other DAE compilers, Ember progressively lowers embedding operations through custom intermediate representations to optimize code at different abstraction levels. In this way, Ember implements all necessary optimizations, both local and global, to match the performance of hand-optimized code, enabling efficient embedding operations at scale.

While the optimizations in Section VII are specifically designed for general embedding operations, Ember can generate DAE code for all sparse-dense tensor algebra expressions. Hence, Ember could implement even more specific optimizations to target, for instance, hypersparse lookups, or it could implement different vectorization schemes to target, for instance, SpMV or tall-skinny SpMM operations widely used in scientific computing [28], [65]. Hence, we believe Ember can have a large impact in other domains too.

Data-Availability Statement

The associated artifact is available on Zenodo [66].

Acknowledgment

This work is supported by the project PID2023-146511NB-I00 funded by the Spanish Ministry of Science, Innovation and Universities MCIU /AEI /10.13039/501100011033 and EU ERDF; by the Ministry for Digital Transformation and Public Service through the Càtedra Chip UPC project [grant number TSI-069100-2023-0015] and within the framework of the Recovery, Transformation and Resilience Plan - NextGenerationEU [REGAGE22e00058408992]; by the Generalitat of Catalunya through contract [2021-SGR-00763]; by DARPA under the Machine learning and Optimization-guided Compilers for Heterogeneous Architectures (MOCHA) program [HR00112520038]; by the PRISM center, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; by the Stanford Portal Center; and by the Arm-BSC Center of Excellence. M. Siracusa has been partially supported through an FI fellowship [2022FI_B 00969]. O. Hsu has been partially supported through a National Science Foundation Graduate Research Fellowship (GRFP). A. Armejach is a Serra Hunter Fellow. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

References

[1] J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, A. Moschitti, B. Pang, and W. Daelemans, Eds. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. [Online]. Available: https://aclanthology.org/D14-1162

[2] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C.-J. Wu, M. Hempstead, and X. Zhang, "Recnmp: accelerating personalized recommendation with near-memory processing," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020, p. 790–803. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00070

[3] M. Zaheer, G. Guruganesh, K. A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, and A. Ahmed, "Big bird: Transformers for longer sequences," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 17 283–17 297. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/c8512d142a2d849725f31a9a7a361ab9-Paper.pdf

[4] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: datasets for machine learning on graphs," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20. Red Hook, NY, USA: Curran Associates Inc., 2020.

[5] U. Gupta, C. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, "The architectural implications of facebook's dnn-based personalized recommendation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, feb 2020, pp. 488–501. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HPCA47549.2020.00047

[6] J. E. Smith, "Decoupled access/execute computer architectures," in *Proceedings of the 9th Annual Symposium on Computer Architecture*, ser. ISCA '82. Washington, DC, USA: IEEE Computer Society Press, 1982, p. 112–119.

[7] M. Siracusa, V. Soria-Pardos, F. Sgherzi, J. Randall, D. J. Joseph, M. Moretó Planas, and A. Armejach, "A tensor marshaling unit for sparse tensor algebra on general-purpose processors," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1332–1346. [Online]. Available: https://doi.org/10.1145/3613424.3614284

[8] Y. Yang, J. S. Emer, and D. Sanchez, "Spzip: Architectural support for effective data compression in irregular applications," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 1069–1082.

[9] M. Orenes-Vera, A. Manocha, J. Balkind, F. Gao, J. L. Aragón, D. Wentzlaff, and M. Martonosi, "Tiny but mighty: designing and realizing scalable latency tolerance for manycore socs," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 817–830. [Online]. Available: https://doi-org.stanford.idm.oclc.org/10.1145/3470496.3527400

[10] A. Barredo, A. Armejach, J. Beard, and M. Moreto, "Planar: a programmable accelerator for near-memory data rearrangement," in *Proceedings of the 35th ACM International Conference on Supercomputing*, ser. ICS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 164–176. [Online]. Available: https://doi.org/10.1145/3447818.3460368

[11] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. A. Patterson, "Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589350

[12] A. Firoozshahian, J. Coburn, R. Levenstein, R. Nattoji, A. Kamath, O. Wu, G. Grewal, H. Aepala, B. Jakka, B. Dreyer, A. Hutchin, U. Diril, K. Nair, E. K. Aredestani, M. Schatz, Y. Hao, R. Komuravelli, K. Ho, S. Abu Asal, J. Shajrawi, K. Quinn, N. Sreedhara, P. Kansal, W. Wei, D. Jayaraman, L. Cheng, P. Chopda, E. Wang, A. Bikumandla, A. Karthik Sengottuvel, K. Thottempudi, A. Narasimha, B. Dodds, C. Gao, J. Zhang, M. Al-Sanabani, A. Zehtabioskuie, J. Fix, H. Yu, R. Li, K. Gondkar, J. Montgomery, M. Tsai, S. Dwarakapuram, S. Desai, N. Avidan, P. Ramani, K. Narayanan, A. Mathews, S. Gopal, M. Naumov, V. Rao, K. Noru, H. Reddy, P. Venkatapuram, and A. Bjorlin, "Mtia: First generation silicon targeting meta's recommendation systems," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589348

[13] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *PyTorch: an imperative style, high-performance deep learning library*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[14] M. A. Ibrahim, O. Kayiran, and S. Aga, "Efficient cache utilization via model-aware data placement for recommendation models," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '21. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3488423.3519317

[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[16] M. K. Rahman, M. H. Sujon, and A. Azad, "Fusedmm: A unified sddmm-spmm kernel for graph embedding and graph neural networks," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 256–266.

[17] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web: A new form of web content that is meaningful to computers will unleash a revolution of new possibilities," *Scientific American*, May 2001.

[18] Nvidia, "Nvidia h100 specs," 2024. [Online]. Available: https://www.nvidia.com/en-us/data-center/h100/

[19] NVIDIA Corporation, "Nvidia hopper architecture in-depth," https://www.nvidia.com/en-us/data-center/technologies/hopper-architecture/, 2022, whitepaper.

[20] M. Pellauer, J. Clemons, V. Balaji, N. Crago, A. Jaleel, D. Lee, M. O'Connor, A. Parashar, S. Treichler, P.-A. Tsai, S. W. Keckler, and J. S. Emer, "Symphony: Orchestrating sparse and dense tensors with hierarchical heterogeneous processing," *ACM Trans. Comput. Syst.*, vol. 41, no. 1–4, Dec. 2023. [Online]. Available: https://doi.org/10.1145/3630007

[21] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, "Deep learning recommendation model for personalization and recommendation systems," *CoRR*, vol. abs/1906.00091, 2019. [Online]. Available: https://arxiv.org/abs/1906.00091

[22] F. Sgherzi, M. Siracusa, I. Fernandez, A. Armejach, and M. Moretó, "Spchar: Characterizing the sparse puzzle via decision trees," *Journal of Parallel and Distributed Computing*, vol. 192, p. 104941, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731524001059

[23] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and E. F. Zulian, "The gem5 Simulator: Version 20.0+," 2020.

[24] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 469–480.

[25] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, 1994. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611971538

[26] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, oct 2017. [Online]. Available: https://doi-org.stanford.idm.oclc.org/10.1145/3133901

[27] O. Hsu, M. Strange, R. Sharma, J. Won, K. Olukotun, J. S. Emer, M. A. Horowitz, and F. Kjølstad, "The sparse abstract machine," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 710–726. [Online]. Available: https://doi-org.stanford.idm.oclc.org/10.1145/3582016.3582051

[28] T. Nguyen, C. MacLean, M. Siracusa, D. Doerfler, N. J. Wright, and S. Williams, "FPGA-based HPC accelerators: An evaluation on performance and energy efficiency," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 20, p. e6570, 2022. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6570

[29] A. Parravicini, L. G. Cellamare, M. Siracusa, and M. D. Santambrogio, "Scaling up HBM efficiency of Top-K SpMV for approximate embedding similarity on FPGAs," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 799–804.

[30] F. Sgherzi, A. Parravicini, M. Siracusa, and M. D. Santambrogio, "Solving large top-k graph eigenproblems with a memory and compute-optimized fpga design," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 78–87.

[31] D. Doerfler, F. Fatollahi-Fard, C. MacLean, T. Nguyen, S. Williams, N. Wright, and M. Siracusa, "Experiences porting the su3_bench microbenchmark to the intel arria 10 and xilinx alveo u280 fpgas," in *Proceedings of the 9th International Workshop on OpenCL*, ser. IWOCL '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3456669.3456671

[32] M. Siracusa, E. Del Sozzo, M. Rabozzi, L. Di Tucci, S. Williams, D. Sciuto, and M. D. Santambrogio, "A comprehensive methodology to optimize fpga designs via the roofline model," *IEEE Transactions on Computers*, vol. 71, no. 8, pp. 1903–1915, 2022.

[33] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: scaling compiler infrastructure for domain specific computation," in *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '21. IEEE Press, 2021, p. 2–14. [Online]. Available: https://doi-org.stanford.idm.oclc.org/10.1109/CGO51591.2021.9370308

[34] LLVM, "Torch-MLIR." [Online]. Available: https://github.com/llvm/torch-mlir

[35] Google, "Mlir sparsifier," 2021. [Online]. Available: https://developers.google.com/mlir-sparsifier

[36] F. Kjolstad, W. Ahrens, S. Kamil, and S. Amarasinghe, "Tensor algebra compilation with workspaces," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. IEEE Press, 2019, p. 180–192.

[37] G. Zhang, O. Hsu, and F. Kjolstad, "Compilation of modular and general sparse workspaces," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, jun 2024. [Online]. Available: https://doi-org.stanford.idm.oclc.org/10.1145/3656426

[38] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[39] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The arm scalable vector extension," *IEEE Micro*, vol. 37, no. 02, pp. 26–39, mar 2017.

[40] A. Bik, P. Koanantakool, T. Shpeisman, N. Vasilache, B. Zheng, and F. Kjolstad, "Compiler support for sparse tensor computations in mlir," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 4, sep 2022. [Online]. Available: https://doi-org.stanford.idm.oclc.org/10.1145/3544559

[41] A. E. Eichenberger, P. Wu, and K. O'Brien, "Vectorization for simd architectures with alignment constraints," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, ser. PLDI '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 82–93. [Online]. Available: https://doi.org/10.1145/996841.996853

[42] L. Fireman, E. Petrank, and A. Zaks, "New algorithms for simd alignment," in *Proceedings of the 16th International Conference on Compiler Construction*, ser. CC'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 1–15.

[43] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[44] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *https://arxiv.org/pdf/1909.01315*, 2020.

[45] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018. [Online]. Available: http://github.com/google/jax

[46] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[47] N. Rotem, J. Fix, S. Abdulrasool, S. Deng, R. Dzhabarov, J. Hegeman, R. Levenstein, B. Maher, N. Satish, J. Olesen, J. Park, A. Rakhov, and M. Smelyanskiy, "Glow: Graph lowering compiler techniques for neural networks," *CoRR*, vol. abs/1805.00907, 2018. [Online]. Available: http://arxiv.org/abs/1805.00907

[48] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '04. USA: IEEE Computer Society, 2004, p. 75.

[49] W. Ahrens, D. Donenfeld, F. Kjolstad, and S. Amarasinghe, "Looplets: A language for structured coiteration," in *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 41–54. [Online]. Available: https://doi-org.stanford.idm.oclc.org/10.1145/3579990.3580020

[50] M. Bansal, O. Hsu, K. Olukotun, and F. Kjolstad, "Mosaic: An interoperable compiler for tensor algebra," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, jun 2023. [Online]. Available: https://doi-org.stanford.idm.oclc.org/10.1145/3591236

[51] S. Kovach, P. Kolichala, T. Gu, and F. Kjolstad, "Indexed streams: A formal intermediate representation for fused contraction programs," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, jun 2023. [Online]. Available: https://doi-org.stanford.idm.oclc.org/10.1145/3591268

[52] R. Senanayake, C. Hong, Z. Wang, A. Wilson, S. Chou, S. Kamil, S. Amarasinghe, and F. Kjolstad, "A sparse iteration space transformation framework for sparse tensor algebra," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi-org.stanford.idm.oclc.org/10.1145/3428226

[53] Z. Ye, R. Lai, J. Shao, T. Chen, and L. Ceze, "Sparsetir: Composable abstractions for sparse compilation in deep learning," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 660–678. [Online]. Available: https://doi-org.stanford.idm.oclc.org/10.1145/3582016.3582047

[54] J. Liu, Z. Zhao, Z. Ding, B. Brock, H. Rong, and Z. Zhang, "Unisparse: An intermediate language for general sparse format customization," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, Apr. 2024. [Online]. Available: https://doi-org.stanford.idm.oclc.org/10.1145/3649816

[55] K. Koul, O. Hsu, Y. Mei, S. Gautham Ravipati, M. Strange, J. Melchert, A. Carsello, T. Kong, P.-H. Chen, H. Ke, K. Zhang, Q. Liu, G. Nyengele, Z. Xie, A. Balasingam, J. Adivarahan, R. Sharma, C. Torng, J. S. Emer, F. Kjolstad, M. Horowitz, and P. Raina, "Onyx: A 12-nm programmable accelerator for dense and sparse applications," *IEEE Journal of Solid-State Circuits*, pp. 1–13, 2025.

[56] O. Hsu, A. Rucker, T. Zhao, V. Desai, K. Olukotun, and F. Kjolstad, "Stardust: Compiling sparse tensor algebra to a reconfigurable dataflow architecture," in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 628–643. [Online]. Available: https://doi-org.stanford.idm.oclc.org/10.1145/3696443.3708918

[57] K. Koul, M. Strange, J. Melchert, A. Carsello, Y. Mei, O. Hsu, T. Kong, P.-H. Chen, H. Ke, K. Zhang, Q. Liu, G. Nyengele, A. Balasingam, J. Adivarahan, R. Sharma, Z. Xie, C. Torng, J. Emer, F. Kjolstad, M. Horowitz, and P. Raina, "Onyx: A 12nm 756 gops/w coarse-grained reconfigurable array for accelerating dense and sparse applications," in *2024 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*, 2024, pp. 1–2.

[58] R. Yadav, A. Aiken, and F. Kjolstad, "Spdistal: compiling distributed sparse tensor computations," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '22. IEEE Press, 2022.

[59] A. Lopoukhine, F. Ficarelli, C. Vasiladiotis, A. Lydike, J. Van Delm, A. Dutilleul, L. Benini, M. Verhelst, and T. Grosser, "A multi-level compiler backend for accelerated micro-kernels targeting risc-v isa extensions," in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 163–178. [Online]. Available: https://doi.org/10.1145/3696443.3708952

[60] T. J. Ham, J. L. Aragón, and M. Martonosi, "Desc: decoupled supply-compute communication management for heterogeneous architectures," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 191–203. [Online]. Available: https://doi-org.stanford.idm.oclc.org/10.1145/2830772.2830800

[61] M. Siracusa and F. Ferrandi, "Tensor optimization for high-level synthesis design flows," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 4217–4228, 2020.

[62] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[63] Z. Wang and T. Nowatzki, "Stream-based memory access specialization for general purpose processors," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 736–749.

[64] A. Jimborean, K. Koukos, V. Spiliopoulos, D. Black-Schaffer, and S. Kaxiras, "Fix the code. don't tweak the hardware: A new compiler approach to voltage-frequency scaling," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: Association for Computing Machinery, 2018, p. 262–272. [Online]. Available: https://doi.org/10.1145/2544137.2544161

[65] O. Selvitopi, B. Brock, I. Nisa, A. Tripathy, K. Yelick, and A. Buluç, "Distributed-memory parallel algorithms for sparse times tall-skinny-dense matrix multiplication," in *Proceedings of the 35th ACM International Conference on Supercomputing*, ser. ICS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 431–442. [Online]. Available: https://doi.org/10.1145/3447818.3461472

[66] M. Siracusa, O. Hsu, V. Soria-Pardos, J. Randall, A. Grasset, E. Biscondi, D. J. Joseph, R. Allen, F. Kjolstad, M. Moretó Planas, and A. Armejach, "Ember artifact," 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17636956

[67] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[68] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[69] Nvidia, "Nvidia nsight systems," 2024. [Online]. Available: https://developer.nvidia.com/nsight-systems

[70] D. Mudigere, Y. Hao, J. Huang, Z. Jia, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, L. Luo, J. A. Yang, L. Gao, D. Ivchenko, A. Basant, Y. Hu, J. Yang, E. K. Ardestani, X. Wang, R. Komuravelli, C.-H. Chu, S. Yilmaz, H. Li, J. Qian, Z. Feng, Y. Ma, J. Yang, E. Wen, H. Li, L. Yang, C. Sun, W. Zhao, D. Melts, K. Dhulipala, K. Kishore, T. Graf, A. Eisenman, K. K. Matam, A. Gangidi, G. J. Chen, M. Krishnan, A. Nayak, K. Nair, B. Muthiah, M. khorashadi, P. Bhattacharya, P. Lapukhov, M. Naumov, A. Mathews, L. Qiao, M. Smelyanskiy, B. Jia, and V. Rao, "Software-hardware co-design for fast and scalable training of deep learning recommendation models," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 993–1011. [Online]. Available: https://doi.org/10.1145/3470496.3533727

[71] E. T. Phipps and T. G. Kolda, "Software for sparse tensor decomposition on emerging computing architectures," *SIAM Journal on Scientific Computing*, vol. 41, no. 3, pp. C269–C290, 2019. [Online]. Available: https://doi.org/10.1137/18M1210691

TABLE III: Typical inputs for graph-learning models.

Model	Input	#Nodes	#Edges	Layers sizes
GNN	arxiv [4]	0.2M	1.2M	128×256×256×40
GNN	mag [4]	1.9M	21.1M	128×256×349
GNN	products [4]	2.4M	61.9M	100×256×256×47
GNN	proteins [4]	0.1M	39.6M	8×256×256×112
MP	com-Youtube [67]	1.1M	6.0M	128
MP	roadNet-CA [67]	2.0M	5.5M	128
MP	web-Google [67]	0.9M	5.1M	128
MP	wiki-Talk [67]	2.4M	5.0M	128
KG	biokg [4]	0.1M	5.1M	512
KG	wikikg2 [4]	2.5M	17.1M	512

TABLE IV: Tested DLRMs.

Property Description	RM1	RM2	RM3
Segments per batch per core	64	32	16
Elements per embedding vector	32	64	128
Lookups per segment	64	128	256

APPENDIX

A. Experimental Setting

All embedding operations we tested are high-performance multicore implementations from the literature. For DLRMs, we used state-of-the-art EB implementations like the one in PyTorch [13] and Glow [47] and tested the three configurations in Table IV, each one running inputs with low (L0), medium (L1), and high (L2) locality [5]. For LLMs, we tested the original BigBird code and setting [3] while varying the blocks sizes. For GNNs and KGs, we used the OGB [4] PyTorch implementations whereas, for MP, we adapted the code from FusedMM [16]. For these graph-learning models, we used the inputs and feature sizes are reported in Table III, and come from OGB [4] or Snap/SuiteSparse [67], [68].

For the DAE processor, we measured inference latency with gem5 [23]. For the GPU, we used Nvidia Nsys [69], and only included the kernel execution time and no host-to-device transfer. GPU power consumption is measured with the `torch.cuda.power_draw` function whereas the power consumption of the DAE processor is estimated with McPat [24].

B. Ember for Other DAE Architectures

Other authors have proposed different DAE architectures to accelerate irregular operations in various domains.

SpZip [8] integrates near-core specialized units in multicore processors to accelerate traversal and (de)compression operations in graph analytics. Similarly to the TMU [7] and other architectures for sparse tensor algebra [27], SpZip access unit can be programmed with a streaming language to traverse sparse data structures and send control tokens and data to the core, which can be programmed to process it with imperative languages. Hence, both access and execute core can be abstracted with the DLC IR.

MAPLE [9] offloads general indirect accesses to specialized units placed in the network-on-chip of a multicore processor. Access/compute units communicate through push/pull operations to/from queues, respectively. Conversely from the TMU and SpZip, MAPLE access units are general-purpose cores. Hence, both access and execute code is imperative. The authors themselves claim that such code can be generated from sparse tensor compilers like TACO [26]. Since SAM implements all hardware primitives of TACO and since the DLC IR can represent all SAM operations necessary for sparse-dense tensor operations, the DLC IR can abstract MAPLE programs.

Planar [10] accelerates scientific workloads by offloading strided and irregular memory accesses from CPUs to tiny near-memory cores. Since Planar has a programming model similar to MAPLE, we can also abstract it with the DLC IR.

C. Lowering from Other IRs

To effectively optimize DAE code, Ember needs the control flow and data flow of the input program. In structured IRs like the SCF IR in MLIR, control flow is expressed as structured loop hierarchies, whereas data flow is expressed with use-def chains in SSA form. Lower-level IRs like the LLVM IR, instead, are generally unstructured, and represent control flow with conditional branches. This substantially complicates code analysis and transformation, making unstructured IRs an unsuitable solution.

We believe that lowering from higher-level tensor IRs like MLIR Linalg is not ideal either. First, lowering from tensor IRs would not allow to leverage algorithmic optimizations like operator fusion, which are generally applied (1) at the tensor level or (2) while lowering to structured IRs. Moreover, current tensor IRs like Linalg in MLIR can only express tensor operators among two operands, but embedding operations in message-passing models (e.g. SDDMM) and knowledge graphs require more operators. Finally, tensor IRs can only represent tensor algebra operations, but embedding operations might be more complicated than that. For instance, EB-like functions might identify the boundaries of the indices of active categories in a feature through segment lengths rather than pointers. While this is not a standard sparse tensor format, it can still be represented with for-loops and the DLC IR, but requires a more flexible solution than sparse tensor formats alone.

The Sparse Abstract Machine (SAM) IR [27] defines primitives to represent sparse tensor algebra expressions as dataflow programs. However, SAM cannot express DAE code separation nor data (de)serialization. Hence, while the SAM IR can be used as another Ember frontend IR, it cannot be used for DAE code optimization. Moreover, there is currently no lowering from higher-level IRs (e.g. Linalg) to SAM, preventing automatic PyTorch compilation.

Therefore, we believe that structured IRs are the most suitable input for Ember.

D. Other HW-SW Optimizations

As mentioned in Section VII, the SLC IR is designed to be flexible enough to incorporate new optimizations. In addition to the model-specific techniques described in Section VII-D, there are several others worth highlighting.

As shown in Figure 7, the EB function encodes all indices of all segments in a single vector, while storing the begin and end pointers of each segment in a second vector. This representation closely resembles the CSR (Compressed Sparse Row) format used in sparse linear algebra. However, modern EB implementations [13] typically avoid storing explicit begin and end pointers; instead, they store only the segment lengths and compute boundaries via prefix accumulation. To support this more efficient representation, we extend both the SLC IR and DLC IR with *accumulation streams*, which track EB segment boundaries by accumulating lengths rather than relying on explicit offsets [70]. This reduces memory overhead and matches contemporary runtime practices.

Another extension targets COO-like formats, which differ fundamentally from CSR-style encodings. In these cases, callbacks cannot be naturally triggered at fiber boundaries, since segment structure is not encoded explicitly. To address this, we introduce *toggle callbacks*, a mechanism that enables computation to be triggered directly on sparse coordinate formats [71]. This enhancement broadens the applicability of the IR to workloads that rely on COO representations and ensures that compiler-driven optimizations remain effective across diverse sparse tensor formats.