JOINT OPTIMIZATION OF DEEP NEURAL NETWORKS WITH DYNAMIC PROGRAMMING AND SPATIAL COMPRESSION

SAMUEL J. KAUFMAN

A dissertation submitted in
partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2026

Reading Committee:
René Just, Chair
Rastislav Bodik
Vikram Iyer
Zachary Tatlock

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science & Engineering

University of Washington

Abstract

JOINT OPTIMIZATION OF DEEP NEURAL NETWORKS WITH DYNAMIC PROGRAMMING AND SPATIAL COMPRESSION

Samuel J. Kaufman

Chair of the Supervisory Committee:
Associate Professor René Just
Paul G. Allen School of Computer Science & Engineering

Implementing high-performance deep neural network inference requires deciding how to fuse operations, how to tile loops, where to place data and when to move it, how to pack data in buffers, and which instructions and microkernels to select, among other decisions. These choices interact; for instance, increasing a tile size might amortize the cost of packing data, or fusing operators might adjust a trade-off between recomputation and storage. Because these choices interact, finding an optimal implementation requires selecting from the Cartesian product of possible decisions, a space that is typically too large to enumerate exhaustively. Compilers and auto-schedulers manage this combinatorial explosion by exploring only a small fraction of the search space, using some combination of fixed transformation sequences, heuristic search, and computation graph partitioning where sub-graphs are optimized separately.

In contrast, this dissertation develops a novel method of exhaustively searching spaces of implementations. The approach has two parts. First, it formulates the optimization problem as one with optimal substructure by developing a compositional intermediate representation and correspond-

ing cost model. Optimization proceeds by recursively rewriting program specifications into partial programs whose leaves may be smaller specifications. Optimal implementations of those sub-specifications compose into an optimal implementation of the parent specification. Second, the approach improve the space efficiency of memoizing optimal implementations with a spatial database that maps specifications to integer coordinates and compresses adjacent, identical optima.

We evaluate this approach by developing the Morello compiler. Using Morello, we synthesize and benchmark implementations of matrix multiplication and softmax for two x86 CPUs. The generated code is competitive with, and sometimes faster than, mature baselines. We also evaluate Morello's spatial memoization database, showing that it reduces storage requirements by several orders of magnitude relative to storing each specification's optimum independently.

CONTENTS

ACKNOWLEDGMENTS

I'm indebted to my advisor Ras Bodik for his infectious optimism and many years of patient mentorship, and to my advisor René Just for his guidance and, in particular, for modeling what it means to uphold high methodological standards.

Before beginning my dissertation work, I worked first on program invariant synthesis with Grigory Fedyukovich and later on accelerator performance modeling with Mike Burrows and Mangpo Phothilimthana. All three were remarkably generous with their time and advice, for which I am deeply grateful.

I'm also thankful to my friends and labmates Julie Newcomb, Gilbert Bernstein, Kayur Patel, Cat Bohannon, Mike Palazzolo, Ardi Madadi, Sam Elliott, Denys Shabalin, as well as to everyone in the PLSE lab. They made getting a PhD so much more fun than it otherwise would have been.

Finally, I'm grateful to Jenna, who went through it all with me, offering unflagging encouragement, treats, and DCOM references.

1. INTRODUCTION

Deep neural networks (DNNs) now underpin state-of-the-art vision, speech recognition, and language modeling systems. Their task performance—measured by metrics like accuracy and perplexity—improves as the amount of computation spent on training and inference increases [30]. This observation has driven the development of very large models trained to perform substantial latent computation, and the expense of training and operating these models has changed the economics of programming DNN-based systems [37, 58]. It is now cost-effective to spend significant effort optimizing training and inference speed and efficiency. When training a model costs upwards of $10 million dollars [18], even single-digit performance improvements are easy to justify. This has motivated the development of new accelerators such as the Tensor Processing Unit (TPU) and CPU functional units like AMD's AI Matrix Core, as well as a bevy of new techniques for aggressively optimizing programs targeting these processors.

Today, much of the burden of generating high-performance tensor programs falls on compiler and operator library authors. Machine learning practitioners typically express a neural network architecture as a high-level computation graph[1] using frameworks such as PyTorch or JAX, and a tensor compiler lowers these graphs to target-specific executables. Automatically generating code that matches the performance achievable by a human expert remains difficult, so it remains common for tensor compilers to generate code that calls expert-optimized libraries of performance-critical operators such as general matrix multiplication (GEMM). Though the resulting executable benefits from these external operators' efficiency, the operators are opaque to the compiler. This prevents the compiler from performing whole-program optimizations, most notably operator fusion. On modern hardware, where compute throughput outstrips memory bandwidth, fusion is critical: it keeps data in registers or cache, avoiding expensive round-trips to memory. By treating operators as black boxes, libraries effectively block the most effective transformation available for alleviating the memory bottleneck.

[1] A *computation graph* is a functional program that composes operators applied to tensors (multi-dimensional arrays).

Recent systems have therefore begun to move more optimizations out of opaque libraries and into compiler-visible intermediate representations (IRs) such as TRITON [64]. This shift allows the compiler to make fusion and other target-specific optimization decisions that library bound-

aries hide. However, making these choices visible to the compiler does not make them easy to select. A compiler must choose from an intractable number of combinations of interacting optimizations. For example, tiling a loop nest to improve cache locality interacts with vector instruction selection, which in turn depends on memory layout [33]. A tiling decision made to optimize cache usage might prevent the use of efficient vector instructions if it results in tile sizes or layouts that are incompatible with the instruction. An ideal compiler would consider all of these optimizations jointly, but the full product of decisions forms a space that is too large to exhaustively explore for any non-trivial DNN.[2]

Both traditional compilers and auto-schedulers avoid the full combinatorial explosion by exploring smaller implementation spaces in practice. Compilers do very little explicit search at all. They instead apply fixed sequences of heuristic transformations. This process can be viewed as a mostly greedy search through the space with a heuristic defined by the transformation logic, and as a consequence, compilers suffer from the *phase ordering problem*: decisions made in early phases permanently prune optimal implementations from the space of programs under consideration. In contrast, auto-schedulers often find better implementations by conducting a large explicit search of different candidate lowerings with fewer subdivisions of the search space. Still, auto-schedulers usually implement a heuristic search that explores only a small subset of expressible programs. Although auto-schedulers can be made arbitrarily effective with enough effort, by increasingly specializing the search space or heuristics,[3] neither traditional compilation nor auto-scheduling guarantees selection of an optimal (e.g., cost-minimizing) implementation.

[2] For example, Steiner et al. [60] estimate that there are more than 10^{1239} implementations of VGG-19 expressible in HALIDE.

[3] Tollenaere et al. [65] provide a useful example: for 2D CPU convolutions with fixed vectorization, reuse, and microkernel-composition dimensions, they make random sampling effective through choices such as pre-filtering microkernels and restricting tilings to avoid awkward boundary cases.

1.1. CASE STUDY: COMPOSING MATRIX MULTIPLICATIONS

To motivate a joint search of the kind we address in this dissertation, we evaluate four implementations of a matrix product pipeline $D \leftarrow D + (BC)A$, which differ in loop fusion choice and tile shapes. Specifically, we compare:

- a two-stage baseline that computes BC and then D;
- a variant that fuses those two stages by inlining production of BC into the consumer loop, a transformation typically performed by a high-level, graph-level optimizer;
- a variant that changes the output tile shape from 4×16 to 2×32, a decision typically made by a low-level kernel optimization pass; and

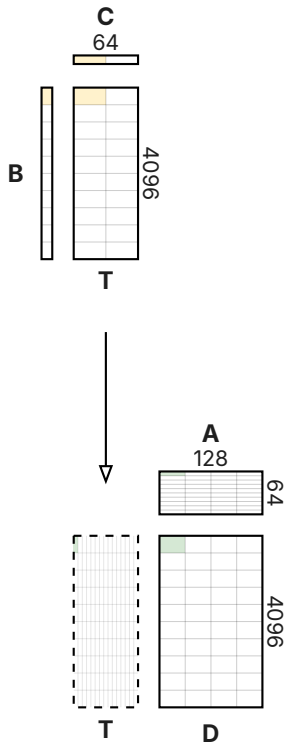


Figure 1. Two-stage matrix multiplication implementation.

- a combined variant that both fuses the stages and uses a 2×32 output tile.

In isolation, neither fusion nor output tile reshaping improves upon the performance of the baseline. In fact, both slightly degrade performance, so a local search equipped with an accurate cost model considering each optimization independently is unlikely to select either. Similarly, a lowering phase that performs either optimization without guaranteeing the other risks degrading performance. However, combining both high- and low-level optimizations improves throughput by 15%, so a search over the product of these decisions is likely to find this optimum. This type of of joint optimization can be discovered by the approach we will develop in this dissertation.

In this case study, the matrix A has shape $K \times N$, while B and C are vectors of shapes $M \times 1$ and $1 \times K$, respectively. Data are all 32-bit floats in row-major layout, and we set $M = 4096$, $K = 64$, and $N = 128$.

1.1.1. BASELINE IMPLEMENTATION

The two-stage baseline implementation follows:

```
// Producer loop: computes T = B C (outer product)
for (int i = 0; i < M; i++) {
  // vf8 is an 8-float, 256-bit AVX2 vector type, and
  // _mm256_set1_ps fills all elements with a scalar
  vf8 b_vec = _mm256_set1_ps(B[i]);
  for (int j = 0; j < K; j += 8) {
    vf8 c_vec = *(vf8 *)(C + j);
    *(vf8 *)(T + i * K + j) = b_vec * c_vec;
  }
}

// Consumer loop: computes D += T A
for (int i = 0; i < M; i += 4) {
  for (int j = 0; j < N; j += 16) {
    vf8 acc[4][2] = {0};  // 4×16 accumulator tile

    for (int k = 0; k < K; k++) {
      vf8 w[4] = {_mm256_set1_ps(*(T + (i + 0) * K + k)),
                  _mm256_set1_ps(*(T + (i + 1) * K + k)),
                  _mm256_set1_ps(*(T + (i + 2) * K + k)),
                  _mm256_set1_ps(*(T + (i + 3) * K + k))};
      vf8 a[2] = {*(vf8 *)(A + k * N + j + 0),
                  *(vf8 *)(A + k * N + j + 8)};
      for (int r = 0; r < 4; ++r) {
        for (int c = 0; c < 2; ++c)
          acc[r][c] += w[r] * a[c];
      }
    }
```

```
      for (int r = 0; r < 4; ++r) {
        float *d_row = D + (i + r) * N + j;
        for (int c = 0; c < 2; ++c)
          *(vf8 *)(d_row + c * 8) += acc[r][c];
      }
    }
  }
```

To simplify, the preceding program does not implement common GEMM[4] optimizations like matrix packing. However, two key design choices—separate producer and consumer stages and a 4×16 accumulator shape—are representative of the code generated by tensor compilers. These compilers will typically lower to two individual calls to a BLAS library, and BLAS libraries targeting AVX2 usually spend most of their time in a kernel with a 6×16 or 16×6 accumulator shape. (Our case study uses 4×16.)

Since B and C are vectors, T is an outer product. While B and C together occupy about 16 KiB of memory, T requires 1 MiB, which is larger than a typical core's L2 cache. As a result, the producer spills `T` out of L2 and effectively streams those writes through the slower, shared cache hierarchy. At the same time, each element of T is the result of a single multiplication, so the producer has very low arithmetic intensity. It performs 1 FLOP for every 8 bytes accessed (counting 4B read from C and 4B written to T, ignoring the factored-out `B[i]` value), and is therefore heavily memory-bound.

[4] General Matrix Multiplication (GEMM), the standard BLAS operation, computes $C \leftarrow \alpha AB + \beta C$. The scalar factors α and β typically have negligible impact on performance, so this dissertation will often compare GEMM and matrix multiplication directly.

1.1.2. OPTIMIZATION 1: FUSION

First, to avoid the cost of writing and reading the large intermediate matrix T to slow memory, we will fuse the two stages. To do this, we delete the producer loop entirely and replace the loads of `T` with inline scalar multiplications, as follows:

```
vf8 w[4] = {_mm256_set1_ps(B[i + 0] * C[k]),
            _mm256_set1_ps(B[i + 1] * C[k]),
            _mm256_set1_ps(B[i + 2] * C[k]),
            _mm256_set1_ps(B[i + 3] * C[k])};
```

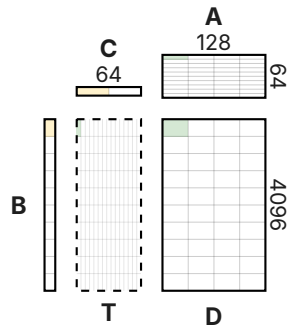


Figure 2. Fused matrix multiplication implementation.

On a reference Zen 1 machine, this new implementation is about the same speed as (or slightly worse than) the two-stage baseline implementation. It avoids the write to and later reread of the 1 MiB T buffer, but it replaces those memory operations with extra rematerialization work in the inner loop. This increases the program's total number of multiplication operations, because each value of T is computed $\frac{N}{16} = 8$ times.

More importantly, fusion lengthens the critical path through the microkernel. Instead of loading a scalar from `T` using execution pipes independent of those needed for fused multiply–adds (FMAs), the fused kernel computes `B[i+r] * C[k]` using the same FPU pipes. Since these scalar multiplications compete with the vector FMAs for execution slots, they cannot be pipelined; this resource contention lengthens the critical path and correspondingly reduces throughput. There is relatively little memory latency for the core to overlap with this extra dependency chain because the other operands are small enough to mostly remain resident in L1 cache (and the only truly streaming traffic is writing `D`). As a result, in a near-peak-throughput kernel, these extra multiplications are less likely to be hidden by pipelining and reduce throughput.

The net effect is a shift from a memory-heavy producer stage to a compute-heavy fused stage, and the performance gain from eliminating memory traffic is negated by the cost of this additional computation and pipeline contention. (Note that FlashAttention relies on a similar optimization. It avoids materializing the QK^T matrix, which is usually too large to store efficiently [19].)

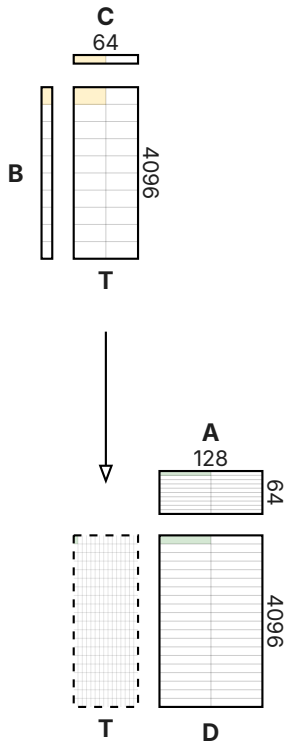


Figure 3. Two-stage matrix multiplication implementation with a shorter, wider output tile.

1.1.3. OPTIMIZATION 2: RESIZE OUTPUT TILE TO 2X32

Just as the fused implementation rematerializes each element of T multiple times, so does the baseline implementation read each value of T multiple times in its consumer stage. Instead of fusing the two stages, let's evaluate changing the consumer stage's accumulator tile shape from 4×16 to 2×32. This change halves the number of times T is re-read from the cache by doubling the number of accumulated output values computed per each element loaded from T. Since T spills from L2 cache while A fits comfortably in L1, an optimizer might select this transformation to trade expensive loads of T for cheap L1 loads of A, theoretically improving performance.

To implement this, we keep the producer stage unchanged and modify the consumer loop nest: change `acc[4][2]` to `acc[2][4]`, change `w[4]` to `w[2]`, update the `i` and `j` loop strides from 4 to 2 and 16 to 32 respectively, reshape the writeback loop to accommodate the new `acc` shape, and load a larger tile of A:

```
vf8 a[4] = {*(vf8 *)(A + k * N + j + 0),
            *(vf8 *)(A + k * N + j + 8),
            *(vf8 *)(A + k * N + j + 16),
            *(vf8 *)(A + k * N + j + 24)};
```

This optimization also slightly degrades performance by increasing the number of cache misses when reading `A`. The root cause is that, while the 4×16 kernel loads one cache line of `A` per line of `T`, the 2×32 kernel loads two cache lines of `A` per line of `T`. At the same time, the wider tile shortens `A`'s reuse distance: with fewer `j`-tiles per `i`-iteration, less data moves between consecutive accesses to the same `A` line. This improves the miss rate per access, but not enough to compensate for doubling the access count. The result is more absolute cache misses.

1.1.4. COMBINING OPTIMIZATIONS 1 & 2

However, if we apply both optimizations—fuse the stages and use the 2×32 consumer tile—the resulting program's throughput improves by about 15%. The two transformations compensate for each other's weaknesses. First, the wider tile shortens the critical path through the fused kernel. Recall that fusion introduced extra scalar multiplications that compete with vector FMAs for CPU resources; this lengthened the critical path and reduced throughput. The 2×32 tile performs twice as many FMAs per scalar broadcast, so the extra multiplications become a smaller fraction of the work. Second, fusion eliminates T, removing the L1 contention that caused the 2×32 tile to miss more when applied to the unfused baseline. Without fusion, both A and T compete for L1 capacity; with fusion, A has the cache to itself.

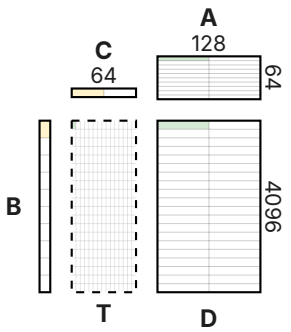


Figure 4. Fused matrix multiplication implementation with a shorter, wider output tile.

Of course, the preceding analysis is specific to a particular problem size, data type and layout, and target machine. The preceding implementations elide important optimizations needed to approach peak throughput such as data packing, loop reordering (e.g., interchanging loops to $i \to k \to j$), cache tiling, explicit loop unrolling, and maximizing vector register utilization. Furthermore, the efficacy of the optimizations presented here depends heavily on specific problem and system parameters, including matrix dimensions, the system memory hierarchy characteristics, and the amount of contention for memory bandwidth induced by parallelism.

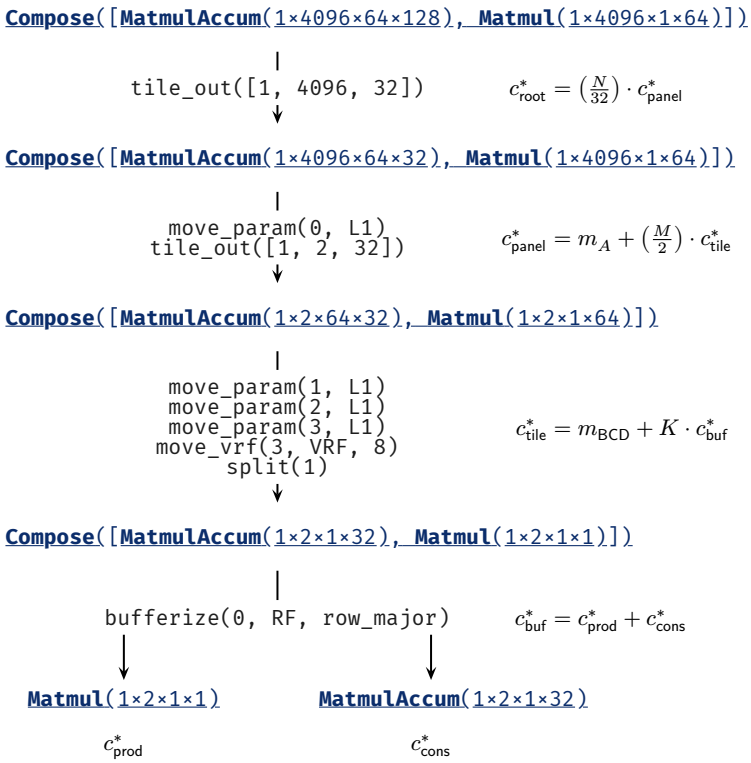
Nevertheless, this case study shows that, even for a workload as well-studied as matrix multiplication, finding an optimal lowering is rarely a matter of applying a single clever transformation. This interaction between the two optimizations demonstrates a central challenge for both compilers and auto-schedulers: transformations that look harmful in isolation can be beneficial in combination because one transformation changes the bottleneck that determines the other's payoff. A local search that ranks fusion and tiling choices independently—or a lowering pipeline that commits to either optimization without guaranteeing the other—can

therefore get stuck in a locally optimal configuration even when a different combination is globally better. However, a joint search over the product space of these decisions would select the fused-and-2×32 variant.

1.1.5. SYNTHESIZING THE COMBINED OPTIMIZATION

The compiler we develop in this dissertation discovers just these kinds of joint optimizations. It represents decisions about tile shapes and fusion —and, more importantly, the hardware cache behavior implied by those decisions—in a single IR. As a result, our compiler can define a cost model for that IR, weighing the joint impact of these decisions.

With a cost model defined, the compiler searches possible lowerings of a program specification by recursively building programs, top-down, via a terminating rewrite system. Costs are computed bottom-up when rewriting terminates. For example, the following rewrite tree summarizes construction of the combined implementation in the preceding Section 1.1.4:



The tree first fixes the tiling and data-movement decisions. Moves into L1 model cache-miss costs rather than corresponding to instructions in the final code. We write m_A for the cost of the A move into L1 and m_{BCD} for

the combined cost of moving B, C, and D into L1 and staging the output D in the vector register file (VRF). The final `bufferize` splits the composition into sequential matrix multiplications, costing c^*_{prod} and c^*_{cons}, which communicate through the register file (RF). Together, these determine the joint cost of the final program:

$$c^*_{\mathsf{root}} = \left(\frac{N}{32}\right) \cdot \left(m_A + \left(\frac{M}{2}\right) \cdot \left(m_{\mathsf{BCD}} + K \cdot \left(c^*_{\mathsf{prod}} + c^*_{\mathsf{cons}}\right)\right)\right).$$

The factor $\frac{N}{32} = 4$ results from applying `tile_out([1, 4096, 32])` to the full composition. The consumer's N dimension is independent of the producer's dimensions, so each 32-column tile executes the producer separately.

In contrast, the unfused implementation with a 2×32 consumer tile (described in Section 1.1.3) begins with `bufferize(0, GL, row_major)`, which materializes T and immediately splits the composition into independent producer and consumer matrix multiplications. Because this split precedes tiling, there is no top-level $\frac{N}{32}$ factor multiplying the costs of both matrix multiplications. Instead, each matrix multiplication independently accrues its own data-movement costs, including the producer's write and the consumer's reload of T. Taken together, these costs exceed the fused implementation's movement costs by more than the saved producer work, so the unfused implementation has the larger total cost. (The implementation in Section 1.1.2 also has a higher cost.)

1.2. SCALABLE SYNTHESIS WITH DYNAMIC PROGRAMMING

The thesis of this dissertation is that *there exists a compositional IR and cost model with both optimal substructure and rank accuracy that enables selecting globally cost-optimal DNN implementations from joint search spaces of interacting optimizations previously considered intractable.*

We validate this thesis with an existence proof: we introduce a dynamic programming approach to selecting empirically fast implementations from larger spaces than was previously possible. Our approach leverages two key innovations. First, we developed an IR and associated cost model that have an optimal substructure (a minimal-cost program is composed of minimal-cost sub-programs). We recursively decompose program specifications into smaller specifications reachable via a set of rewrites—for example, tiling rewrites of a matrix multiplication into loops of smaller matrix multiplications—and then compose a final program by choosing rewrites that minimize a cost model. Second, to reduce space requirements, we represent the memoization table by mapping specifications to

coordinates in $\mathbb{N}^n$ (where $\mathbb{N} = \{0, 1, 2, ...\}$) and then merge adjacent, matching solutions into hyperrectangles that cover the set of coordinates. We index this table with an R*-Tree.

To evaluate our approach, we implemented Morello, a synthesizing compiler that lowers specifications to C targeting x86 using our compositional IR. It is available at: https://github.com/samkaufman/morello.

We target CPUs because they remain important execution engines for tensor programs, particularly when workloads are small, irregular, latency-sensitive, or fused with host-side computation. Recent CPUs also provide wide SIMD units, low-precision arithmetic, and CPU-integrated matrix extensions, giving them important similarities to accelerators: high-throughput arithmetic units, hierarchical memories, and performance that depends on arranging computation to reuse data and match the target's vector or matrix operations.

We evaluated Morello along two dimensions: (1) the memory footprint of our memoization database representation and (2) the throughput of synthesized matrix-multiplication and softmax implementations. We found:

- Our database representation *reduces the storage requirements of our approach by about four orders of magnitude* over naively storing each solution (e.g., in a hash map indexed by individual specifications).
- With modest manual intervention, Morello generates matrix-multiplication kernels for AMD Zen 5 and Zen 1 CPUs that *approach OpenBLAS performance and often outperform AOCL and Intel MKL*.
- Morello generates softmax implementations much faster than those of PyTorch and are competitive with XNNPACK on Zen 1 and, in many cases, Zen 5.

Taken together, these results suggest that exhaustive search is viable for implementing fast operators and few-operator modules.

1.3. DISSERTATION OUTLINE

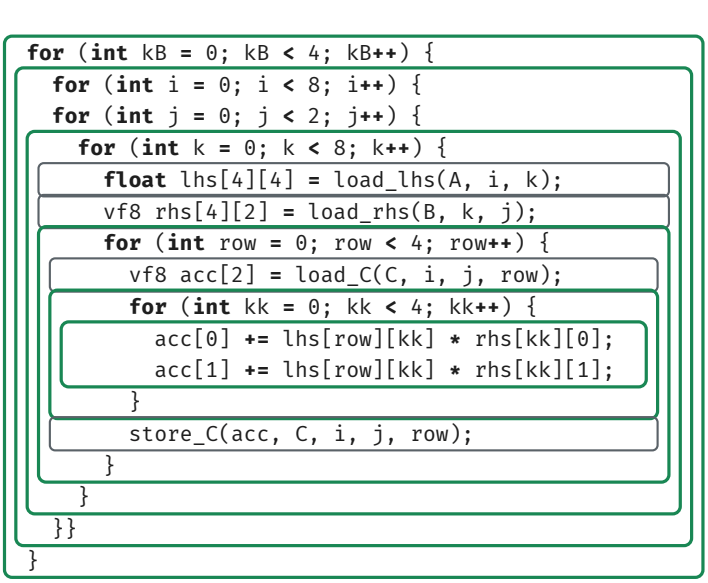
The remainder of this dissertation consists of the following four chapters.

Chapter 2 presents the Morello IR and its scheduling language. It develops the IR and compositional cost model with an optimal substructure and demonstrates that the former is expressive and the latter accurate enough to generate fast x86 matrix multiplication and softmax implementations, as described in Chapter 4.

Chapter 3 presents our dynamic-programming-based approach to synthesizing optimal neural network inference code over this IR. It presents both top-down (Section 3.2) and bottom-up (Section 3.4) synthesis algorithms, as well as analyses of their complexity (Section 3.3).

Chapter 5 presents the novel memoization database design that enables our synthesizer to scale. It discusses the design trade-offs motivating our method of storing memoized solutions as rectangles and evaluates the compression ratio of this representation, showing several orders of magnitude improvement over naive memoization.

Finally, Chapter 6 compares our approach to related work from the program synthesis and database literature.



```
for (int kB = 0; kB < 4; kB++) {
  for (int i = 0; i < 8; i++) {
  for (int j = 0; j < 2; j++) {
    for (int k = 0; k < 8; k++) {
      float lhs[4][4] = load_lhs(A, i, k);
      vf8 rhs[4][2] = load_rhs(B, k, j);
      for (int row = 0; row < 4; row++) {
        vf8 acc[2] = load_C(C, i, j, row);
        for (int kk = 0; kk < 4; kk++) {
          acc[0] += lhs[row][kk] * rhs[kk][0];
          acc[1] += lhs[row][kk] * rhs[kk][1];
        }
        store_C(acc, C, i, j, row);
      }
    }
  }}
}
```

implements	sizes	memories
MatmulAccum	(32,128,32)	(GL,GL,L1)
MatmulAccum	(32,32,32)	(L1,L1,L1)
MatmulAccum	(4,32,16)	(L1,L1,L1)
Move	(4,4)	(L1,RF)
Move	(4,16)	(L1,VRF)
MatmulAccum	(4,4,16)	(RF,VRF,L1)
Move	(1,16)	(L1,VRF)
MatmulAccum	(1,4,16)	(RF,VRF,VRF)
MatmulAccum	(1,1,16)	(RF,VRF,VRF)
Move	(1,16)	(VRF,L1)

Figure 5. A simple matrix multiplication implementation. For each `MatmulAccum`, the three displayed sizes are (M, K, N), and the memory labels RF and VRF denote the scalar and vector register files, respectively. The implementation is hierarchical: each colored box corresponds to a sub-program that implements either data movement (gray) or a matrix multiplication (green).

A high-performance DNN implementation is usually organized as a hierarchy of nested sub-programs mapped to the target's memory hierarchy. For example, GEMM is usually implemented as one or more loops containing 'smaller' GEMMs, where a smaller GEMM is applied to smaller tiles that fit in faster memories, as in Figure 5 [26, 71]. This hierarchical pattern appears in both individual operators like GEMM and in compositions of those operators into DNN pipelines. For instance, DeepSpeed-Inference [2] tiles an entire layer normalization block to keep the loop body's intermediate tiles resident in on-chip memory.

Morello's IR models this hierarchical structure. It is a language of tensors (multi-dimensional arrays), loops, memory-movement instructions, and, at the leaves, microkernel applications. The IR's key novelty is that a program's leaves may be specification applications, which denote sub-programs that still lack an implementation. Programs are *completed* by repeatedly rewriting specification applications into nested sub-programs that may themselves contain specifications and recursing until no specifications remain. We refer to this process as *scheduling* a *partial* program. A

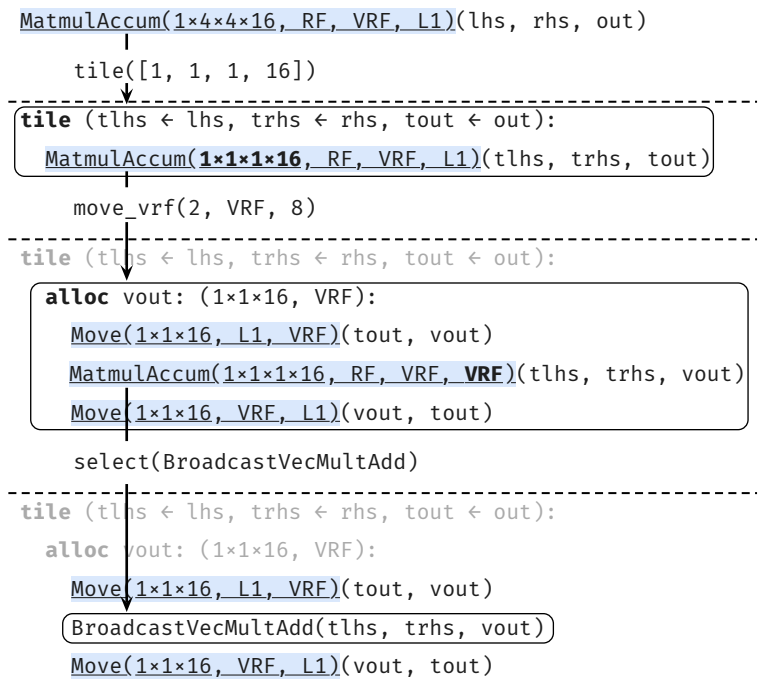


Figure 6. A partial lowering of a matrix multiplication specification with dimensions $(B, M, K, N) = (1, 4, 4, 16)$. The specification is tiled to $(1, 1, 1, 16)$, moves the output parameter into vector registers (VRF), and replaces `MatmulAccum` with `BroadcastVecMultAdd`. The three operands of the post-move `MatmulAccum` reside in RF, VRF, and VRF, respectively. The `Moves` into and out of the vector-register tensor `vout` remain unimplemented.

practical benefit of this IR is that it is extremely straightforward to lower to C code with syntax-directed translation. (Contrast with, for example, a polyhedral representation.)

2.1. A BRIEF TOUR OF PROGRAMMING WITH MORELLO

We now elaborate on the IR by scheduling the multiplication of a 4×4 matrix by a 4×16 matrix, as summarized in Figure 6. In particular, we implement an *accumulating* matrix multiplication, which adds its results to a given output tensor rather than replacing it.

We begin with a goal specification application: `MatmulAccum(1×4×4×16, RF, VRF, L1)(lhs, rhs, out)`.[5] A specification always carries a concrete shape, here ordered B (batch), M, K, N, which determines the shapes of its parameters. Critically, a specification also assigns each parameter a resident memory: the first operand in the scalar register file (RF), the second in the vector register file (VRF), and the output in the L1 cache. The specification can be applied only to tensor arguments whose shapes and

[5] Specifications are shown in blue and underlined.

memories match; in this case, those arguments are the externally provided `lhs`, `rhs`, and `out` tensors.

Tiling. We apply the tiling rewrite `tile([1, 1, 1, 16])` to this specification application, producing a loop over the output tile and its input dependencies:

```
tile (tlhs: (1×1×1, RF) ← lhs, trhs: (1×1×16, VRF) ← rhs,
      tout: (1×1×16, L1) ← out):
  MatmulAccum(1×1×1×16, RF, VRF, L1)(tlhs, trhs, tout)
```

The loop iterates over all perfect tiles of the specification's shape. At each iteration, the newly introduced symbolic tensors `tlhs`, `trhs`, and `tout` are shifted to be concrete, data-dependency-respecting views of their source tensors (e.g., `lhs` for `tlhs`) according to the correspondence between the specification coordinate space and the coordinate spaces of its parameters. The loop body contains a smaller specification applied to these views.

Had we chosen a tile size that did not evenly divide the specification shape, lowering would produce a loop with a main body for the interior tiles and one or more epilogues specialized to the boundary tiles. For example, applying `tile([1, 1, 1, 5])` to the same specification would yield:

```
tile (tlhs: (1×1×1, RF) ← lhs, trhs: (1×1×5, VRF) ← rhs,
      tout: (1×1×5, L1) ← out):
  // main body
  MatmulAccum(1×1×1×5, RF, VRF, L1)(tlhs, trhs, tout)
  // boundary, executed once
  MatmulAccum(1×1×1×1, RF, VRF, L1)(tlhs, trhs.tail, tout.tail)
```

The full-tile main body covers the three `1×1×1×5` tiles, while the epilogue is specialized to the final `1×1×1×1` boundary tile.

Data Movement. Next, we rewrite the smaller, nested `MatmulAccum` application by applying the `move_vrf(2, VRF, 16)` rewrite. This introduces a move of the output tile (parameter 2) from the L1 cache into the vector register file. The argument `16` selects 512-wide (16-value) register names (`zmm` on AVX-512).

The result is an `alloc`-binding.[6] The name `vout` is bound to a new tensor, and the binding's body contains an application of the redex specification whose tensor specification has been modified to reside in VRF. The body also contains two `Move` specifications: a load from L1 into VRF and a store from VRF to L1.

[6] These are let-bindings that carry a tensor cost derived from their destination tensor, $c_t(p_{\text{dest}})$, as discussed in Section 3.1.

```
tile (tlhs: (1×1×1, RF) ← lhs, trhs: (1×1×16, VRF) ← rhs,
      tout: (1×1×16, L1) ← out):
  alloc vout: (1×1×16, L1):
    // load from L1 into vector registers
    Move(1×1×16, L1, VRF)(tout, vout)
    MatmulAccum(1×1×1×16, RF, VRF, VRF)(tlhs, trhs, vout)
    // store from vector registers into L1
    Move(1×1×16, VRF, L1)(vout, tout)
```

In general, a move rewrite's result includes a load when the data movement is software-controlled—specifically, when the destination memory is software-managed or when the `Move` changes layout or data type. Stores are inserted under the same conditions, provided the moved parameter is an output. Load and store moves are present in the preceding example because the vector register file is software-managed and the parameter is an output. Section 2.4 discusses our modeling of memory, including hardware caches and tensor layouts (e.g., `row_major`).

Microkernel Selection. Finally, we apply `select(BroadcastVecMultAdd)` and `select(VectorMove)` to substitute `BroadcastVecMultAdd` for the `MatmulAccum` specification and `VectorMove` for each `Move` specification. This results in the following program:

```
tile (tlhs: (1×1×1, RF) ← lhs, trhs: (1×1×16, VRF) ← rhs,
      tout: (1×1×16, L1) ← out):
  alloc vout: (1×1×16, VRF):
    VectorMove(tout, vout)
    BroadcastVecMultAdd(tlhs, trhs, vout)
    VectorMove(vout, tout)
```

At this point, the implementation is complete: no specifications remain, and all leaves are microkernel applications. During code generation (Section 2.9), a microkernel application lowers to an arbitrary block of C code. In Morello, these are typically small blocks of straightline code. In the preceding example, `BroadcastVecMultAdd` lowers to a pair of **`vbroadcastss`** and **`vfmadd132ps`** instructions, and `VectorMove` lowers to **`vmovups`** instructions.

The rewrites we have seen—tiling, data movement, and microkernel selection—are sufficient to express a range of important optimizations: strip-mining and register blocking are tiling, vectorization is selecting a vectorized microkernel, layout packing is a `move` between layouts, and loop fusion is tiling a composed specification rather than one of its components.[7] Table I and Table III list the supported rewrites.

[7] Morello does not control loop unrolling. LLVM's unrolling heuristics have been sufficient in our experiments. Morello unrolls only to reference unique names of registers in the target code.

Values in the Morello IR (e.g., `tlhs` and `vout`) are *tensors*, which are multi-dimensional array views into memory that span one or more buffers or registers. A tensor's specification abstracts over both tiles and "ordinary" tensors that fill a buffer, so any program that can be applied to a non-tile tensor can be applied to a tile with the same specification.

2.2. SPEC LANGUAGE

So far, we presented a simplified specification language. However, the Morello specification language is more expressive. We call an expression in this language a Spec, and Specs can have five fields: (1) a type, (2) a shape, (3) a list of tensor specifications, (4) a serial-only flag, and (5) peak storage limits per memory (registers or cache lines). A *tensor specification* describes a parameter and has three required fields: (1) data type, (2) memory, and (3) data layout. For parameters resident in vector registers, it additionally carries a vector size. The serial-only flag, when set, guarantees that an implementation uses at most one thread.

We can incorporate these additional fields into the goal specification from Figure 6 to produce the following example of a complete Morello Spec:

```
(MatmulAccum(
  1×4×128×16,
  (bf16, L1, row_major),
  (bf16, L1, (col_major, c0)),
  (f32, VRF, row_major, v16),
  serial),
[16, 32, 32768, 1073741824])
```

In this syntax:

- `c0` indicates that no tensor values are guaranteed to be adjacent in memory. More generally, `cn`, where `n` is a non-negative integer, counts the number of innermost physical dimensions guaranteed to be contiguous in memory. Section 2.4.2 elaborates on tensor contiguousness.
- `v16` means that each 16-value strip of values shares a vector register (i.e., data are placed in `zmm` registers on AVX-512). These vector size selections are specified for parameters in vector registers (`VRF` for x86 targets).[8]
- `[16, 32, 32768, 1073741824]` limits the peak memory that a satisfying implementation can allocate in each memory of the hierarchy. These memories are ordered: scalar register file, vector register file, L1 cache, then "global" main memory. Registers are

[8] An alternative design could model these widths as aliased but separate memories, mirroring vector register indexing.

Table I. All Morello-supported rewrites except softmax Spec rewrites, which are listed in Table III. Each rewrite transforms a Spec $\mathcal{S}$ into an implementation node containing one or more sub-Specs.

Rewrite	Transform
Tiling	
`tile_out(shp)`	`𝒮(i.., o)` ⇝ `tile (i' <- i .., o' <- o): 𝒮'(i'..,o')` Tile the output to shape `shp`. The loop may be parallel.
`split(k)`	`𝒮(i,j,..)` ⇝ `tile (i' <- i, j' <- j, ..): 𝒮'(i',j',..)` Tile the outermost primitive Spec reduction dimension to `k`.
Move	
`move_param(idx, mem)`	`𝒮(a)` ⇝ `alloc a': Move(a, a'); 𝒮(a'); Move(a', a)` Move parameter at index `idx` to memory `mem`.
`move_relayout(idx, mem, layout)`	Like `move_param`, also changing the data layout to `layout`.
`cast(idx, dtype, mem, layout)`	Like `move_relayout`, also widening the data type to `dtype`.
Other	
`select(kernel)`	`𝒮(a..)` ⇝ `kernel(a..)` Substitute a microkernel application.
`to_accum()`	`𝒮(a.., out)` ⇝ `Fill(out); 𝒮`$_{\text{accum}}$`(a.., out)` Replace $\mathcal{S}$ with a `Fill` and its accumulating variant $\mathcal{S}_{\text{accum}}$.
`bufferize(idx, mem, layout)`	`Compose(P.., Q..)(a.., b..)` ⇝ `pipeline (t): (P..)(a.., t); (Q..)(t, b..)` Split the composition, breaking the `idx + 1` outermost primitive Specs `P..` into their own sub-Specs. The resulting sub-Specs `P..` and `Q..` communicate via a new intermediate tensor `t` in the given memory with the given layout.
`spatial_split()`	`ConvAccum(img, f, out)` ⇝ `tile (img' <- img, f' <- f): MatmulAccum(img', f', out)` Convert a convolution whose filters span the full image into a loop of `MatmulAccum`s over spatial dimensions (like im2col [9]).
`broadcast_first(mem, layout)`	`DivideVecScalar(numer, s)` ⇝ `pipeline (b): Broadcast(s, b); DivideVec(numer, b)` Rewrite a `DivideVecScalar` to first broadcast the divisor into a new tensor with the given memory and layout, then update the numerator in place.

counted for register files. Cache lines are counted for other memories; as in the preceding example, these are displayed and written as bytes (lines multiplied by line size) as a convenience for sharing specifications between targets with differing cache line sizes. See Section 2.4.1 for detail.

A Spec's *type* is one of 23 primitives, such as `Move`, `Matmul`, `Conv`, or `Softmax`, or is a composition of these primitives. Fourteen of those primitives are paired in that they have both an accumulating and non-accumulating variant; for example, `MatmulAccum` adds its results to the output, while `Matmul` replaces the output. For each supported pair, Morello can implement the non-accumulating variant as an output-initializing sub-Spec followed by its accumulating variant. See Table II for the full list of supported primitive Spec types.

The shape of a Spec, combined with its type, defines the shapes of its parameters. For example, a `Matmul` has a $B \times M \times K \times N$ shape. We call the number of dimensions in a Spec or tensor shape its *rank*; this `Matmul` Spec has rank 4. From this, we can derive the shapes of its parameters:

$$\begin{aligned} \text{LHS} &\quad B \times M \times K \\ \text{RHS} &\quad B \times K \times N \\ \text{Output} &\quad B \times M \times N \end{aligned}$$

Parameter dimensions need not be equal to a Spec dimension. For example, a `Conv` Spec has the shape $B \times F \times C \times H_o \times W_o \times H_f \times W_f$. From this, we derive the following parameter shapes:

$$\begin{aligned} \text{Image} &\quad B \times C \times (H_o + H_f - 1) \times (W_o + W_f - 1) \\ \text{Filters} &\quad F \times C \times H_f \times W_f \\ \text{Output} &\quad B \times F \times H_o \times W_o \end{aligned}$$

Maintaining per-Spec shapes leads to more concise Spec notation and is a simple way of respecting data dependencies during tiling rewrites.

2.3. PIPELINES

Neural network pipelines are expressed by composing specifications. For example, the following is an application of a composition of a producer 1×64×1024×32 matrix multiplication with a consumer 1×64×32×128 matrix multiplication:

```
(Compose(
  [Matmul(1×64×32×128), Matmul(1×64×1024×32)],
```

Table II. Spec types implemented in Morello. The "Acc." column indicates whether an accumulating variant of the primitive is supported (e.g., `Matmul` and `MatmulAccum`). The "Params." column indicates the number of parameters (inputs and outputs).

Spec	Acc.	Params.	Description
`OnePrefix`		2	Reshape $d_0 \times ... \times d_n$ to $1 \times d_0 \times ... \times d_n$.
`Fill`		1	Fill a tensor with a constant value.
`Move`		2	Copy data, potentially changing layout or type.
`Matmul`	✓	3	Compute matrix multiplication.
`Conv`	✓	3	Compute convolution.
`Broadcast`		2	Broadcast a tensor along a dimension.
`DivideVec`		2	Divide elementwise, updating the numerator in place.
`DivideVecScalar`		2	Divide by a broadcasted scalar, updating the numerator in place.
`Max`	✓	2	Compute a maximum reduction.
`Softmax`		2	Compute softmax.
`SoftmaxComplete`		4	Compute softmax from precomputed denominator and max.
`SoftmaxDenominatorAndMax`	✓	3	Compute the max and denominator (sum of exponentials).
`SoftmaxDenominatorAndMaxFromParts`	✓	4	Combine per-tile maxima and denominators into the maximum and denominator over the full reduction.
`SoftmaxDenominatorAndUnscaled`		3	Compute the denominator and unscaled scores.
`SoftmaxDenominatorAndUnscaledFromMax`	✓	4	Compute the denominator and unscaled scores from max.
`SoftmaxDenominator`	✓	3	Compute the softmax denominator.

```
    [(f32, GL, row_major), (f32, GL, row_major),
     (f32, GL, row_major), (f32, GL, row_major)],
    serial),
  [16, 32, 32768, 1073741824])(a, b, c, out)
```

A composed specification binds its component primitives from outermost to innermost, with each primitive's output passed as the first input to the next, while the remaining inputs are read left-to-right. The rightmost parameter is the final output parameter.

Like primitive specifications, compositions can be tiled and have their parameters moved. For example, we can tile and move the output tensor of the preceding specification to produce:

```
tile (ta: (1×32×32, GL) <- a, tb: (1×32×1024, GL) <- b,
      tout: (1×32×32, GL) <- out):
  alloc resident: (1×32×32, L1) <- tout:
    (Compose(
      [Matmul(1×32×32×32), Matmul(1×32×1024×32)],
      [(f32, GL, row_major), (f32, GL, row_major),
       (f32, GL, row_major),
       (f32, L1, row_major)],
      serial),
    [16, 32, 16384, 1073741824])(ta, tb, c, resident)
```

The result is a loop in which no decision has yet been made about where the intermediate data (the result of the producing `Matmul`) will be materialized. Tiling a composed specification resembles a traditional compiler's loop fusion transformation: both transformations choose a common iteration space for operations that would otherwise be computed in separate loop nests. This enables reuse of producer results by their consumers within each tile and can avoid materializing a full-sized intermediate, reducing memory traffic.

Ultimately, compositions are lowered to their separated primitives via `bufferize` rewrites. These rewrites "split" a composition into producer and consumer parts, which respectively write into and read from a new, intermediate tensor. For example, applying `bufferize(0, L1, row_major, None)` to the nested composition produces:

```
tile (ta: (1×32×32, GL) <- a, tb: (1×32×1024, GL) <- b,
      tout: (1×32×32, GL) <- out):
  alloc resident: (1×32×32, L1) <- tout:
    pipeline (intermed: (1×32×32, L1)):
      (Matmul(1×32×1024×32,
        (f32, GL, row_major), (f32, GL, row_major),
        (f32, L1, row_major),
```

```
    serial),
  [16, 32, 8192, 1073741824])(tb, c, intermed)
  (Matmul(1×32×32×32,
    (f32, L1, row_major), (f32, GL, row_major),
    (f32, L1, row_major),
    serial),
  [16, 32, 8192, 1073741824])(intermed, ta, resident)
```

This new program contains a *pipeline*, which is a node for sequential execution.

Unlike plain blocks, pipelines imply interlocking lifetimes for intermediate tensors: intermediates live only during the execution of their producing or consuming sub-program. For example, a longer pipeline may contain overlapping, but non-identical, lifetimes:

```
pipeline (x, y):
  Matmul(..)(..., a)
  Matmul(..)(a, ..., b)
  Matmul(..)(b, ...)
```

Here, `a` is live during the first two sub-programs and `b` is live during the last two. Neither intermediate is live for the entire pipeline.

2.4. DATA MOVEMENT

For most deep learning workloads, reaching peak performance requires careful scheduling of data layouts and movement. Tiles should be staged into caches and registers to improve throughput and arithmetic intensity, trading off limited capacity between buffers to avoid unexpected spills or cache misses. Frequently, peak performance demands reordering data to support sequential access and direct indexing by SIMD instructions.

Morello supports these optimizations by tracking peak memory limits and tensor data layouts for each Spec, as well as by explicitly scheduling data movement operations (including layout changes) with `Move` Specs. Memory limits ensure that neither caches nor registers are oversubscribed. Layouts constrain Spec-microkernel compatibility. For example, `Move` Specs can be implemented by a `vmov` instruction only when both arguments have the same layout and are contiguous. Layouts also map tensors to the number of backing cache lines for accurate cost modeling even if there are repeated or partial accesses to a line. No simulation of the memory system is needed.

2.4.1. MEMORY LIMITS

`alloc`-binding and `pipeline` nodes define clear lifetimes for introduced tensors, so Morello tracks per-memory consumption by decreasing the corresponding memory limits of nested Spec applications. For register files, Morello subtracts the number of backing registers. For other memories, the number of cache lines is subtracted and snapped to the lower power of two or zero.[9]

In the common case, memory limits are tuples of integers tracking available capacity at each memory level (e.g., `[16, 32, 8192, 1073741824]`). However, when a `pipeline` is substituted for a `Compose` application, the nested `Compose` Specs' memory limits carry additional information to support flattening nested `pipeline`s. These extended memory-limit values are called `pipemem`s. For example, interposing a buffer between primitive Specs `b` and `c` of `Compose([a, b, c], M)` results in an implementation like:

```
pipeline (buffer_c):
  c(..., M - buffer_c.mem)(..., buffer_c)
  Compose([a, b, c], pipemem(M, first=M-buffer_c.mem, last=M)) \
    (buffer_c, ...)
```

The primitive nested Spec `c` has a tuple-only memory limit. There is no rewrite from a primitive Spec to a `pipeline`, so attaching tuple-only memory limits to `c` is precise. Further splitting the nested `pipemem`-carrying `Compose` produces:

```
pipeline (buffer_c):
  c(..., M - buffer_c.mem)(..., buffer_c)
  pipeline (buffer_b):
    b(..., M - buffer_c.mem - buffer_b.mem) \
      (buffer_c, ..., buffer_b)
    a(..., M - buffer_b.mem)(buffer_b, ...)
```

Notice that the preceding program subtracts `buffer_c`'s memory consumption from the memory limits of `c` and `b` but not of `a`, which maps correctly to `pipeline`'s introduced tensors' lifetimes.[10]

Fragmentation. This memory accounting is optimistic for GL-resident heap allocations: it models an ideal packing of live buffers, ignoring fragmentation. Placement gaps can cause a heap workspace to be larger than the sum of its live buffers, in which case Morello underestimates the required memory capacity. An ideal memory model would remain compositional while performing precise—or at least conservative—static planning, so that the model accounts for gaps. Such planning is difficult in

[9] Only rewrites producing `alloc`-bindings or `pipeline`s lower memory limits. We do not lower the memory limits of tiled Specs to account for loop iterators. We make the assumption that LLVM will, via spilling to the L1 cache, hide any performance impact from live loop iterators. This is simpler and avoids artificially limiting the loop depth of generated programs. In practice, high-performance x86 implementations are constrained more by the number of vector registers than scalar registers, so this assumption has little impact.

[10] Nested `pipeline`s are flattened before code generation and pretty-printing.

general, but simple for programs that allocate only through `alloc`-bindings and microkernels. This is discussed further in Section 2.9.1.

2.4.2. LAYOUTS

To model rewrite applicability and memory-system performance, each tensor specification includes a *layout*. A layout is a *buffer indexing expression* paired with a *contiguousness value*. A former is a function that maps a tensor shape to a bijection between tensor coordinates and buffer offsets; the latter is the number of contiguous physical dimensions.

During code generation, tensors are realized by recursively composing such indexing expressions. A tensor backed directly by a buffer is accessed using the layout-derived expression for the named backing buffer, while a tile composes that expression with variables introduced by the enclosing tiling loops to refer to the corresponding region of its backing tensor. For tensors in vector register files, Morello first interprets the resulting offset over the tensor's flattened element space, then splits it during code generation into a vector-register name and an in-register lane.

Layouts are constructed with the `layout!` macro, which lists *physical dimensions* by their type, size, and associated logical dimension. As an example, a row-major layout for a 3-dimensional tensor can be constructed with `layout![0, 1, 2]`, which is equivalent to the `row_major` constructor we saw previously. Alternately, packing 16-value-wide strips of the middle logical dimension can be done with `layout![0, 1, 2, 1 p(16)]`.

Morello supports layouts that map logical tensor dimensions to regularly nested physical dimensions (e.g., column-major: $mj + i$), optionally with a strip dimension of size s (e.g., $m\left\lfloor \frac{j}{s} \right\rfloor + si + (j \bmod s)$). This includes common and important layouts like row-major and NCHW, as well as packed layouts such as Goto-style matrix packing [26] or NCHWc [25]. Additionally, strips may be odd-even (Faro) shuffled. For example: $m\left\lfloor \frac{j}{s} \right\rfloor + si + \sigma(s, j \bmod s)$, where

$$\sigma(s, x) = 2\left(x \bmod \left\lfloor \frac{s}{2} \right\rfloor\right) + \left\lfloor \frac{2x}{s} \right\rfloor.$$

This enables lane shuffling between vector registers when the vector size is $\frac{s}{2}$. To express these layouts, Morello supports affine buffer indexing expressions extended with floor division and mod.

A layout's contiguousness value models whether a tensor's values are adjacent in memory. This field affects the costs of data movements and applicability of certain microkernels. Morello models contiguousness as the number of inner physical dimensions for which values are adjacent to each other in the underlying buffer. A tensor is said to be *fully contiguous*

if that number is equal to the number of physical dimensions in the layout. Tracking the length of this suffix, rather than only whether the tensor is fully contiguous, lets Morello determine when further tiling removes the remaining non-contiguous outer dimensions and thereby produces a fully contiguous tile.

Cache Set Associativity. Cache-resident tensors have a different approximation: Morello counts cache lines but does not model cache set associativity. Two buffers with the same total cache-line footprint can behave differently if their addresses map to the same cache sets. This approximation is usually acceptable because optimal programs typically pack the working set into a linear access order, which is friendly to set-associative caches. The model is therefore most likely to be wrong for pathological access patterns, such as power-of-two strides that repeatedly conflict in the same cache sets.

2.4.3. MOVE SPECS

While `alloc`-bindings account for the cost and capacity effects of hardware-managed caches, explicit copies, relayouts, and data type widenings must be scheduled explicitly. In Morello, these transformations are expressed as `Move` Specs, which denote value-preserving transfers from one tensor to another, potentially changing memory, layout, and/or data type.

Copies. When the source and destination tensor specifications differ only in their assigned memories, the Spec denotes a direct copy between them, such as `Move(4×4, (u8, L1, row_major), (u8, RF, row_major))`, which loads a tile from L1 into registers.

Layout Changes. When only layouts differ, the Spec defines a relayout operation into a separate buffer within the same memory. For example, `Move(4×4, (u8, L1, row_major), (u8, L1, col_major))` copies data from a tensor with a row-major layout into one with a column-major layout.

A single `Move` can change both memory and layout. For example, moving a tile from global memory into L1 while reorganizing its data into a packed layout models Goto-style matrix packing [26]. This is represented as:

```
Move(128×2048,
  (f32, GL, row_major),
  (f32, L1, [1, 0, 1 p(16)]))
```

A `Move` Spec is often implemented as a nest of `Move` implementations that stream data from global memory into registers, shuffle it if necessary, and store it into the packed destination. For example, the following code

implements the preceding `Move` by streaming 8-value micropanels from global memory to L1, into vector registers, and back to L1:

```
// Move(128×2048, (f32, GL), (f32, L1, [1, 0, 1 p(16)]))
tile (src_a: (128×64, GL, ([0, 1], c1))    ← src,
      dst_a: (128×64, L1, [1, 0, 1 p(16)]) ← dst)
  // Move(128×64, (f32, GL, ([0, 1], c1)),
  //              (f32, L1, [1, 0, 1 p(16)]))
  alloc src_b: (128×64, L1, ([0, 1], c1)) ← src_a
    // Move(128×64, (f32, L1, ([0, 1], c1)),
    //              (f32, L1, [1, 0, 1 p(16)]))
    tile (src_c: (1×8, L1) ← src_b,
          dst_b: (1×8, L1) ← dst_a)
      // Move(1×8, (f32, L1), (f32, L1))
      alloc intermed: (1×8, VRF, 8)
        // Move(1×8, (f32, L1), (f32, VRF, 8))
        Assign(src_c, intermed)
        // Move(1×8, (f32, VRF, 8), (f32, L1))
        Assign(intermed, dst_b)
```

Each 16-value micropanel is already contiguous in the destination layout, so the packing is realized entirely by where each vector is written, as defined by the `dst_b` tile. In this case, no in-register shuffling is needed.

Data Type Widening. Memory and layout changes can also be combined arbitrarily with data type widening in a single `Move`. This is useful, for example, when computing the float32 dot product of bfloat16 vectors on an architecture without native bfloat16 support. In that case, a software-emulated dot product widens the bfloat16 operands to float32 before multiplication, which can be expressed by a Spec like:

```
Move(1×1×2048,
  (bf16, GL, row_major),
  (f32, L1, layout![2, 2 oe(16)]))
```

In addition to widening the data type, the destination layout packs each vector-pair micropanel into odd-even order. Consequently, each consecutive pair of bfloat16 values widens into same-indexed lanes of two float32 vectors: one holding the even elements and the other the odd elements. This avoids lane-crossing shuffles during both widening and subsequent FMA.

Expressing widening and relayout as a single `Move` preserves their association during lowering, which enables substitution of a single microkernel that satisfies both requirements. For example, the `VectorInterleaveBf16F32` microkernel lowers to code such as:

```
*(__m256 *)out =
  (__m256)_mm256_slli_epi32(*(__m256i *)&src, 16);
*(__m256 *)(out + 8) =
  (__m256)_mm256_blend_epi16(_mm256_setzero_si256(),
                             *(__m256i *)&src,
                             0xAA);
```

This C code converts bfloat16 to float32 by zero-padding the lower 16 bits. It executes `_mm256_slli_epi32` to shift the even bfloat16 elements into the upper 16 bits of each 32-bit lane (automatically zeroing the lower half) and applies `_mm256_blend_epi16` with a `0xAA` mask to blend the odd elements into the upper halves of each lane while leaving the lower halves zero.

2.5. PARALLELISM

The MORELLO IR expresses thread-level parallelism using fork-join loops. Such loops are introduced by applying a `tile` rewrite with its parallel flag set.

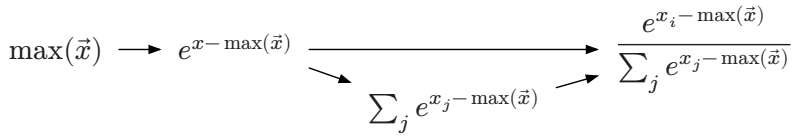
To preserve semantics, a parallel tiling rewrite must ensure that the introduced loop is free of data races. In the MORELLO IR, this check is straightforward: parameters are never aliased, programs always terminate, and exhibit static control. Most importantly, whether a SPEC is accumulating, together with the reduction dimensions implied by its indexing relations, is sufficient to determine whether loop iterations write to disjoint tiles. When the tiles are not disjoint, the tiling rewrite inserts synchronized reductions.

To simplify the cost model for multithreading, MORELLO disallows nested parallelism. Recall that rewriting is monotonic with respect to a SPEC's serial-only flag: a serial-only SPEC is never rewritten into an implementation that contains a non-serial-only sub-SPEC. Accordingly, when `tile` introduces a parallel fork-join loop, it rewrites the loop body so that all constituent SPECS are marked serial-only. Consequently, the rewrite system cannot introduce nested parallel loops.

2.6. SOFTMAX DECOMPOSITIONS

As described in Section 2.3, `Compose` defines linear producer-consumer chains with a fixed communication pattern. Softmax is an example of a specification that cannot reasonably be expressed this way. Its data dependencies form a small branching graph in which the input contributes to a reduction $m = \max_k a_k$, a reduction $d = \sum_j e^{a_j - m}$, and the final pointwise normalization $\text{out}_i = \frac{e_i}{d}$ for $e_i = e^{a_i - m}$.

Naturally, there are many ways an implementer may wish to decompose the equation, corresponding to introducing buffers for edges in the following graph:



Fortunately, while higher-level abstractions like `Compose` are useful, multi-operator programs (in effect, computation graphs) can be named arbitrarily just like any other Spec. The rewrite system only needs additional rules to decompose those programs into graph partitions. Therefore, rather than implementing a more general graph variant of `Compose`, Morello exposes a set of softmax-specific primitive Specs and the decomposing rewrites that relate them. Figure 7 illustrates the relationships among these Specs and rewrites.

The two rewrites from `Softmax` expose a trade-off between recomputing or storing data. The `to_softmax_parts` rewrite writes the full-size tensor e of unscaled exponentials directly into the output together with the reduced denominator tensor d, and then normalizes the output in place with `DivideVecScalar`; this avoids recomputing exponentials without allocating a distinct full-size intermediate, but requires an additional pass over the output. The `to_softmax_parts_recompute` rewrite instead materializes only the reduced m and d tensors and then applies `SoftmaxComplete`, which recomputes the exponentials when producing the final output. This avoids the separate normalization pass at the cost of extra arithmetic.

Softmax implementations also differ in whether they implement the Milakov–Gimelshein *online* [44] algorithm[11] or the naive, *offline* algorithm. This can be a critical optimization; when memory bandwidth is the constrained resource, the online algorithm can reduce memory traffic by doing additional arithmetic, much like the decision to recompute exponentials. `SoftmaxDenominatorAndMax` can use either algorithm, whereas `SoftmaxDenominatorAndUnscaled` follows the offline decomposition. The `split_softmax_denominator_and_max` and `parallel_split_softmax_denominator_and_max` scheduling operators rewrite into serial and parallel versions of the online algorithm respectively.

[11] Online softmax allows the fused computation of the maximum and denominator to be tiled. Per-tile summaries (m_i, d_i) can be combined by taking their maximum and rescaling each d_i to that common maximum before summation. This follows from the identity $e^{a_j-m} = e^{a_j-m_i}e^{m_i-m}$, which converts terms normalized by a tile's maximum m_i to terms normalized by the combined maximum m.

2.7. CANONICALIZATION

Upon construction, Morello canonicalizes Specs. It quotients Specs by a small set of symmetries and chooses a single representative from each

Table III. Rewrites of softmax Specs supported by Morello. (Non-softmax Spec rewrites are listed in Table I.) As in Table I, vector-size parameters are omitted from rewrite names for readability.

Rewrite	Transform

`to_softmax_parts(`
`  denom_mem, denom_layout)`
	`Softmax(a, out) ⇝`
	` pipeline (d):`
	`  SoftmaxDenominatorAndUnscaled(a, d, out)`
	`  DivideVecScalar(out, d)`

`to_softmax_parts_recompute(`
`  max_mem, max_layout,`
`  denom_mem, denom_layout)`
	`Softmax(a, out) ⇝`
	` pipeline (m, d):`
	`  SoftmaxDenominatorAndMax(a, m, d)`
	`  SoftmaxComplete(a, m, d, out)`
	Recomputes exponentials from the max.

`to_max_and_denominator()`	`SoftmaxDenominatorAndMax(a, m, d) ⇝`
	` Max(a, m)`
	` SoftmaxDenominator(a, m, d)`

`split_softmax_denominator_and_max(k)`	`SoftmaxDenominatorAndMaxAccum(a, m, d) ⇝`
	` tile (a' <- a):`
	`  SoftmaxDenominatorAndMaxAccum(a', m, d)`

`parallel_split_softmax_denominator_and_max(`
`  k, partials_mem, partials_layout)`
	`SoftmaxDenominatorAndMax(a, m, d) ⇝`
	` pipeline (p_m, p_d):`
	`  parallel tile (a' <- a, p_m' <- p_m,`
	`  p_d' <- p_d):`
	`  SoftmaxDenominatorAndMax(a', p_m', p_d')`
	`  SoftmaxDenominatorAndMaxFromParts(p_m, p_d, m,`
	`d)`

`to_max_and_unscaled(`
`  max_mem, max_layout)`
	`SoftmaxDenominatorAndUnscaled(a, d, e) ⇝`
	` pipeline (m):`
	`  Max(a, m)`
	`  SoftmaxDenominatorAndUnscaledFromMax(a, m, d, e)`

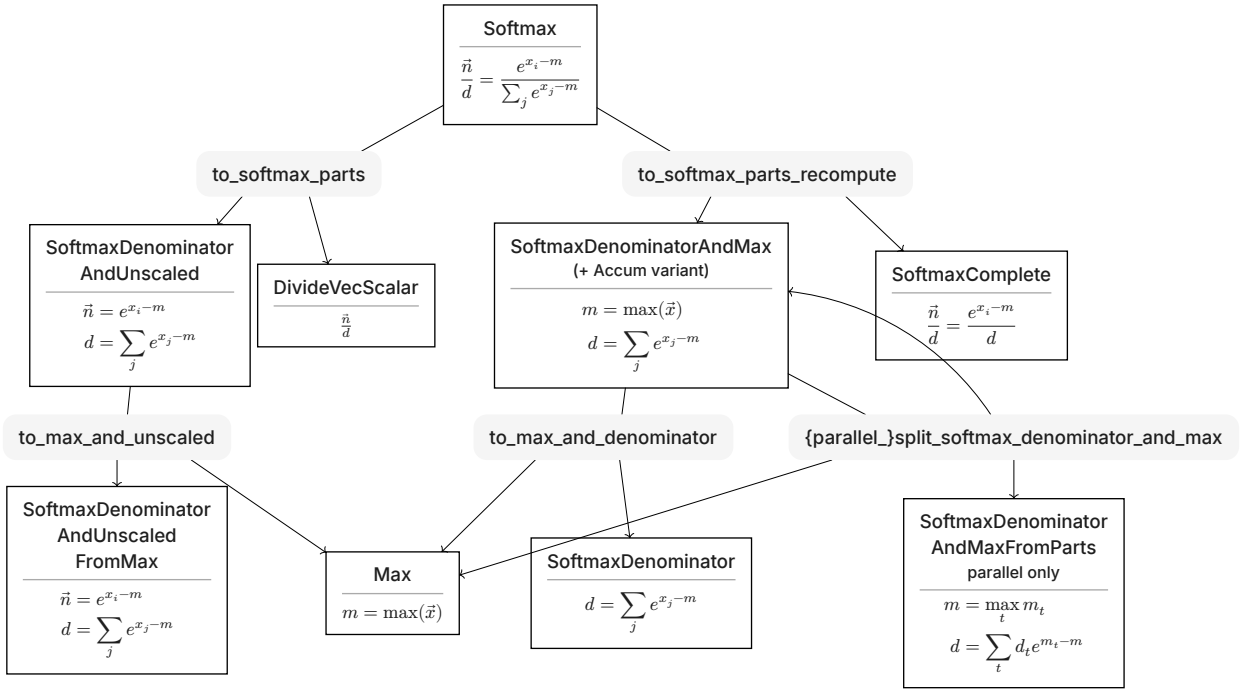


Figure 7. Graph of the relationships between the Softmax Spec decomposition rewrites. Nodes are labeled with Spec types and the (sub-)expressions they compute. Each edge is annotated with the name of a rewrite and leads from the redex Spec type to the sub-Specs produced by that rewrite.

equivalence class. Canonicalization improves the readability of Specs and simplifies rewrite rule implementation by simplifying Spec-matching applicability tests. It also slightly improves memoization table hit rates by reducing the set of key Specs, though this reduction is small and does not affect asymptotic complexity.

Most of these symmetries arise from layout equivalences. Two layouts are equivalent if they yield the same buffer indexing expression after simplifying with respect to the tensor shape terms, and `Move` Specs introduce an additional symmetry: for fully contiguous copies with fixed source and destination memories and data types, the layouts are irrelevant. For example, a tensor specification like `(2×1024, u8, L1, [0, 1, 0 p(4)])`, which has a relatively complex layout, obscures the fact that the tensor's physical layout is equivalent to the simpler column-major layout `(2×1024, u8, L1, [1, 0])`. Here, the outer, packed coordinate is constant over the tensor's shape, so simplification identifies the two layouts. The canonical form is both more readable and easier to match: a microkernel that applies when one of its arguments is a column-major matrix need not itself determine whether the packed dimension `p(4)` can be ignored.

The same idea underlies the other canonicalization rewrites. These erase layouts for tensors whose values live in separate scalar registers (e.g., `RF`), drop physical dimensions corresponding to logical tensor dimensions of size 1, merge consecutive physical dimensions for the same logical dimension when they are equivalent to a single standard physical dimension, erase outer, packed physical dimensions when the outer, packed coordinate is constant over the tensor's shape, and collapse equivalent fully contiguous copies to a single row-major representation. We now examine each of these in more detail.

Scalar-Register Memories. For a tensor in a scalar register file such as `RF`, canonicalization erases its layout to the empty layout (i.e., treating every dimension as having size 1). Morello models tensors assigned to scalar register files as sets of single-value buffers, so, once each value occupies its own register, there is no meaningful inter-value layout or buffer indexing expression to preserve. As an example:

```
(2×2×8, f32, RF, [1, 0, 2])

     |
canonicalize
     ↓

(2×2×8, f32, RF)  // no layout
```

By contrast, vector register files retain layouts because they contain multiple values each. This canonicalization rule could be generalized by defining layouts only within individual vector registers rather than across an entire VRF-resident tensor. That would preserve order within each register while discarding any higher-level ordering among distinct vector registers.

Size-1 and Consecutive Dimensions. Canonicalization drops physical layout dimensions corresponding to logical tensor dimensions of size 1. It also merges consecutive physical dimensions for the same logical dimension whenever those adjacent dimensions are equivalent to a single standard physical dimension (i.e., when a standard dimension directly encloses a packed dimension). For example, the following layout drops the size-1 dimension `1` and merges the contiguous packed dimension `0 p(4)` with its enclosing standard dimension `0` to produce a simple row-major layout:

```
(4×1×8, f32, L1, [0, 1, 0 p(4), 2])

     |
canonicalize
     ↓

(4×1×8, f32, L1)  // no layout = row-major = [0, 2]
```

Recall that a layout's contiguousness value counts the number of inner physical dimensions that are adjacent in memory, so changes to those inner dimensions may affect this value. If canonicalization removes or merges physical dimensions that are part of the layout's contiguous suffix, the contiguousness value must be decreased by the number of eliminated dimensions within that suffix. If a canonicalization step merges a dimension inside the contiguous suffix with one outside of it, the new contiguousness value cannot, in general, be recovered exactly from this summary alone. Morello therefore conservatively approximates it by excluding the merged dimension entirely. Therefore, this rewrite is not strictly an equivalence on the full layout abstraction: it preserves indexing exactly, but it may replace contiguousness with a sound lower bound.

Degenerate Packings. Canonicalization also drops packings that do not affect indexing for the current tensor shape: if the outer, packed coordinate is constant over the tensor, the explicit packed dimension can be removed. This rewrite also lowers contiguousness when the removed physical dimension had previously been counted in the contiguous suffix. For example:

```
(2×1024, u8, L1, [0, 1, 0 p(4)])  // 4 >= 2

     ↓
canonicalize
     ↓

(2×1024, u8, L1, [1, 0])
```

This does not apply to an outer odd-even dimension because such layouts can still change the mapping from logical indices to physical positions even when that outer coordinate is constant.

Moves. `Move` Specs have an extra symmetry-breaking rule: when the source and destination have the same data type and matching fully contiguous layouts, the copy is canonicalized to a row-major bitwise move for memories with layouts; any `RF` operand remains without an explicit layout because its values are in separate scalar registers. For example:

```
(Move(1×8, (f32, L1, [1, 0]),
           (f32, L1, [1, 0])), [16, 32, 32, 64])

     ↓
canonicalize
     ↓

(Move(1×8, (f32, L1), (f32, L1)), [16, 32, 32, 64])
```

This collapses equivalent contiguous copies to a single memoized Spec.

Taken together, these canonicalization rules rewrite equivalent Specs into more readable forms and simplify determination of when rewrites and microkernels apply.

2.8. USER SCHEDULING

Though the Morello IR is designed to support automatic optimization, rewrites are also exposed to programmers via a halide-style scheduling language [54] embedded in Rust.

For example, the following schedule yields the matrix multiplication implementation developed in Section 2.1:

```
let goal = spec!(MatmulAccum(1×4×4×16, RF, VRF, L1), ..);
let implementation = goal
  .tile(&[1, 1, 1, 16])
  .move_vrf(2, VRF, 16)
  .subschedule(&[0], |load_spec| load_spec.select(VectorMove))
  .subschedule(&[1], |matmul_spec| {
    matmul_spec.select(BroadcastVecMultAdd)
  })
```

```
  .subschedule(&[2], |store_spec| {
    store_spec.select(VectorMove)
  });
```

Selecting Leaves. Rewrites are applied to the leaves of partial implementations. When the program has only one nested Spec, the target of a rewrite is unambiguous. This is the case for the `tile` and `move_vrf` rewrite applications in the preceding schedule. However, when a program node has multiple children, the rewrite target is ambiguous. This is the case for the implementation resulting from `move_vrf`, which contains three sub-specifications: a `Move` loading data into `VRF`, a nested `MatmulAccum`, and a `Move` storing data from `VRF` back into `L1`. In this case, the programmer chooses the rewrite's target with the `subschedule` operator. In the preceding example, child Specs at indices 0 and 2 are replaced with the `VectorMove` microkernel, and the child Spec at index 1 is replaced with `BroadcastVecMultAdd`.

Default Child Selection. We expect that programmers will often be interested in manually scheduling programs' main arithmetic loops and leave the implementation of `Move` Specs to generic helper functions or synthesis (discussed in the next section). To support this workflow, we implement a default child selection shorthand: when an implementation has multiple children and would normally require `subschedule` to specify the target Spec, the rewrite instead applies to the sole non-Move child if there is exactly one. For example, the same schedule can be written more directly as:

```
let implementation = goal
  .tile(&[1, 1, 1, 16])
  .move_vrf(2, VRF, 16)
  .select(BroadcastVecMultAdd)  // Applies to non-Move leaf
  .subschedule(&[0], |load_spec| load_spec.select(VectorMove))
  .subschedule(&[2], |store_spec| {
    store_spec.select(VectorMove)
  });
```

2.8.1. MIXED SCHEDULING

Programmers can mix manual and automatic scheduling by applying some rewrites manually and synthesizing the remaining sub-Specs. For instance, a programmer might manually schedule the main arithmetic loop of the preceding `MatmulAccum` but have Morello synthesize the `Move` Specs:

```
let db = todo!("Allocate a memoization database");
let goal = spec!(MatmulAccum(1×4×4×16, RF, VRF, L1), ..);
let implementation = goal
  .tile(&[1, 1, 1, 16])
  .move_vrf(2, VRF, 16)
  .select(BroadcastVecMultAdd)
  .synthesize_all(&db, None);
```

Mixed scheduling is useful for scheduling programs that are too large to be fully synthesized within some time budget or controlling higher-level scheduling decisions such as pipeline decomposition or top-level tilings.

As an example, we used mixed scheduling to quickly discover a faster bfloat16-to-float32 vector–matrix multiplication kernel for gemma.cpp, an open-source inference system for Google's open-weight large language models (LLMs) [36]. Specifically, we scheduled an AVX2 bfloat16 multiplication of a 1×2048 vector by a 2048×16384 matrix that is critical to gemma.cpp. Our schedule improved throughput over the existing implementation by about 10%.

We first synthesized a representative 1×128×128, serial kernel:

```
MatmulAccum(
    [1, 1, 128, 128],
    (bf16, GL, row_major),
    (bf16, GL, layout![0, 2, 1]),
    (f32, GL, row_major),
    serial)
```

We noticed a key difference between the resulting implementation and gemma.cpp's existing implementation: a factored-out layout change.

In gemma.cpp's AVX2 implementation, bfloat16 multiplication first widens vector lanes to float32. This widening produces two float32 vectors: one containing the odd-indexed lanes and one containing the even-indexed lanes. In Morello, this is expressed as a `Move` from a `row_major` bfloat16 tensor to a float32 tensor whose layout includes an odd-even physical dimension (splitting 16-element chunks into evens followed by odds), `layout![0, 1, 2, 2 oe(16)]`.

In the original gemma.cpp implementation, this layout change occurred only inside Highway's bfloat16 dot product implementation, which the vector–matrix multiplication loop used to compute each matrix row's dot product. On targets without native bfloat16 dot-product support, Highway performs the bfloat16-to-float32 widening internally. Widening the bfloat16 values from the current matrix row is still necessary for each row, but the other dot-product operand comes from the same input vector.

As a result, gemma.cpp repeatedly paid the widening cost for vector values that were reused across many row dot products.

Because Morello represents the same conversion as a `Move` whose cost is modeled, the schedule can choose where it belongs. The synthesized implementation hoisted the vector conversion into a prologue that materializes an L1-resident per-thread buffer of blocks of even and odd values. This pays the vector widening and relayout cost once, then amortizes that cost across the row dot products. The row microkernel still widens the bfloat16 values loaded from each matrix row, but it no longer repeats the conversion for the reused input vector.

We then used this synthesized kernel as a guide for a mixed schedule of the full vector–matrix multiplication. The schedule manually hoists the vector `cast` into the odd-even layout and fixes the top-level tiling and parallelization; `synthesize_all` fills in the remaining sub-Specs:

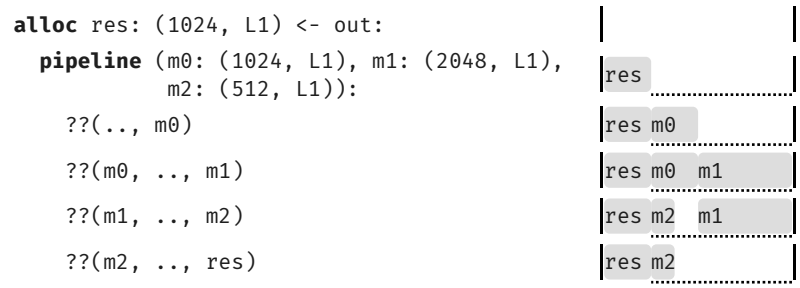
```
let spec: Spec<Avx2Target> = spec!(MatmulAccum(
    [nz!(1u32), M, K, N],
    (bf16, GL, row_major),
    (bf16, GL, bcm_layout),
    (f32, GL, row_major)
));

let interleaved = layout![0, 1, 2, 2 oe(16)];
let implementation = spec
    .cast(0, Float32, L1, interleaved, None)
    .subschedule(&[0], |mv| mv.tile_out(&[1, 1, 16]))
    .tile_out_parallel(&[1, 1, 128])
    .tile_out(&[1, 1, 1])  // treat columns independently
    .synthesize_all(&db, None);
```

This allowed us to evaluate the production vector–matrix sizes used in gemma.cpp without fully synthesizing the entire 1×2048×16384 implementation before integrating the change into gemma.cpp.

2.9. CODE GENERATION

Once an implementation tree has been fixed, a program in the Morello IR is lowered to C. The IR itself is structurally similar to C and fixes important optimization decisions, so efficient code generation can be implemented as syntax-directed translation in a single pass over the program. Morello lowers by mapping programs to C source code strings that contain the source code of nested programs (or accumulate `#include` statements to be prepended).

Listing 1. An example program in the Morello IR alongside the placements of live tensors' storage in the workspace buffer at the point that line begins executing. The `alloc`-binding places `res`'s buffer at the earliest free address, which is the beginning of the workspace. The `pipeline` allocates a region indicated by a dotted line, within which it alternates placing intermediate tensors at the beginning (`m0` and `m2`) or end (`m1`) of that region.



2.9.1. STATIC MEMORY PLANNING

To avoid the runtime cost of heap allocations for buffers that don't fit on the stack, Morello plans memory statically. Because the Morello IR uses destination-passing style and has both static control and static tensor sizes, the compiler can easily determine the lifetimes of buffers and pack all heap allocations into a single, global *workspace* allocation. In Morello, planning is done for each thread scope: a workspace is allocated at the start of execution and others for each launched threads.[12]

[12] Shaikhha et al. describe a similar optimization in [59], which transforms a functional array language into destination-passing style for the purpose of hoisting and statically planning allocations. MLIR likewise implements a "one-shot bufferization" pass [45].

Before code is generated, a simple analysis determines the size of the needed workspaces from the maximum number of bytes that will be live on the heap at any point during execution. During code generation, the workspace is shared among heap-backed buffers by maintaining a quasi-stack discipline. Alloc-bindings use the workspace region beginning at the lowest free address. The interlocking lifetimes of tensors introduced by `pipeline` nodes preclude a pure stack discipline; intermediate tensors do not strictly outlive all others. Instead, a `pipeline` reserves a region large enough to contain the largest pair of concurrently live tensors, then alternately places tensors at the beginning or end. See Listing 1 for an example.

As with dynamic memory allocators, there is overhead from fragmentation. In Morello's case, this results from the inner gaps introduced `pipeline` nodes; a program of only `alloc`-bindings has no fragmentation. Simple improvements would be to reduce the size of pipeline regions at the beginning and tail of the pipeline (i.e., when only `m0` or `m2` in Listing 1) and to greedily place buffers inside gaps rather than strictly after `pipeline` regions. Of course, increased interest in DNNs has led to more effective and sophisticated methods of static memory planning (e.g., [40, 61]) which

could be integrated straightforwardly. Addressing optimal memory planning is out-of-scope for this dissertation.

2.10. CONCLUSION

This chapter introduced Morello, a hierarchical IR in which specifications denote incomplete sub-programs and schedules complete them by rewriting those specifications into loops, data movement, pipelines, and microkernel calls. The language makes the main performance-critical choices of DNN code generation explicit: tiling and fusion over composed operators, memory placement and layout transformations, and SIMD and thread-level parallelism.

The compositional IR and scheduling rewrites we introduced in this chapter form the foundation for the remainder of this dissertation. In the next chapter, we formulate the problem of synthesizing implementations in this IR as the optimization problem posed in the introduction.

3. DYNAMIC-PROGRAMMING-BASED SYNTHESIS

Having defined the Morello IR, we turn to the problem of generating optimal implementations in that IR. As mentioned, a key contribution of this dissertation is an optimization method based on dynamic programming. The dynamic program indexes sub-problems by Specs, and the solution to each sub-problem is an optimal rewrite of the Spec, which corresponds to a sub-program in the IR.

This chapter introduces the cost model that defines the method's optimization objective (Section 3.1) and discusses the top-down dynamic programming algorithm (Section 3.2) and its asymptotic time complexity (Section 3.3). Additionally, the chapter develops a bottom-up algorithm to compute optima for many Specs in parallel (Section 3.4).

3.1. COST MODEL

Morello aims to minimize the reciprocal throughput (cycles-per-execution) of synthesized code while respecting memory constraints. For example, in our x86 target model, **`vfmadd231ps`** has a reciprocal throughput of 0.5 because two instructions can be issued per cycle.

To that end, we define a cost model $c : \mathscr{P} \times \mathbb{N}^{|M|} \to \mathbb{R}^+ \cup \{\top\}$ where $\mathscr{P}$ is the set of programs expressible in Morello's IR, $\mathbb{N}^{|M|}$ are limits on the peak allocation in each memory $m \in M$, and $\mathbb{R}^+$ includes all non-negative reals.[13] Programs that violate memory constraints map to $\top$, and others map to an estimate of reciprocal throughput. This section first introduces the throughput cost model $c^* : \mathscr{P} \to \mathbb{R}^+$, then defines c by extending c^* to respect memory constraints.

[13] In practice, costs are multiples of 0.5. As a small performance optimization, Morello encodes costs as unsigned integers.

We model reciprocal throughput assuming that no data dependencies impede pipelining. Throughput typically predicts execution time accurately in the most common setting for deep learning workloads: processing large batches of inputs where work is available for all functional units and across pipelined loop iterations. Additionally, for optimal programs, the reciprocal throughput composes well over the program tree: the highest-throughput program tends to be composed of the highest-throughput sub-programs. We expect this model will accurately rank sets of optimal programs but will be inaccurate for programs with scant total work or false dependencies. Fortunately, we are primarily concerned with the former. A multi-objective extension of this cost model might model execution

time in different settings more accurately by separating throughput and latency, but we leave that to future work.

3.1.1. THROUGHPUT MODEL

Let p_{root} be the root node of an implementation $p \in \mathscr{P}$. Its immediate children are denoted $p_1, .., p_n$. We define the throughput cost model c^* for all program types by cases: c^*_{loop}, c^*_{alloc}, c^*_{pipeline}, c^*_{block}, and c^*_{kernel}.

First, recall from Chapter 2 that a loop p has both a main body and zero or more boundary bodies. The main body is executed $\mathsf{iterations}(p)$ times, and each boundary body is executed once. We denote the main and boundary body sub-programs as p_{main} and $\cup_{i \in \ \mathsf{boundaries}(p)} p_i$ respectively.

For the target machine, let P be its modeled processor count, setup the fixed parallel-loop overhead, and batch the overhead for each group of at most P concurrently executed iterations. The loop cost is:

$$c^*_{\mathsf{loop}}(p) = \sum_{i \in \ \mathsf{boundaries}(p)} c^*(p_i) + \begin{cases} \left\lceil \frac{\mathsf{iterations}(p)}{P} \right\rceil \cdot (c^*(p_{\text{main}}) + \mathsf{batch}) + \mathsf{setup} & p \text{ is parallel} \\ \mathsf{iterations}(p) \cdot c^*(p_{\text{main}}) & p \text{ is serial.} \end{cases}$$

We calibrated setup and batch using microbenchmarks on the target machines.

The two memory-allocating program nodes, `alloc`-bindings and pipelines, accumulate *tensor costs* $c_t(t)$ for each accessed tensor t:

$$c^*_{\mathsf{alloc}}(p) = c_t(p_{\text{src}}) + c_t(p_{\text{dest}}) + \sum_i c^*(p_i)$$

$$c^*_{\mathsf{pipeline}}(p) = \sum_{t' \in p_{\mathsf{intermediates}}} c_t(t') \quad + \sum_i c^*(p_i)$$

The tensor cost models the reciprocal throughput of streaming from the tensor's underlying memory. It is calculated as the product of the memory's reciprocal throughput ($\mathsf{rspeed}_{\mathsf{memory}(t)}$), the estimated number of accessed cache lines ($\mathsf{lines}(t)$), and a penalty $\kappa(t)$ for non-contiguous access:

$$c_t(t) = \mathsf{lines}(t) \cdot \mathsf{rspeed}_{\mathsf{memory}(t)} \cdot \kappa(t),$$

$$\kappa(t) = \begin{cases} 1 & t \text{ is fully contiguous} \\ 2 & \text{otherwise.} \end{cases}$$

The cache line estimate, $\mathsf{lines}(t)$, is an upper bound. It is computed by rounding up the byte size of the contiguous sub-tensor to a number of cache lines, then multiplying by the number of positions in the remaining non-contiguous dimensions. This pessimistically assumes that every non-

contiguous step fetches a new cache line. The additional binary penalty $\kappa(t)$ is a coarse heuristic for address-generation and irregular-access overhead; it does not distinguish among non-contiguous stride patterns.

Finally, we model the cost of blocks as the sum of each of its subprograms:

$$c^*_{\mathsf{block}}(p) = \sum_i c^*(p_i).$$

Kernel costs c^*_{kernel} were chosen manually for each kernel by running microarchitectural simulations with llvm-mca.

3.1.2. MEMORY LIMITS

In addition to maximizing throughput, Morello respects constraints on memory consumption, as we described in Section 2.4.1. To do this, we model the peak memory consumption in memory m with the function $\mathsf{peak}_m(p)$. In short, `alloc`-bindings, pipelines, and some microkernels allocate memory, increasing peak memory consumption for relevant memories over what was allocated by their children:

$$\mathsf{peak}_m(p) = \max_i \mathsf{peak}_m(p_i) + \begin{cases} \mathsf{units}_m(p_{\text{dest}}) & p \text{ is an } \texttt{alloc}\text{-binding} \\ \mathsf{pipealloc}_m(p) & p \text{ is a pipeline} \\ \mathsf{kalloc}_m(p) & p \text{ is a kernel} \\ 0 & \text{otherwise.} \end{cases}$$

Here, $\mathsf{units}_m(t)$ is the amount of memory allocated by tensor t in memory m. As mentioned in Section 2.2, Morello counts individual registers for scalar and vector register files and counts cache lines for other memories. Additionally, cache line units are discretized to zero or a power of two.

The maximum additional memory allocated by a pipeline is modeled by:

$$\mathsf{pipealloc}_m(p) = \max_{(t,t') \in (p_{\text{intermediates}})^2 \mid \exists_i p_i \phi(p_i,t,t')} \mathsf{units}_m(t) + \mathsf{units}_m(t'),$$

where $\phi(p,t,t')$ is true if the application of p takes t as its first input argument and t' as its output argument (i.e., t and t' have overlapping lifetimes per Section 2.3).

$\mathsf{kalloc}_m(p)$ counts storage used internally by the kernel, such as accumulator and helper registers, if any.

Finally, we extend our cost model to accept limits on the available memory in each memory m and map programs that violate those limits to $\top$:

$$c\big(p, b_1, .., b_{|M|}\big) = \begin{cases} c^*(p) & \forall_{m \in M} \mathsf{peak}_m(p) \leq b_m \\ \top & \text{otherwise.} \end{cases}$$

Additionally, as a convenience for the next section, we externalize the recursion in $c\big(p, b_1, .., b_{|M|}\big)$ to define the function $d\big(p_{\text{root}}, b_1, .., b_{|M|}, \vec{x}\big)$, where $\vec{x}$ is the vector of child costs for root node p_{root}.

3.2. TOP-DOWN SYNTHESIS

Morello's synthesis algorithm maps a Spec s to the optimal member of $\mathsf{rewrites}(s)$, which is the rewrite that leads to the lowest-cost implementation of s. We now describe the optimization problem and the corresponding algorithm.

3.2.1. PROBLEM

The lowest cost achievable by an implementation of s is:

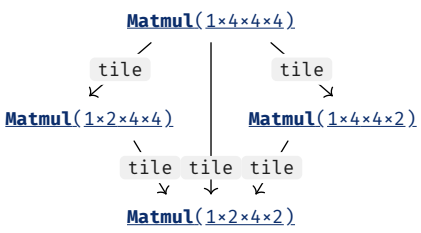
$$\mathsf{synth}^*(s) = \min_{r \in \mathsf{rewrites}(s)} d(r(s)_{\text{root}}, s_{\text{limits}}, .., \mathsf{subcosts}(r, s))$$

$$\mathsf{subcosts}(r, s) = [\mathsf{synth}^*(r(s)_1), .., \mathsf{synth}^*(r(s)_n)].$$

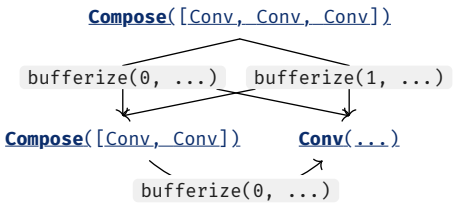
We use the convention that $\forall_{x \in \mathbb{R}^+}\ x < \top$, so $\mathsf{synth}^*(s) = \top$ if all rewrites have cost $\top$ or $\mathsf{rewrites}(s)$ is empty. Additionally, without loss of generality, we assume all rewrites introduce a node $r(s)_{\text{root}}$ and zero or more nested applications of Specs $r(s)_1, .., r(s)_n$.

The set of explored rewrites $\mathsf{rewrites}(s)$ includes all rewrites described in Chapter 2 with the following four exceptions.

1. First, the set includes only power-of-two tilings. As a result, Spec dependencies produced by tiling fall within the memoization database's coordinate space.
2. Second, the set excludes both serial tilings over multiple dimensions and parallel tilings that exceed 2 dimensions. Serial tilings over multiple dimensions are represented through a sequence of single-dimension tilings.
3. Third, the set only includes moves into layouts derived from a fixed set of 3 base orderings, extended with packed variants. Specifically, construct the set of destination layouts by first including row-major, column-major if the tensor has exactly two non-one logical dimensions, and NHWC if the tensor has exactly four non-one logical dimensions. For each of those layouts, extend the set with every layouts produced by appending either a packed or odd-even inner dimension, for all possible logical dimensions, of a size to fit either one or two vector registers. The resulting set includes all packings commonly seen in GEMM and convolution implementations.



(a) Matrix multiplication can be implemented by a loop of matrix multiplication implementations with smaller dimensions. Matrix size symmetries resulting from interleaving single-dimension tilings are broken by memoization.



(b) A pipeline of 3 convolutions can be implemented by a partial program that executes either a single convolution followed by a composition of 2 convolutions or vice versa. In either case, the resulting sub-programs are the same: a single convolution and a shorter pipeline of 2 convolutions. Memoization prevents exponential blow-up with respect to the choice of pipeline decomposition (the `bufferize` rewrite in Morello).

Figure 8. Two simple examples of Specs being decomposed into smaller, overlapping sub-problems. Multiple incoming edges indicate overlapping sub-problems; these are cases where a sub-program implementation can be memoized and re-used.

4. Fourth, moves target only data in the same or a faster memory (e.g., from and to global memory, or from L1 to a register file, but never from a register file to global memory).

Notice that this optimization problem exhibits both optimal substructure and overlapping sub-problems (Figure 8). The optimal rewrite r can be computed solely from the Spec and optimal Spec dependency costs, and optimal implementations of smaller Specs are reused across many larger implementations. Optimization decisions are independent given a Spec, and both the cost model d and Spec limits are defined such that a program's cost is component-wise non-decreasing in the vector child costs.

3.2.2. ALGORITHM

We define $\mathsf{synth}(s)$ to return both the cost $\mathsf{synth}^*(s)$ and the corresponding rewrite r. Using this optimal substructure, we compute $\mathsf{synth}(s)$ with an exhaustive, memoizing search. If the memoization database contains a solution for that Spec, the search procedure returns it. Otherwise, the procedure evaluates rewrites applicable to the goal Spec, recurses on those rewrites' Spec dependencies, and memoizes and returns the optimal rewrite-cost pair. If a dependency has cost $\top$ (i.e., has no valid implementation) the originating rewrite is pruned, so its remaining dependencies need not be visited. This can reduce search time.

When two programs satisfying the same Spec have the same cost, synthesis chooses the program that minimizes a lexicographically ordered tuple of peak memories from fastest to slowest. If those tuples are equal, then synthesis chooses the shallowest program (smallest $\mathsf{depth}(p)$). Beyond that, the order is undefined. Note that this is not equivalent to minimizing a lexicographically ordered cost $(c(p), \mathsf{peak}_m, .., \mathsf{depth}(p))$, which is not monotonic with respect to the combined costs of children, despite being component-wise monotonic. As a result, the corresponding optimization problem lacks an optimal substructure.

Synthesized solutions are guaranteed to have optimal costs $c(p)$ not only for the queried Spec, but for every otherwise-identical Spec with memory limits in the range bounded inclusively by the query Spec's memory limits and the peak memory consumption of the solution. If there were a lower-throughput implementation at any point below the query limits, then that implementation would have been the result instead, and the implementation p would be guaranteed to be valid ($c\left(p, b_1, .., b_{|M|}\right) \neq \top$) for limits b_i that meet or exceed its peak memory. Likewise, if there were no valid implementation at all (cost is $\top$), then there would be no valid implementation at any lower limit, either (i.e., in the multi-dimensional range $\left([0, b_1], .., \left[0, b_{|M|}\right]\right)$). As an optimization, memoization fills the entire range with the solution, not just the queried Spec. As we discuss in Chapter 5, filling multi-dimensional ranges with a single solution is efficient since it does not require enumerating all limits in the range.

The main advantages of this algorithm are that it is complete and simple to implement. It does not depend on integer linear programming (ILP) or more general solvers, which perform unpredictably. Approximate dynamic programming (including reinforcement learning) typically does not guarantee that the optimal program under the model has been returned, so there is a chance that running the algorithm longer might yield better results.

3.3. TIME COMPLEXITY

By casting DNN optimization as a dynamic programming problem, we significantly reduce search time complexity compared to exhaustively enumerating scheduling decisions. Exhaustive enumeration has time complexity $O\left(k(s)^d\right)$, where $k(s)$ upper-bounds the number of rewrites applicable to any sub-Spec reachable from the Spec s (i.e., any sub-Spec obtainable from s via zero or more rewrite applications), and d is the maximum depth of the reduction tree. In contrast, the dynamic program

described in Section 3.2 has time complexity $O(|R(s)|k(s))$, where $R(s)$ denotes the set of sub-Specs reachable from s. Many rewrites commute (e.g., single-dimension tilings), which collapses many equivalent rewrite sequences into shared sub-Specs. As a result, $|R(s)|$ is exponentially smaller than $k(s)^d$.

Nevertheless, though faster, our synthesis algorithm has non-trivial runtime for large Specs. Its asymptotic time complexity is dominated by $|R(s)|$.

This section details an upper bound on asymptotic time complexity parameterized by the query Spec and the target machine model. (It treats the set of Specs and rewrites as fixed.) We derive an upper bound on $|R(s)|$ in Section 3.3.1 and an upper bound on $k(s)$ in Section 3.3.2.

3.3.1. NUMBER OF SPECS

First, we define a relation between Specs. Let $s \to s'$ if s' is a sub-Spec produced by any rewrite of s. More formally,

$$s \to s' \iff \exists r \in \mathsf{rewrites}(s) : s' \in \{r(s)_1, .., r(s)_n\}$$

where $\mathsf{rewrites}(s)$ and the sub-Specs $r(s)_1, .., r(s)_n$ are as defined in Section 3.2. Let $\overset{*}{\to}$ denote the reflexive-transitive closure of $\to$, and let $R(s)$ denote the closure image of s under $\overset{*}{\to}$: $R(s) = \left\{s' : s \overset{*}{\to} s'\right\}$. Any Spec in $R(s)$ is said to be *reachable* from s.

Next, we define an asymptotic upper bound on the number of reachable Specs:

$$|R(s)| \in O(\mathsf{shapes} \cdot (\mathsf{tensor\ specs})^P \cdot \mathsf{factors} \cdot (\mathsf{memory\ limits})).$$

This bound is a product of upper bounds on the number of tilings, parameters, ways to decompose s into factors, and memory limits induced by memory movement rewrites. Each of these terms is implicitly parameterized by s.

shapes is a bound on the number of unique Spec shapes produced by tiling rewrites.[14] Synthesis applies only power-of-two tilings, so:

$$\mathsf{shapes} = \prod_{d \in\, \mathsf{shape}(s)} \log_2(d) + 1,$$

where $\mathsf{shape}(s)$ is the tuple of logical dimension sizes of s.

Morello scales efficiently to Specs with large dimension sizes: shapes grows logarithmically with dimension sizes since the per-dimension factor $\log_2(d) + 1$ grows slowly. For example, synthesizing a 1×2048×2048×2048 **`Matmul`** is more expensive by a polylogarithmic factor of 64x than a 1×4×4×4 **`Matmul`**. In addition, shapes grows exponentially in the number of dimensions since each new dimension adds another multi-

[14] As defined in Section 2.2, a Spec shape is the tuple of logical dimensions describing the Spec's coordinate space (e.g., $B \times M \times K \times N$ for a matrix multiplication). Tensor parameter shapes (e.g., $B \times M \times K$ and $B \times K \times N$) are derived from that tuple via the Spec's indexing relations.

plicative factor. A fifth dimension of size 4 would triple the count; one of size 2048 would multiply it by 12. These factors are relatively small in practice because the base of the exponential is logarithmic in dimension sizes.

tensor specs is a bound on the number of unique per-parameter tensor Specs encountered across reachable sub-Specs of s under rewrites. Each tensor Spec (Section 2.2) consists of a memory, layout, and dtype. Again, the number of tensor Specs can be expressed as a product of the number of choices for each field:

$$\text{tensor specs} \in O(\text{tensor rank} \cdot \text{memories} \cdot \text{layouts} \cdot \text{dtypes}.)$$

To express this upper bound as a single base raised to P, rather than a product of distinct per-parameter terms, each factor in tensor specs is the maximum over all parameters in s. The factors are:

- tensor rank, the maximum number of logical dimensions among all parameters in s. For primitive Specs, this typically equals or is slightly less than |shapes| (e.g., the first parameter of a **`Matmul`** has rank-3 tensors, while the Spec shape has rank 4). For **`Compose`** Specs, tensor parameter ranks tend to remain small while the Spec rank grows.
- memories, the number of memories in the target model, which is 4 for all implemented targets.
- layouts, the number of distinct tensor layouts included in the search space for tensor parameters of rank tensor rank. Recall from Section 3.2 that synthesis evaluates $B = 3$ fixed base orderings of a tensor's non-one logical dimensions, optionally extended with a standard or odd-even packing that fills a small number of the target's vector registers. This yields up to $B \cdot (2 \cdot \text{tensor rank} + 1)$ distinct orderings. Additionally, layouts carry contiguousness values, of which there are tensor rank $+ 2$ possibilities.[15] Thus, we have:

  $$\begin{aligned}\text{layouts} &= B \cdot (1 + 2 \cdot \text{tensor rank}) \cdot (\text{tensor rank} + 2) \\ &\in O(\text{tensor rank}^2).\end{aligned}$$

  The number of layouts grows only quadratically in tensor rank, which is, fortunately, usually between 3 and 5 inclusive for deep learning workloads.
- dtypes, the number of data types obtainable via data type widening from the query Spec's data types. As mentioned in Section 2.4.3, the only possible widening conversion in the Morello IR is from bfloat16 to float32, so this term is at most 2.

[15] `OnePrefix` is an exception: its output has one additional dimension of size 1, but size-1 logical dimensions are elided from layouts, so the bound is unchanged.

Among these factors, layouts is the dominant contributor to the base of $(\text{tensor specs})^P$. For example, a **Matmul** has rank-3 tensor parameters, so tensor rank $= 3$, memories $= 4$, and dtypes ≤ 2, while layouts $= 105$. Of layouts's factors, B is a natural lever for expanding Morello's expressiveness. For example, adding a new base layout (e.g., CHWN) increases B by one and therefore increases $(\text{tensor specs})^P$ by a factor of $\left(\frac{B+1}{B}\right)^P$. If $B = 3$, increasing B to 4 scales $(\text{tensor specs})^P$ by $\left(\frac{4}{3}\right)^P$, or about 2.4 times at $P = 3$.

P is the maximum number of parameters among reachable sub-Specs of s. Note that P is not necessarily the number of parameters of s itself. Instead, let $P = \max(p_s, 2)$ where p_s is the number of parameters of s. Clamping P to a minimum of 2 accounts for the two cases in our rewrite system where a rewrite produces a Spec with more parameters than its parent, both of which yield at most 2 parameters from a 1-parameter Spec: (1) applying a data movement rewrite to a 1-parameter Spec produces a **Move** sub-Spec with two parameters, and (2) applying `bufferize` to a 1-parameter **Compose** (e.g., **Compose**`(Passthrough, RandomGen)(out)`) introduces an intermediate buffer, yielding sub-Specs whose consumer has an additional external input. In all other cases, the number of parameters is non-increasing across rewrites.

In practice, P is small for most workloads. Primitive Specs have between 1 and 4 parameters (see Table II), and P typically equals p_s. For **Compose** Specs, P equals the number of external tensor operands, i.e., those not produced and consumed entirely within the composition. Operations that consume only the output of a preceding sub-computation, such as an elementwise activation, do not change P unchanged, whereas operations that introduce an independent input, such as a weight matrix, each increment P by one.

factors is the number of unique, contiguous subsequences of primitives (including the original sequence) obtained from s by recursively partitioning via `bufferize` rewrites. For a composition of N primitive Specs:

$$\text{factors} = \binom{N+1}{2} \in O(N^2).$$

This bound is precise if each factor is unique, but it overapproximates when the composition has repeating factors, such as a composition of alternating **Matmul** and **Softmax** Specs.

The combination of P, shapes, and factors clarifies pipeline composition costs. Handling long pipelines can be exponentially expensive, but driven by the introduction of new parameters and dimensions rather than graph partitioning. Other optimizers typically explore all possible ways

to partition a computation graph into sub-graphs, relying on heuristics to discard some partitions and reduce the search space. Morello instead optimizes and memoizes unique sub-graphs (e.g., a feedforward layer, which is a matrix multiplication and an activation) just once, as illustrated in Figure 8b. As a result, the search space grows only quadratically in the number of operations (composed primitive Specs), matching the factors bound. Appending elementwise activations, for example, increases runtime only quadratically.

However, the expensive case is composition patterns that introduce new layer parameters or dimensions corresponding to new external inputs and outputs. For example, adding a matrix multiplication (perhaps as part of a feedforward layer) requires a new weight parameter, though its dimensions are fully determined by the input and output shapes; therefore, it does not increase the number of Spec dimensions. In contrast, adding an embedding layer introduces both a new parameter and a new dimension for the vocabulary size. This growth in P and shapes is the primary source of exponential complexity when compiling larger programs.

Finally, memory limits is an upper bound on the number of unique memory limit tuples that synthesis considers. For each memory m, let V_m be the set of valid limits for that memory: $V_m = \mathbb{N}$ for register files or $V_m = \{0\} \cup \{2^v : v \in \mathbb{N}\}$ for caches and main memory. We define the upper bound on memory limits to be the product over all memories:

$$\text{memory limits} = \prod_{m \in M} |\{v \in V_m : v \leq b_m(s)\}|,$$

where $b_m(s)$ is the memory limit of Spec s for memory m, as noted in Section 3.2.2.

In practice, this bound overstates the number of memory limits per otherwise-identical Spec because the range-filling optimization described in Section 3.2.2 visits fewer limit tuples than the full product (usually by an order of magnitude).

Two quantities not included in the bounding function are the serial-only flag and the number of distinct Spec types and ranks in this set of reachable sub-Specs (e.g., synthesizing a **`Matmul`** requires synthesizing **`MatmulAccum`**, **`Fill`**, and **`Move`** Specs). Like the serial-only flag, the number of Spec types is fixed for a given IR and contributes only a constant factor, so it is subsumed by the asymptotic bound. The depth of the inter-type dependency chain—i.e., how many distinct Spec types appear among reachable sub-Specs of s—could in principle be expressed as a query-Spec-dependent term, but in practice it is bounded by a small, IR-determined constant.

Given the optimal costs of dependent Specs, a Spec's optimal implementation is computed by applying a set of rewrites to compute the minimum implementation cost. Optimal implementation time is an affine combination of the time to evaluate each rewrite. We now define a polynomial upper bound on this time, showing that it is polynomial and asymptotically dominated by $|R(s)|$.

Ignoring constant factors, the overhead of loading and memoizing solutions in the database, and rewrites that do not grow with the Spec or machine model, we define a polynomial upper bound on the maximum number of rewrites considered for any Spec in $R(s)$ as

$$k(s) \in O(\mathsf{tilings} + \mathsf{moves} + \mathsf{splits}).$$

We next define each term and show that per-Spec complexity is polynomial.

First, search considers tilings. Recall that synthesis considers at most 2 dimensions simultaneously. Let $d_1 \geq \cdots \geq d_n$ be the non-increasing entries of $\mathsf{shape}(s)$, and define:

$$\mathsf{tilings} = \prod_{i=1}^{\min(n,2)} \log_2 d_i.$$

Second, parameters can be moved to any memory connected to and faster than the parameter's current memory. For simplicity, we assume that any memory is a possible destination. Additionally, moves can change layout or, in some cases, data type. We define:

$$\mathsf{moves} = P \cdot \mathsf{memories} \cdot \mathsf{layouts} \cdot \mathsf{dtypes}.$$

Finally, there are several ways to decompose a **`Compose`** or many of the softmax-family Specs. Applying `bufferize` to a **`Compose`** of N primitive Specs considers $N-1$ split points, each with $\mathsf{memories} \cdot \mathsf{layouts}$ choices for the intermediate tensor's memory and layout: $O(N \cdot \mathsf{memories} \cdot \mathsf{layouts})$ candidates. Some softmax-decomposing rewrites allocate two intermediates, yielding $O(\mathsf{memories}^2\mathsf{layouts}^2)$ choices. Combining these rewrites, we define:

$$\mathsf{splits} = \underbrace{(N-1) \cdot \mathsf{memories} \cdot \mathsf{layouts}}_{\text{Compose decompositions}} + \underbrace{\mathsf{memories}^2 \cdot \mathsf{layouts}^2}_{\text{Softmax splits}}.$$

Some rewrites, such as rewrites that convert to an accumulating variant, apply im2col-style transformations, `BroadcastFirst`, softmax-specific rewrites, and microkernel selection, are omitted because they contribute only constants.

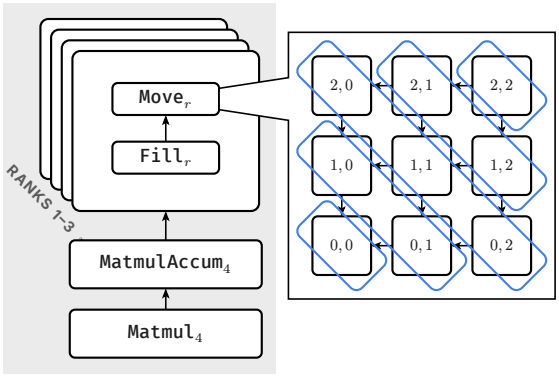


Figure 9. The data dependency graph underlying Morello's parallel database precomputation algorithm, where edges correspond to data dependencies. Nodes correspond to sets of Specs. Blue boxes highlight diagonals of tasks that can be precomputed in parallel.

The key observation is that $k(s)$ is a sum of terms, each of which is bounded by a single factor of the product defining $|R(s)|$. For example, tilings $\leq$ shapes (the same per-dimension log factors, but over fewer dimensions), and moves $\leq$ (tensor specs)^P for $P \geq 2$, since the left side is linear in P and the base while the right side is exponential, and so on. $k(s)$ grows strictly more slowly than $|R(s)|$, so overall complexity is governed by the growth of $|R(s)|$.

3.4. BOTTOM-UP PRE-COMPUTATION

Certain operators and sub-graphs frequently recur across many DNN architectures. Rather than re-optimizing these common Specs on demand, precomputing a database of optimal implementations provides a starting point for more specific queries. Morello implements a parallel procedure to populate the database with optimal implementations of common Specs (specifically, **`Move`**, **`Fill`**, **`MatmulAccum`**, and **`Matmul`** Specs).

This procedure is parameterized by a list of goal Specs that is topologically sorted to satisfy dependency constraints. Each goal Spec defines a sequential *phase* of the procedure. During each phase, the procedure synthesizes an overapproximation of the set of Specs reachable from the goal Spec that have not already been synthesized in earlier phases (i.e., a superset of R(goal) with previously synthesized Specs removed). Specs that share the same type, rank, and tensor data types are synthesized in the same phase. Within each phase, Specs are partitioned into independent tasks, and each task maps uniquely to a point on a grid (see Figure 9). The procedure is defined in the following listing.

```
phase_goals ← [Move(8×8, f32), ..., Fill(8×8, f32), ...]
assert is_toplogically_sorted(phase_goals)
for phase in phase_goals do
  for diagonal in diagonals(phase) do // begin at 0,..,0 diagonal
    parallel for task in tasks(diagonal) do
      compute_task(task) // compute_task is defined later
    end
  end
end
```

Here, the procedure employs a *wavefront parallelization strategy* that enumerates diagonals serially, starting from the zero-coordinate diagonal, and executes all tasks on that diagonal in parallel. Each task synthesizes its constituent Specs using the top-down procedure defined in Section 3.2 (top_down), which never recurses because dependencies are guaranteed to have been synthesized during earlier phases or diagonals.

A phase's task grid is the Cartesian product of total orders over each dimension size, memory level, and the serial-only flag, each of which is mapped to an axis of the grid. The resulting grid respects the data dependencies defined by $\xrightarrow{*}$; each task depends only on the results of tasks with component-wise smaller coordinates or those in previous phases.

Within each task, the procedure exploits the fact that implementation costs are weakly monotonic with respect to memory limits (Section 3.2.2), so it can skip large regions of the memory-limit space after finding an implementation. Tasks are computed using the algorithm defined in the following listing:

```
def compute_task(task)
  worklist ← task
  while worklist != {} do
    stage_results ← top_down(worklist)
    next_stage ← {}
    for (spec, result) in stage_results do
      if result then
        next_stage.extend(next_limits(spec, result.cost.peak))
      end
    end
    worklist ← next_stage
  end
end
```

The algorithm traverses memory limits by iteratively stepping to successively lower limits. An inner loop first computes the optimal implementation at the maximum limits in the target memory model. It then computes optimal implementations at limits just below the peak memory consumption of the resulting implementation for each memory. This process continues until all limits reach zero or no valid implementation is found. This procedure does not guarantee that the worklist exclusively contains points on the Pareto frontier of memory limits; a limit tuple may be subsumed by another already in the worklist. However, resolving such subsumed queries via cache lookups avoids introducing significant overhead in practice.

4. EVALUATION OF SYNTHESIZED CODE

In this chapter, we assess the expressiveness of the IR, sufficiency of the cost model, and speed of our synthesizer when applied to a set of important DNN modules. Specifically, we evaluate the following research questions:

- Are Morello's IR and cost model sufficient to synthesize competitive float32 matrix-multiplication and softmax implementations?
- When synthesizing softmax, can Morello's cost model correctly select between online and offline algorithms as sequence length and the degree of parallelism change?

The rest of this chapter demonstrates that Morello generates matrix-multiplication implementations competitive with mature, hand-tuned baselines and softmax implementations competitive with XNNPACK [17] and generally faster than PyTorch. It will also demonstrate the benefits of Morello's ability to choose either online or offline softmax algorithms depending on problem size.

4.1. BENCHMARK MACHINES

The experiments in this chapter were run on two machines, one equipped with an AVX-512-supporting server CPU released in 2024 and the other with an AVX2-supporting desktop CPU released in 2017. Table IV details the machines.

Each of the following experiments targets both machines to show that Morello is not over-specialized to a single ISA or processor. Naturally, code generation and microkernel selection differ depending on the available instructions; AVX-512, for instance, has extensions for float16 and bfloat16 which can obviate the need to widen those types, as well as mask registers which can simplify control flow by eliminating boundary cases. There are also important microarchitectural differences:

- The newer machine has larger caches and therefore typically changes which tilings are optimal.
- The newer machine has larger reorder buffers and load queues, which allows the processor to hide the latency of poorly scheduled load and store instructions to a much greater degree. This takes some of the pressure off of Morello to schedule loads and stores efficiently.

Table IV. Technical specifications of the machines used for performance benchmarks.

	Zen 5	**Zen 1**
CPU (AMD)	EPYC 9275F	Ryzen Threadripper 1950X
Physical Cores	24	16
Base Frequency	4.1 GHz	3.4 GHz
All-Core Peak Freq.	4.5 GHz	3.7 GHz
Single-Core Peak	4.8 GHz	4.0 GHz
L1d Cache (per core)	48 KiB	32 KiB
L2 Cache (per core)	1 MiB	512 KiB
L3 Cache	256 MiB *split into 8 instances*	32 MiB *split into 4 instances*
L1d Read (per core)	128 B/cycle	32 B/cycle
L2 Read (per core)	64 B/cycle	32 B/cycle
L3 Read (per core)	32 B/cycle	32 B/cycle
Reorder Buffer Size	448 (224 w/ SMT)	192
Load Queue	202	72
Host	"Bare metal" machine hosted by Latitude.sh	Self-hosted

In general, despite the more complex ISA, generating optimal implementations of the following programs for the newer CPU is easier because the processor is more able to accommodate inefficient code dynamically (e.g., larger reorder, load, and store buffers).

4.2. THROUGHPUT OF SYNTHESIZED PROGRAMS

We evaluate the throughput of code synthesized by Morello on two classes of programs: matrix multiplication, where performance is dominated by dense arithmetic and register/cache blocking, and softmax, where performance is more quickly limited by cache and memory bandwidth. These experiments test whether the schedules found by Morello lead to competitive implementations on contemporary CPUs when compared with optimized library baselines.

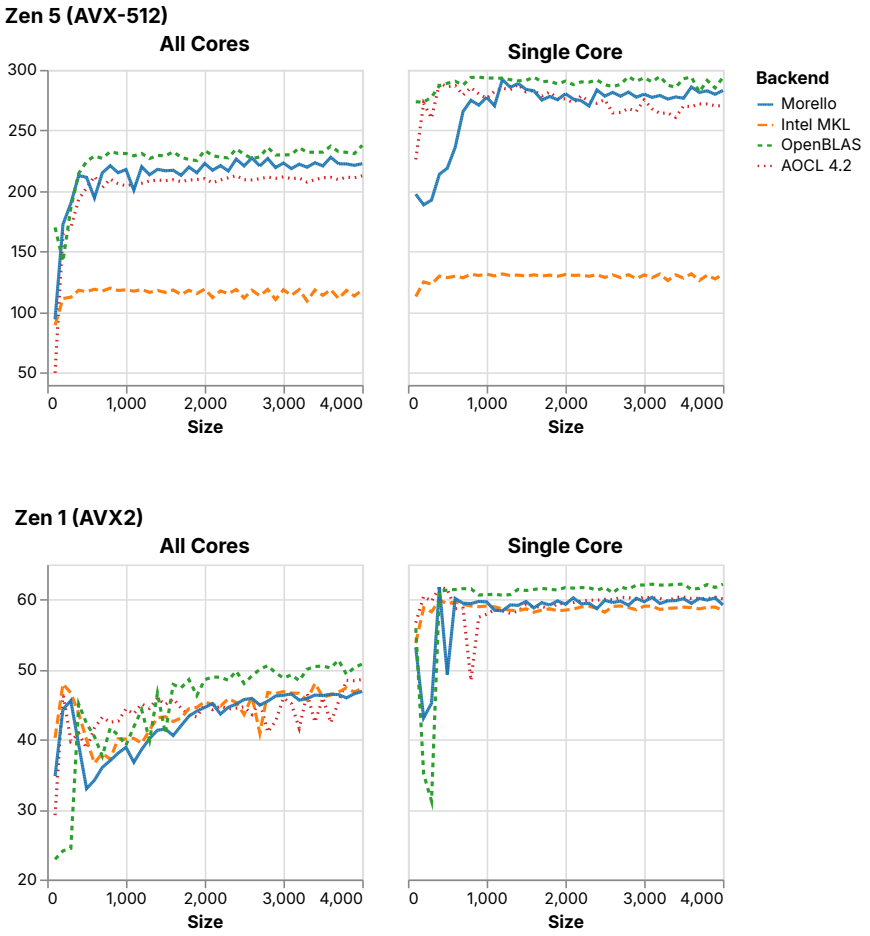


Figure 10. Throughput (GFLOP/sec/thread) of matrix multiplications across sizes $M = K = N$ that are multiples of 100 and work on float32, row-major matrices. Each point is the median of 10 measurements. Measurement variance is not plotted, but small; the median and maximum coefficients of variation are 0.41% and 1.76% for Zen 5 (AVX-512) and 0.35% and 11.13% for Zen 1 (AVX2) , respectively.

4.2.1. MATRIX MULTIPLICATION, FLOAT32

We scheduled float32 matrix multiplications for a range of square shapes by manually deciding high-level control and memory tiling identically for each machine, following the well-known BLIS [71] generalization of Goto and van de Geijn [26]. All configurations use the same manually chosen tile sizes $m_c, k_c, n_c = 528, 528, 1056$, loop order, and packed layouts. We chose tile sizes to occupy reasonable percentages of the cache, but they were not tuned in any way. MORELLO synthesized all `Move` and `MatmulAccum` SPECS with dimensions 8 or smaller. These SPEC sizes correspond to micro-kernels that are typically not portable between targets, while tiling and

packing decisions are well-understood and more often shared between targets.

We measured both single-threaded and batch-parallel performance. We chose to evaluate parallelism across the batch dimension because it models a common machine learning scenario in which many independent matrix multiplications are executed in parallel, one per inference or training example. We configured baseline SGEMMs to use a single thread and simultaneously executed multiple instances across a set of threads. We do not evaluate intra-instance parallelism in this work.

We compared performance to state-of-the-art baseline SGEMM implementations from Intel MKL 2025.2.0, AOCL 4.2, and OpenBLAS 0.3.30. We ran benchmark measurement for at least 5 iterations and 10 second, then took 10 measurements of each benchmark. Figure 10 reports the medians. The maximum wall-clock synthesis time across Morello's implementations was 29 minutes for AVX2 and 7 hours 53 minutes for AVX-512. The longer AVX-512 time is consistent with its larger search space: compared with AVX2, the target models a larger vector register file and additional 512-bit `zmm` register names.

Broadly, despite only manually scheduling the high-level loops and moves, Morello's implementations are competitive with carefully tuned baselines in terms of throughput. On the Zen 5 (AVX-512) machine, at most sizes and in both parallel configurations, Morello outperforms AOCL and underperforms OpenBLAS. Intel MKL performance on the Zen 5 machine is very poor, likely because it does not target AVX-512 on this CPU. On the Zen 1 machine, Morello is roughly tied with Intel MKL for performance, again modestly underperforming OpenBLAS and alternately under- and outperforming AOCL.

4.2.2. SOFTMAX

In addition to evaluating matrix multiplication synthesis, we synthesized softmax implementations across a range of sizes. We were particularly interested in whether Morello's cost model was sufficient to select between the two main softmax algorithms, online and offline, described in Section 2.6. We hypothesized that online softmax, which performs more arithmetic in exchange for fewer reads, would become profitable only at larger sequence lengths as the fraction of data read from and written to slower memories increased, particularly in parallel settings where memory contention is higher.

Experiment Design. In addition to XNNPACK and PyTorch baselines, Figure 11 plots the throughputs of two sets of Morello-synthesized soft-

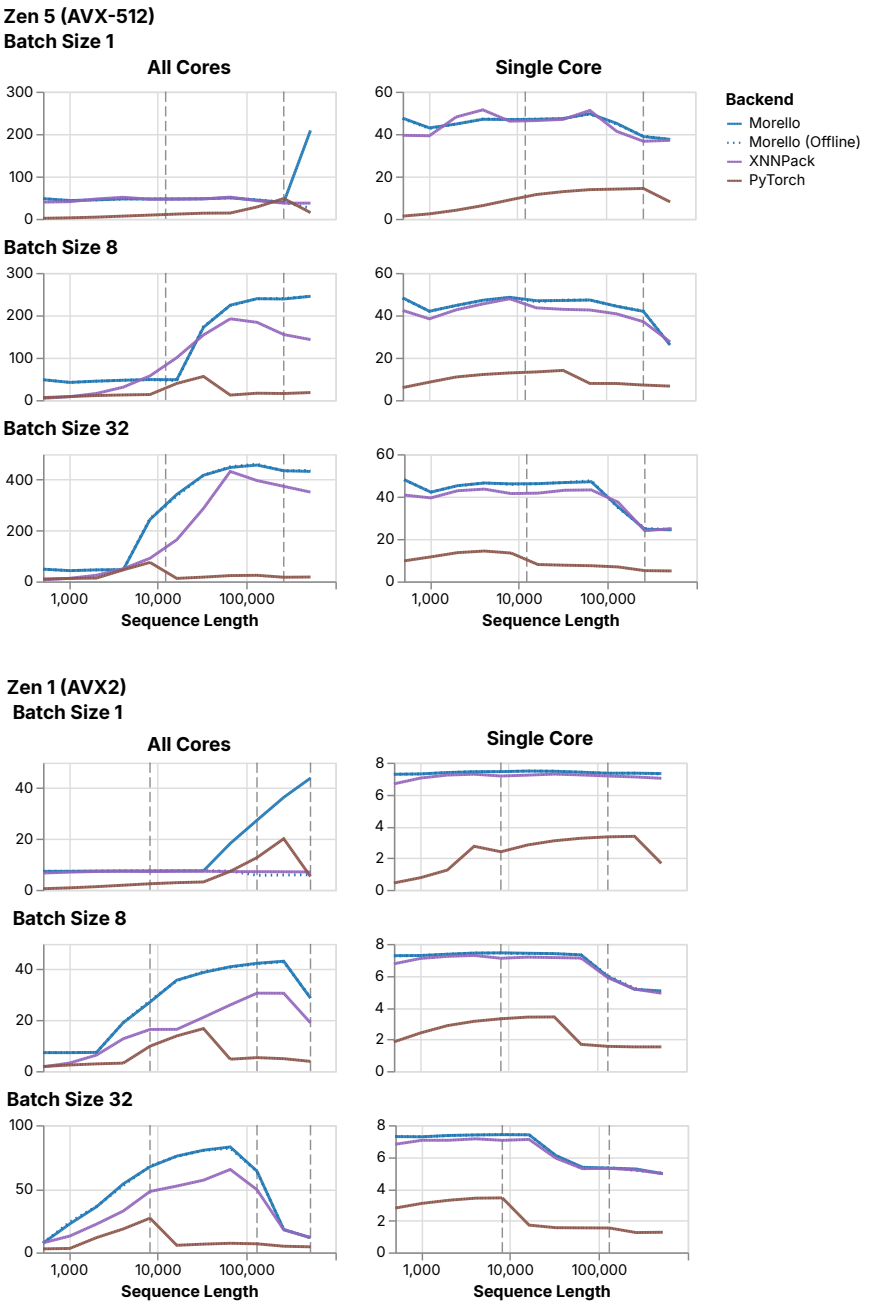


Figure 11. Throughput (GFLOP/sec) of float32 softmax implementations across sequence lengths on the two evaluation machines. As in Figure 10, each plotted value is the median of 10 measurements for one batch-size configuration, and measurement variance is omitted. Rows facet the measurements by batch size (the number of independent sequences normalized separately). Within the plotted range, vertical gray lines mark the sequence lengths where a single sequence would fill per-core L1, per-core L2, or the effective per-core share of L3 for that core configuration.

max implementations: unrestricted runs (labeled "Morello"), in which synthesis may choose either algorithm family, and runs forced to use only offline schedules. We measured only power-of-two sequence lengths and otherwise followed the matrix-multiplication benchmarking protocol described above.

Additionally, we restricted the Morello search spaces by requiring row-major intermediate layouts—that is, for storing the maximum and denominator data—and an initial, optionally parallel tiling to size 1 over the batch dimension (i.e., processing each sequence independently). In the all-core settings, we allowed Morello to use either batch-level or within-sequence parallelism. The maximum wall-clock implementation synthesis time was 12 minutes for AVX2 and 3 hours 49 minutes for AVX-512. As with matrix multiplication, this difference is consistent with the larger AVX-512 search space.

Results. As shown in Figure 11, Morello's softmax implementations generally match and frequently outperform those of the strongest baseline, XNNPACK, on both targets and across problem sizes. Morello searches implementations that may use offline or online algorithms and parallelize along either the batch or sequence dimension, whereas the XNNPACK implementation parallelizes only across the batch dimension. The online alternative is not universally beneficial: whether reduced memory traffic or additional within-sequence parallelism outweighs its extra arithmetic depends on the batch size, sequence length, and core configuration. Morello discovers this trade-off automatically, selecting an online schedule only where it pays off and retaining an offline schedule elsewhere.

For example, when computing the softmax of a single sequence on the 16-core Zen 1 target, Morello chose a serial, offline schedule at sequence length 32,768. At that length, the unrestricted and forced-offline implementations both achieve 7.5 GFLOP/sec. At lengths 65,536 and 131,072, Morello switched to a parallel online algorithm, trading additional arithmetic and two sequential fork-joins for more parallel work. The resulting schedules achieved approximately 18 and 27 GFLOP/sec, respectively: 2.5 and 4.8 times the forced-offline throughput and 2.5 and 3.8 times XNNPACK's throughput.

Together, these results show that Morello's IR and cost model are sufficient to synthesize competitive matrix-multiplication and softmax implementations on both evaluation targets. The softmax results further

show that Morello can adapt its choice and composition of online and offline algorithms to the problem size and available parallelism.

5. A SPATIAL DATABASE REPRESENTATION

Synthesis can quickly compute a large number of optimal scheduling decisions, but storing those decisions becomes difficult as problem sizes approach those of real DNNs. For example, under a 16 GiB budget, if we were to materialize every entry visited while synthesizing a `Matmul`, the the largest `Matmul` we would be able to compute has shape $B = 1$ and $M = K = N = 4$. This is too small to be useful. In this chapter, we solve this problem by introducing a database representation that can store all entries reachable from a $B = 1$, $M = K = N = 32,768$ `Matmul` under the same budget.[16]

[16] For each Spec, Morello stores the optimal rewrite decision and its cost, depth, and memory consumption, which requires 11 bytes per entry. The size-4 and size-32,768 `Matmul` tables have 1.5 billion and 4.4 trillion entries respectively.

Our database builds on the insight that optimal decisions and costs are highly regular: the optimal rewrite of one Spec is often identical to that of a Spec with a slightly varied shape or memory limits. For example, the optimal size-128 and size-64 matrix multiplications are often implemented as loops around the same microkernel. In this case, the stored rewrite for each of these matrix multiplications is `tile(k)` where `k` is the microkernel shape; the optimal rewrite matches. Additionally, after normalizing costs to the volume of the Spec—we call this the *cost intensity* of the implementation—identical scheduling decisions often have the same cost as well.

Our database design compacts these matching table entries. Our memoization database maps Specs to coordinates in $\mathbb{N}^n$ such that adjacent solutions are likely to be equal. It then stores adjacent, matching solutions as hyperrectangles that fill the space covering the matched Spec coordinates.

The remainder of this chapter will discuss the data structure itself (Section 5.1), how data are inserted and merged into rectangles (Section 5.2), how to design an encoding and coordinate mapping that compresses well (Section 5.3), and, finally, an evaluation of the database's space saved relative to materializing every entry (Section 5.4).

5.1. DATA STRUCTURE

The database consists of multiple tables, each of which has a fixed rank (number of dimensions). Morello maps every Spec to a key identifying a table and a coordinate in that table. The key contains the Spec type, shape rank, and parameter data types.[17] While tables logically map Specs to their optimal rewrite and corresponding cost intensity, program depth, and peak memory consumption, they physically represent this mapping

[17] Parameter data types could be linearized and mapped to coordinates as well. They are instead included in the table key as a convenience. Each table is stored on disk in a unique file, so we can easily copy or prune databases by table key with filesystem tools.

as high-dimensional rectangles that cover Specs with identical values for all four fields.[18] Tables are indexed by R*-Trees[19] for average $O(\log n)$ insertion and lookup where n is the number of rectangles in that table (though the specific choice of index structure is not fundamental to this approach).

More formally, we define a surjection from Specs to table keys $k : \text{Specs} \to K$ which composes the projection of a few Spec fields:

$$K = \text{Spec types} \times \mathbb{N}^+ \text{ (shape rank)} \times \text{parameter data types}$$

The remaining fields of a Spec s are mapped to a unique coordinate in the table identified by $k(s)$. The mapping is defined by the indexed family of disjoint bijections:

$$\left(f_q\right)_{q \in K} : \{s \in \text{Specs} \mid k(s) = q\} \leftrightarrow \mathbb{N}^{w(q)}$$

where $w(q)$ is the rank of the table identified by q (i.e., the number of Spec fields not in K). Together, the functions $k(s)$ and $f_{k(s)}(s)$ form a bijection between Specs and key-coordinate pairs. Visiting a value in the database is then a matter of mapping a Spec to the key, loading the associated table, and traversing the R*-Tree to the rectangle which intersects that coordinate (if any).

In practice, Morello divides tables into constant-sized blocks. Each block maps to an individual file on disk and carries a mutex, which allows parallel synthesis work to update disjoint blocks independently. Smaller blocks expose more parallelism but limit compression because rectangle merging stops at block boundaries.

[18] While each table is backed by a single R*-Tree, we also considered materializing a separate R*-Tree for each field: one for rewrites, another for cost intensity, and so on. While this transformation could potentially improve per-field compression at the cost of additional index overhead, we found that these fields are sufficiently correlated across Specs that this trade-off is not cost-effective.

[19] An R*-Tree [7] is a data structure for storing rectangles and other geometry. Each internal node stores the axis-aligned bounding box of its children, so lookup descends through bounding boxes that intersect the queried point or rectangle. Insertion splits nodes when necessary to keep bounding boxes compact and minimally overlapping.

5.2. INSERTION & COMPACTION

Finding a minimum rectangle representation is NP-hard [20]. Within each block, the problem decomposes into independent minimum rectangular partition problems, one for each distinct stored value. This family contains, as a special case, the NP-complete problem of deciding whether a three-dimensional region can be partitioned into at most k axis-aligned boxes.

Morello therefore uses the incremental, best-effort insertion procedure listed in Listing 2 and illustrated in Figure 12; we term this procedure *merge-insert*. A rectangle is inserted into the database and then greedily merged with adjacent, same-valued rectangles whenever their union forms a rectangle. (Typically, the initial rectangle results from the memory-limit-range-filling property described in Section 3.2.2 and therefore its side lengths only exceed one in the memory-limit dimensions.)

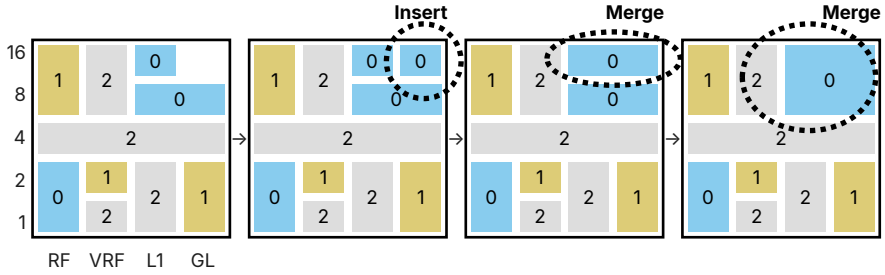


Figure 12. An illustration of Morello's database merge-insert procedure. Morello's database tables contain space-filling rectangles describing the optimal decision or cost for the Specs covered by that space. Morello inserts a rectangle into its R∗-Tree-backed database and then greedily merges it with adjacent rectangles until a fixed point. In this illustration, two dimensions are shown: a tensor-size dimension and a memory dimension. In reality, tables have at least one dimension per Spec field not in the table key.

The resulting partitioning depends on the order of merge-insert calls, and may produce more rectangles than in an optimal partitioning. This order is driven entirely by the search algorithm described in Section 3.2.2.

In practice, we optimize the algorithm in Listing 2 to avoid redundant tree traversals. Notice that the `union_is_rectangular` case restarts iteration over a fresh call to `tree.rects_intersecting(neighborhood)`, and each `tree.remove(candidate)` call appears to require an independent tree traversal. Morello instead implements these lines with one R*-Tree traversal. It removes absorbed candidates in place and continues through unvisited subtrees using the enlarged neighborhood; among previously scanned nodes, it revisits only those intersecting the newly relevant region introduced by the absorbed candidate.

Runtime Complexity. For fixed rectangle dimensionality, let n be the number of stored rectangles and g the number of rectangle-growing merges performed by an insertion. Each rectangle-growing merge can cause up to n stored rectangles to be revisited, so the merge traversal takes $O((g+1)n)$ time. Overlap repair scans $O(n)$ rectangles and emits $O(n)$ fragments, whose reinsertion takes $O(n \log n)$ time. Merge insertion therefore takes $O((g+1)n + n \log n)$ time overall. Because $g \leq n$, this is $O(n^2)$; the bound is tight because $\Theta(n)$ rectangle-growing merges can each cause $\Theta(n)$ candidate visits. However, across 4.93 million insertions in our `matmul 2` and `matmul 4` measurements, g never exceeded four and 99.993% of insertions performed at most two rectangle-growing merges. Thus, we observed no long merge chain capable of producing quadratic behavior in these runs.

Listing 2. Morello's merge-insert algorithm.

```
def merge_insert(tree, inserted)
  overlap_seen ← false
  repeat
    // expand(r) grows each bound outward by 1 with saturation.
    neighborhood ← expand(inserted)
    grew ← false
    for each candidate in tree.rects_intersecting(neighborhood) do
      if candidate.value != inserted.value then continue
      // Rectangle containment includes shared boundaries.
      if candidate.contains(inserted) then
        return
      else if inserted.contains(candidate) then
        tree.remove(candidate)
      else if candidate.union_is_rectangular(inserted) then
        tree.remove(candidate)
        inserted ← bounding_box(inserted, candidate)
        grew ← true
        break
      else if candidate.overlaps(inserted) then
        overlap_seen ← true
      end
    end
  until not grew

  // r.subtract(s) returns disjoint r-valued rectangles covering r - s.
  if overlap_seen then
    fragments ← []
    for candidate in tree.drain_same_value_intersections(inserted) do
      fragments.extend(candidate.subtract(inserted))
    end
    tree.insert_all(fragments)
  end
  tree.insert(inserted)
end
```

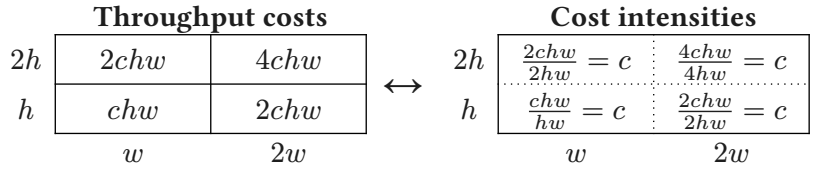
Empirical Runtime. In practice, this insertion algorithm results is sufficiently fast. While synthesizing global-memory matrix multiplications of sizes $M = K = N = 4$ and $M = K = N = 8$, 29% of total runtime is spent inserting data into the R*-Tree, while 5% is spent reading. While synthesizing a 4×4, the percentages were 15% and 4%, respectively. These synthesis jobs used a single thread, so potential lock contention is ignored. However, anecdotally, we've seen similar percentages while profiling highly parallel, bottom-up jobs.

5.3. ENCODING FOR COMPRESSION

The merge-insert procedure can merge database entries only when their stored values match and their coordinates are adjacent along an axis, so changes to the encoding or spatial arrangement of entries can substantially affect compression. This section focuses on the few important, non-obvious decisions underlying Morello's encodings and coordinate mappings.

5.3.1. COST INTENSITY

Morello does not directly store the throughput cost $c^*(p)$ of an implementation p because it changes rapidly with tiling. For example, a Spec with twice the shape volume typically requires about twice as much work and therefore has roughly twice the throughput cost. Instead, Morello stores the implementation's cost intensity: $c^*(p)$ divided by its shape volume. The following two-dimensional (h and w) example shows this normalization when throughput cost is c times shape volume:



Dividing out this predictable growth makes stored costs more nearly invariant across shape dimensions, increasing the number of matching entries that the database can merge.

5.3.2. COORDINATE MAPPING

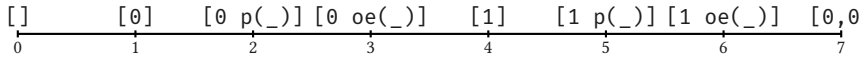
The cost-intensity encoding reduces the number of distinct stored values, thereby increasing the odds of a merge; coordinate mapping addresses the other requirement for merging by determining which Specs can be adjacent in the database. Most Spec features map to coordinates straightforwardly: memory levels, for instance, are ordered from fast to slow, and the serial-only flag maps to 0 or 1. We therefore discuss only the two non-obvious cases: shapes and layouts.

Shape Discretization. Consistent with the tile sizes included in our synthesizer search space, tile sizes are discretized as powers of two when mapped to $\mathbb{N}^n$ such that each successive Spec along a shape dimension is twice the preceding one. The primary coordinate mapping is not defined for non-power-of-two tensor shapes, so Morello stores those Specs in a

fallback hash table that maps Specs-without-memory-limits to R*-Trees spanning memory limits.

Layout Mapping. Morello maps each layout to three coordinates: one for its physical-dimension sequence, one for its packing sizes, and one for its contiguousness. We distribute layout information this way to expose opportunities for compression, limit the amount of information that must fit in any single unsigned 32-bit database coordinate, and allow inexpensive encoding and decoding.

For a tensor of logical rank r, the first coordinate omits the packing sizes of packed and odd-even dimensions. It encodes each physical dimension, from outermost to innermost, as the base-$3r$ digit $3i + k$. Here, i is the logical-dimension index, and k is 0, 1, or 2 for a standard, packed, or odd-even dimension, respectively. The encoding enumerates these digit strings first by sequence length—the number of physical dimensions—and then by ordinary radix value. The following shows the beginning of this raw coordinate space for a rank-two tensor:



Meanwhile, the second coordinate encodes the sizes attached to packed and odd-even dimensions. For each logical dimension, powers of two from 2 through the largest power no greater than the dimension size occupy a consecutive prefix, followed by the non-powers of two. This makes the vector-register-derived packing sizes that are typically explored in Section 3.2 consecutive while retaining codes for non-power-of-two sizes. When a layout contains multiple such dimensions, Morello combines their size codes using the Cantor pairing function $\pi(a, b) = \frac{(a+b)(a+b+1)}{2} + b$. Right-nested applications fold multiple size codes into a single coordinate. The following table illustrates this encoding:

Physical Dimensions	Size Codes	Coordinate
`p(4)`, `oe(8)`	$(1, 2)$	$\pi(1, 2) = 8$
`p(4)`, `oe(8)`, `p(16)`	$(1, 2, 3)$	$\pi(1, \pi(2, 3)) = 208$

Together, the tensor shape and the first coordinate determine how many size codes the decoder must recover and how to decode them; repeated unpairing reverses the nesting.

When tiling changes a tensor's shape without causing canonicalization to remove or merge physical dimensions or make a packing redundant, these first two layout coordinates remain invariant. The resulting sub-

Specs then differ along fewer database axes, increasing opportunities for compression.

The third and final coordinate represents contiguousness as the number of physical dimensions outside the contiguous suffix, $p - c$, where p is the number of physical dimensions and c is the contiguous-suffix length (Section 2.4.2). In a common case, tiling a logical dimension to size one causes canonicalization (Section 2.7) to remove one physical dimension from within the contiguous suffix without otherwise changing p or c. This removal decreases both by one, leaving the coordinate invariant. This stability avoids diagonal movement across the affected shape and contiguousness axes.

5.4. Evaluation of compression

To evaluate how effectively our spatial database representation compresses solutions, we computed databases of primitive Specs, each bounded by a power-of-two size through 4. For each database size, we synthesized Specs from a subset of Spec types and calculated the compression ratios of tables. We also measured each table's uncompressed storage size so we could track how compression changes as the table contents grow.

The results are plotted in Figure 13. Table-variant compression ratios range from approximately 2.14×10^3 to 8.79×10^6, with a median of 1.13×10^4. Computing these databases took 5.9 hours in aggregate. The complete size-4 database takes 128.9 MiB to store on disk with zstd compression; uncompressed, it would take 990.8 MiB. These measurements show that the complete plotted database remains small enough to store locally, while the per-table curves show that compression varies substantially by Spec type and rank.

5.5. Relaxing the power-of-two tiling restriction

The database described in this chapter and the search space described in Section 3.2.1 restrict tilings to powers of two. This restriction is useful because it keeps the dynamic program and memoization database simple: each shape dimension is mapped to a single coordinate by taking its base-2 logarithm, adjacent coordinates correspond to doubling or halving a dimension size, and vector registers typically have power-of-two sizes. However, the restriction is not fundamental to Morello; it is a point in the design space that trades expressiveness for space and time.

A natural generalization is to encode dimension sizes by their prime factorization. Instead of storing one coordinate per logical dimension,

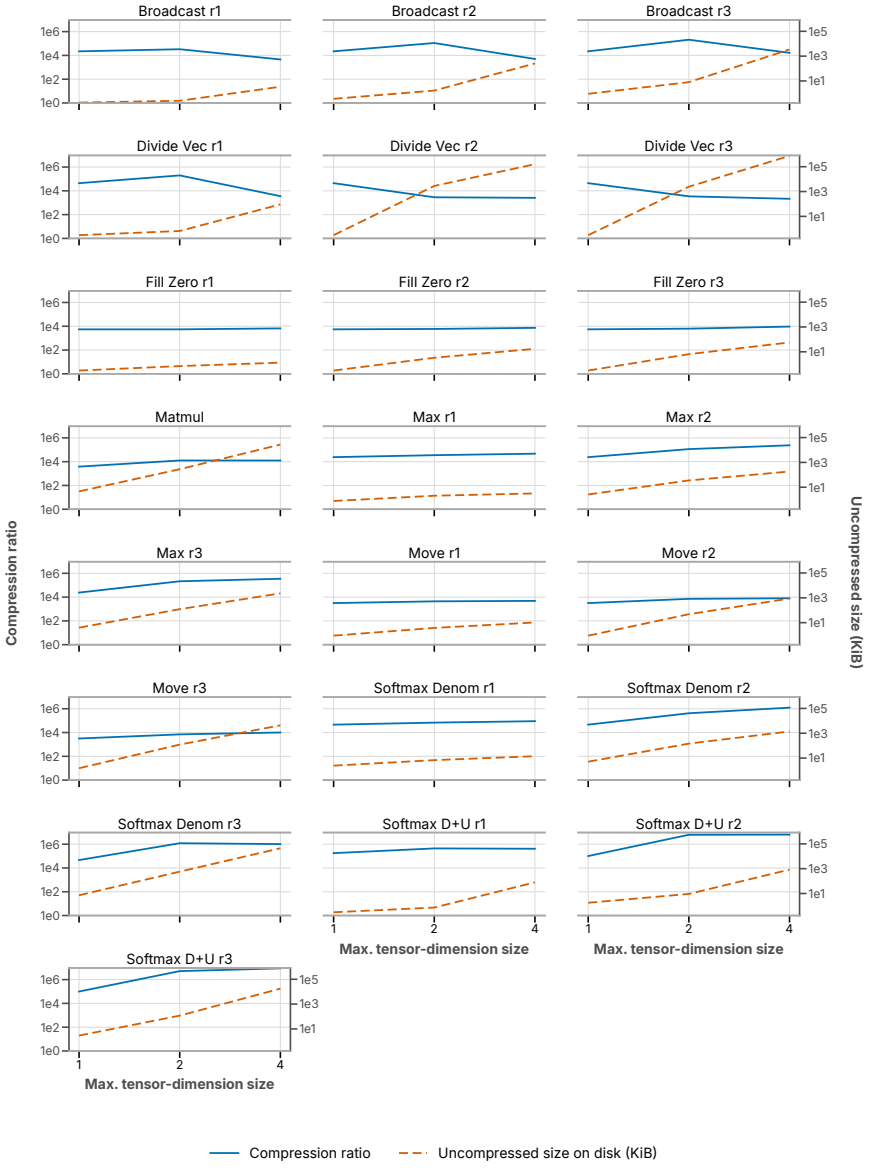


Figure 13. Compression ratios (number of Specs divided by the number of rectangles stored) and uncompressed storage sizes for precomputed database tables. The horizontal axis gives the maximum tensor-dimension size included, through 4. Solid blue lines show compression ratio on the left axis, and dashed orange lines show uncompressed table size on disk in KiB on the right axis.

the database fixes a shared prime basis and encodes dimensions by their exponents. For example, if a dimension size is restricted to $2^a 3^b$, then the database can represent the dimension with two coordinates, a and b. This keeps adjacent multiplicative sizes adjacent in the database: multiplying a tile size by 2 or by 3 shifts a coordinate by one.

This can matter because state-of-the-art kernel dimensions are not always powers of two, especially in boundary cases. The x86 SGEMMs from OpenBLAS [13] illustrate the point: the Haswell SGEMM's 12-column boundary uses 16×6 and 8×6 sub-kernels [15], while the Skylake-X SGEMM implementation's 24-column boundary uses 16×24, 8×24, 4×24, 2×24, and 1×24 kernels [14]. A power-of-two-only map can represent some of these dimensions, but it cannot represent 6, 12, or 24, which contain the factor 3. As shown in Chapter 4, Morello usually achieves good performance without searching such tile sizes, for example through Clang loop unrolling and hardware pipelining, but this may not be the case for all compiler-processor targets.

Proof-of-Concept. As a proof-of-concept, we implemented a restricted version of this factorized-coordinate representation in Morello. The proof-of-concept keeps the exponent of 2 and adds an exponent for the factor 3, restricted to 0 or 1, so it is able to represent tile sizes $3 \cdot 2^n$ and 2^n for any $n \in \mathbb{N}$. We evaluated it by computing all rank-2 `L1 Move` Specs through size 8, and the measured overhead was roughly proportional to the expected increase in search-space size: $\left(\frac{6}{4}\right)^2 = 2.25\times$. Wall-clock computation time increased by 1.85× and database space increased by 1.95× relative to the power-of-two baseline. The compression ratio was roughly 20% lower overall, though the total storage increase remained below the full 2.25× shape-count increase because canonicalization eliminates some Specs. This suggests that relaxing the power-of-two restriction is practical when the additional tile sizes are important enough to justify a roughly proportional increase in precomputation cost.

In practice, Morello implements the power-of-two database as well as the simpler fallback database described above. Because synthesis restricts tilings to powers of two, only a minority of Specs use the fallback map; any Spec dependency that appears after a tiling maps to the power-of-two database, as do its transitive dependencies.

6. RELATED WORK

In this chapter, we primarily compare Morello with systems that automatically optimize tensor programs for DNN inference (Section 6.1). Additionally, we briefly compare Morello's user-schedulable IR with its closest representational precedent, fireiron (Section 6.2), and its search procedure with selected dynamic-programming approaches in numerical code generation (Section 6.1.5) and database query optimization (Section 6.3).

6.1. AUTOMATIC OPTIMIZATION OF TENSOR PROGRAMS

Many systems share the goal of lowering tensor programs to high-performance machine code. They can be organized broadly into two categories: traditional compilers and auto-schedulers. The former category includes well-known compilers like xla [50], glow [56], PyTorch 2's TorchInductor [4], and tvm [11]. The latter category includes a broad class of systems known variously as auto-schedulers, auto-tuners, program synthesizers, or superoptimizers; this chapter refers to them simply as auto-schedulers.

Traditional compilers, which we will discuss only in their capacity as optimizers, are distinct from auto-schedulers in two major ways. First, as discussed in the introduction, compilers are designed for interactive use and therefore designed to work under a much tighter time budget than auto-schedulers. Second, the space of programs that forms the output domain of a compiler is harder to characterize since it is derived from the whole optimization pipeline, which is a composition of many interacting, non-associative, non-commutative optimizations, while the output domains of auto-schedulers can typically be described compactly.

6.1.1. TRADITIONAL COMPILERS

halide [54] is an influential domain-specific compiler that was originally designed for image processing pipelines but implements a general programming model—bounded functions over multi-dimensional arrays—which is also applicable to deep learning. In fact, later work extended halide with a relatively competitive x86 GEMM example [53], support for automatic differentiation [43], and, in 2019, an auto-scheduler which demonstrated speedups on on x86 ResNet-50 inference [1].

The compiler's key idea is to separate the algorithm, which is a target-independent, logical description of the program, from the 'schedule',

which is a target-specific, user-facing meta-program which controls the optimizations applied during lowering. While the separation itself was not novel—it was preceded by SEQUOIA [22]'s 'mappings' and CHILL's 'transformation scripts' [10], both of which are also examples of target-specific meta-programs exposed to the programmer—HALIDE's success is responsible for the contemporary popularity of the idea.

A HALIDE schedule's semantics differs from that of a MORELLO or CHILL schedule in that HALIDE's scheduling instructions configure the compiler's opaque lowering pipeline rather than rewrite a user-visible IR. This complicates re-targeting HALIDE, which is unfortunate given the rapid diversification of instruction set architectures (ISAs) and accelerators. In systems like MORELLO, CHILL, LIFT [62], and EXO [31], scheduling instructions rewrite within a user-facing IR, which simplifies re-targeting. For instance, the task of extending MORELLO with a new microkernel is scoped by a well-defined interface: it is a matter of adding a codelet-generating (lowering) function, defining a cost, and matching one or more SPEC patterns. In contrast, extending HALIDE requires arbitrary modifications to its internals.

Additionally, HALIDE does not expose a model of the target's memory hierarchy. Instead memory is controlled by scoping buffers to specific loop levels with an optional argument specifying placement into explicit memories which controls lowering (e.g., heap or stack, shared or texture memory). This obscures performance cliffs resulting from memory spills or thrashing. Systems such as MORELLO, FIREIRON, and SEQUOIA avoid this problem by requiring the programmer to map programs to the memory hierarchy. Unlike HALIDE, these systems benefit from statically known buffer sizes, which simplifies checking memory capacity constraints.

TVM [11] is a research compiler for tensor programs that builds on the ideas of—and, in fact, originally used code from—HALIDE [16]. It originally lowered code through a traditional pipeline: a model was transformed to a graph-level IR, optimized, lowered to a low-level per-operator IR, optimized again, and finally translated to machine code [55]. Later work on the RELAX [39] IR simplified interleaving of some graph- and operator-level rewrites and analyses, breaking the typical requirement that all computation-graph-level optimizations apply before all operator-level optimizations. This simplifies joint optimization. Still, TVM's lowering process remains a pipeline of heuristic passes where each phase is constrained by decisions made in earlier passes.

The TVM team has also produced two auto-schedulers, AutoTVM [12] and ANSOR [73], which are discussed in the following section.

MLIR [41] is a compiler infrastructure built around a novel IR which itself composes other IRs, called *dialects*, into a single, SSA-style program representations. MLIR is a core component of many tensor compiler stacks: StableHLO is an MLIR dialect that can be produced by TensorFlow and consumed by the XLA compiler [47, 48], modern TRITON IRs are MLIR dialects [38, 63], and IREE is an MLIR-based compiler and runtime [32], while Mojo uses MLIR as part of its compiler and runtime stack [42]. Like the RELAX IR—which was developed later—MLIR allows for the interleaving of graph- and operator-level rewrites and analyses.

MLIR's Linalg dialect is the closest point of comparison within MLIR. Like MORELLO and RELAX, it represents tensor computations as operations that can be decomposed through transformations such as tiling while preserving basic operation semantics; for example, a tiling a `linalg.matmul` produces a loop around a `linalg.matmul`, rather than becoming a loop nest of instructions with independent and unrelated semantics [41, 46]. This places Linalg closer to MORELLO than a more general IR. With the addition of a dialect to model cache behavior and nested specifications and the discipline to avoid rewrites of non-specification nodes, the MORELLO IR could have been implemented as one or a few MLIR dialects, though has few advantages.

6.1.2. HEURISTIC AUTO-SCHEDULERS

DNN auto-schedulers, unlike compilers, perform large and often time-consuming searches over spaces of program variants to find the variant with highest throughput, lowest latency, and/or lowest energy consumption.

Auto-schedulers are built on top of a compiler, augmenting or substituting the compiler's optimizer. They will generate (frontend) code for and then invoke a compiler such that they expect their optimization decisions to be preserved through the compiler's optimization pipeline—perhaps by generating pragmas or some other explicit schedule, or by assuming that the compiler will preserve certain features of the frontend code (e.g., loop order and memory layout)—or they are implemented as new optimization phases in the compiler itself. As a result, as with MORELLO, program costs are, in part, a function of the compiler.

Many of these systems treat optimization as a sequential decision-making problem over an action space of a scheduling language (like HALIDE's or TVM's) or other semantics-preserving rewrites. Most use a learned cost model to guide a heuristic search. Examples include the HALIDE auto-schedulers by Adams et al. [1] targeting small DNNs (ResNet-50 and

Conv+ReLU blocks) on x86 and Anderson et al.'s [3] targeting GPUs, in addition to the systems discussed below. While this section focuses on auto-schedulers whose primary architectures are built around heuristic search, other auto-scheduling systems that rely on exact mathematical formulations—such as polyhedral optimization and equality saturation—are discussed in subsequent sections.

AutoTVM [12] is an early auto-tuner for TVM that optimizes individual tensor operators by searching through implementations instantiated from templates. It explores the parameter spaces exposed by these templates (such as tile sizes, loop orders, and unroll factors), alternating between predicting performance with an online cost model and measuring runtimes on target hardware. Optimal implementations are indexed by exact workloads (e.g., operator shapes, data types, and data layouts) and hardware targets, meaning AutoTVM cannot reuse sub-problem schedules within a larger problem, as MORELLO does. Finally, manually designing AutoTVM templates remains labor-intensive: Zheng et al. [73] reported that "the code repository of TVM already contains more than 15K lines of code for [AutoTVM] templates."

ANSOR [73] is a subsequent TVM auto-scheduler that, unlike AutoTVM, is not dependent on manually constructed schedule templates. For each operator or fused sub-graph, ANSOR enumerates program sketches by recursively applying a small set of derivation rules. These sketches determine high-level program structure, including multi-level tiling, cache stages, reduction factorization, and fusion choices. ANSOR then randomly annotates the sketches to obtain complete programs and uses evolutionary search with a learned cost model to fine-tune low-level choices such as tile sizes, unroll factors, and parallel factors.

ANSOR demonstrates a practical consequence of heuristic auto-schedulers lacking optimality guarantees which we discussed in Chapter 1, it is not clear that performance plateaus as search runs longer. For example, ANSOR's evaluation of networks like ResNet-50 shows that runtime continues to improve even after tens of thousands of measurement trials, suggesting the search might continue finding improvements for an arbitrarily long time without any saturation (or optimality) guarantee. Furthermore, although ANSOR can share tuning information across repeated sub-graphs, the program it selects is tied to a concrete partitioned sub-graph, fixed shapes, and a hardware target; this selection does not establish that the same sub-program is optimal inside a different or larger computation.

FlexTensor [74] is another TVM auto-scheduler, developed concurrently. Like ANSOR, it generates a collection of programs by enumerating and heuristically pruning sequences of scheduling-primitive applications, then searches among those candidate programs with simulated annealing, using a Q-function to guide neighbor selection. The Q-function is updated online during optimization. FlexTensor minimizes measured candidate runtime when hardware measurement is practical and the output of an analytic model when it is not.

ROLLER [75] is another auto-scheduler whose key innovation is to design a search space containing only programs that satisfy a number of performance-engineering best practices. Concretely, it constructs only programs whose tiles align with execution-unit granularities, use hardware-aligned memory accesses, avoid memory-bank conflicts, add only limited padding for uneven tensor dimensions, and fit in their assigned memory. ROLLER then performs a greedy, heuristic search through this space of programs. It orders the search with a local data-reuse heuristic, favoring choices that save more memory traffic for each additional byte of storage they require. ROLLER then measures the first k complete programs found by this search and chooses the fastest.

Perhaps unsurprisingly, ROLLER's search is many orders of magnitude faster than that of AutoTVM or ANSOR. The resulting kernels are often competitive; most of ROLLER's generated kernels have runtimes within 10% of those generated by AutoTVM or ANSOR. However, ROLLER still falls outside the 10% band for a substantial minority of operators, mainly small operators or irregular tensor shapes. This illustrates the brittleness of heuristic search-space constraints: the same rules that make ROLLER fast can also exclude good implementations.

Steiner et al. [60] evaluate a value-learning approach for generating HALIDE schedules. They approximate the value function with a bidirectional LSTM trained to map a HALIDE program to the best runtime achievable by applying additional scheduling operators. The HALIDE program is represented by a sequence of vectors of hand-engineered features.

Steiner et al.'s value-learning formulation, like any value learning approach, could be recast over Morello-style specifications: rather than approximating the value of a concrete program or schedule prefix, it could approximate the value of a specification. This would make the learned state space follow the same sub-problem decomposition that Morello uses instead of the larger space of partial schedules.

MetaFlow [35] is an optimizer which searches over sequences of graph rewrites. Its key novelty is using a search algorithm which will tolerate a bounded reduction in a graph's analytical cost before backtracking. This allows it to discover some optimizations which are only profitable after 2 or more rewrites. Like other auto-schedulers, this is made feasible by first heuristically partitioning the computation graph.

TASO [34] is a DNN graph optimizer that contributes a method of automatically constructing a set of graph substitutions. It enumerates candidate computation graphs, identifies likely equivalences between graphs by testing, and then verifies that manually supplied operator properties entail functional equivalence of each candidate substitution's source and target graphs. These substitutions are then applied as part of a backtracking search that yields the lowest-cost graph discovered.

As a graph-level optimizer, many of TASO's optimizations are well out of MORELLO's scope, though some could be expressed as SPEC substitutions. For instance, the TASO paper advertises a transformation of two parallel matrix multiplications into one matrix multiplication that consumes a concatenated matrix. Expressing this would require the addition of, at least, a `Concat` primitive SPEC and a rewrite from one composed SPEC to another. The principle challenge would be implementing this as a system of rewrites that preserves acyclicity in the SPEC dependency graph.

Nevertheless, a TASO-like approach to generating rewrite rules for the MORELLO IR is appealing. It could be an efficient way to expand MORELLO's relatively small set of rewrites (recall Table I).

PET [68] is an unusually expressive optimizer. It can apply graph rewrites that are correct for only some of the tensor values computed by the graph, then introduce additional operations to "correct" the outputs for the remaining values. For instance, PET can combine independent convolution operations that differ in their number of output channels by padding inputs, fusing the operations, and adding a cleanup operation that removes the padded-output zeros. When these optimizations expose more parallelism or let the underlying tensor compiler use a different algorithm, such as Winograd convolution, their benefits can outweigh the overhead of the correction operations.

6.1.3. POLYHEDRAL OPTIMIZATION

Polyhedral optimization methods represent statement instances, such as loop iterations, as integer sets defined by affine constraints over loop indices. They represent data accesses as maps from statement instances

to array elements and dependencies as relations between statement instances. Compilers optimize loops by constructing an affine schedule: a relation that maps each statement instance to a multidimensional logical time. Those logical-time dimensions determine the generated loop nesting, but they need not correspond one-to-one to the source loops. Classic optimizations like fusion, tiling, or skewing are achieved by modifying this mapping. The compiler preserves original data dependencies by verifying that, under the new mapping, the source of every dependence lexicographically precedes its destination. The polyhedral model is unusually expressive in that it can naturally describe, for example, non-rectangular domains and skewed or wavefront execution orders.

Several tensor compilers leverage the polyhedral model to schedule the dense, static control flow typical of neural network kernels. Examples include CHILL [10], TIRAMISU [5], DIESEL [21], TENSOR COMPREHENSIONS [66], DECLARATIVE LOOP TACTICS [8], and AKG [72].

While expressive, polyhedral optimization can be relatively complex and slow. There are two main reasons for this. First, dependency analysis and integer-set simplification usually have exponential worst-case complexity [49, 51, 67]. Second, bridging the representation gap between an abstract syntax tree and a polyhedral representation is non-trivial. A polyhedral optimizer must compute a schedule and then "lower" the transformed polyhedra back into a concrete loop nest; if optimizing existing code, it must also first identify and "raise" static-control regions of the syntax tree into polyhedra [6].

In developing MORELLO, we have not needed the additional expressiveness of polyhedral methods. Much like Lattner et al. [41], we have found rectangular tiling and data packing—albeit with support for opaque swizzles such as odd-even packings—to be sufficient and to naturally expose a hierarchical substructure. A caveat is that we do not evaluate workloads that typically benefit from wavefront parallelism (i.e., skewing the iteration order), such as recurrent neural networks. While we expect this form of loop skewing and parallelism to be compatible with our IR by introducing additional rewrite rules, we have not verified this.

6.1.4. EQUALITY SATURATION

TENSAT [70] optimizes computation graphs with a novel application of equality saturation (built on EGG) [69]. It adapts the TASO [34] graph representation, graph substitutions, and cost model, but it replaces TASO's search algorithm with an approach that lowers a saturated e-graph to an ILP problem. Solving this problem extracts a cost-minimal (and acyclic)

computation graph, although the e-graph construction typically does not fully saturate.

Like Morello, tensat can compute minimal solutions to larger search problems by consolidating overlapping substructure (though that substructure is not optimal). However, Morello is not directly comparable to tensat because Morello's IR represents lower-level decisions, such as tile sizes, which are not typically represented in a computation graph.

In principle, equality saturation could be directly applied to Morello's IR and system of rewrites. Saturation would result in an e-graph where each e-class contains at least one Spec and zero or more partial programs in which nested Specs are parameters (i.e., references to other e-classes), and extraction could be done greedily. This is equivalent to materializing Morello's entire search tree ahead of time, which, unfortunately, would have intractable memory requirements: in addition to abandoning the benefits of Morello's spatial database, memory requirements would increase roughly proportional to the mean breadth of the search tree. This problem would need to be solved for the approach to be practical. Additionally, given that Morello's rewrite system is acyclic, there would be no performance benefits.

Equality saturation is likely a better fit for Spec rewriting, which is Morello's closest analog to the computation graph rewrites done by tensat. Currently, Morello does only a very small amount of Spec-level rewriting ('canonicalization' described in Section 2.7). If these Spec-level rewrites were to grow, it would quickly become more difficult to develop and maintain a confluent system. Constructing an e-graph could make this much simpler.

6.1.5. NUMERICAL CODE GENERATION

Automatically scheduling high-performance code using search and dynamic programming has a long history in the scientific computing community. Early numerical code generators successfully applied techniques such as algorithmic rewriting and cost-driven search that are now central to modern tensor compilers.

fftw [23, 24] in particular is an early and extremely influential system that demonstrated the value of automatically searching for, rather than hand-coding, implementations of numerical algorithms: in this case, discrete Fourier transforms. Its planner uses dynamic programming to select among plans built from recursive transform decompositions, data-packing choices, and fixed-size kernels: mechanisms roughly analogous to

Morello's tilings, data movements, and microkernels. Beyond having a smaller space of sub-problems (i.e., Specs), fftw is distinct from Morello in that its microkernels (called 'codelets') are themselves generated ahead of time (by `genfft`).

spiral [52] is a framework, in development since 1998, for automatically generating and scheduling fast implementations of numerical kernels, including those addressed by our work (e.g., matrix multiplication). Principally, it is a compiler that, as its first phase, searches for a program via a sequence of rewrites from an acyclic, terminating system.

As in Morello, optimization begins by specifying a program's semantics as an expression in a specification IR. In spiral's case, that is the *tagged Operator Language (tOL)*. A tOL expression can encode much of the information represented by a Morello Spec: operator parameters encode the operation and dimensions, operator structure encodes layouts, and tags or wrappers supply hardware context such as vector width, thread count, and, in specialized rule sets, memory capacity. Unlike Morello's Spec fields, these annotations need not all be present. Supplying an annotation enables and constrains the corresponding hardware-specific rules; if it is absent, those rules generally do not apply.

Rewrites are defined over the tOL language, and search visits sequences of rewrite applications, either until termination (no rewrites apply and no nonterminal expressions remain), or some search-algorithm-specific condition is met.

Among spiral's search algorithms is a dynamic-programming-based search, where normalized tOL formulae are the memoization table's keys and programs (with references to other entries) are the values. However, spiral's DP-based search differs from that of Morello in two important ways.

The first key difference is that some optimizations which are integrated into Morello's IR design, such as the decision to fuse two operators, are not represented in tOL. Like a traditional compiler, spiral implements a pipeline of deterministic lowering phases after search that perform optimizations such as fusing and choosing the order of loops and reordering load and store instructions.

The second key difference is that the dynamic programming algorithm computes costs through hardware benchmarking. Specifically, the optimal rewrite is chosen by benchmarking the entire program from that rewrite, selecting the sub-program that minimizes execution time. This is true both of leaf (terminal) and non-leaf (nonterminal) search states; the programs are composed from children according to cost, and then the composition

is benchmarked again. Runtimes are context-dependent: the implementation that benchmarks fastest for a sub-problem in isolation may not be fastest when embedded in a parent. By retaining only the former, SPIRAL may discard a component of the fastest complete program and therefore does not guarantee a global runtime optimum.

This context-dependency challenge could be partially addressed with an analytically derived cost model. SPIRAL, for instance, implements an alternative model which counts arithmetic operations, and dynamic programming is guaranteed to produce a global optimum with respect to that model. However, this introduces a phase-ordering problem: it is unclear how to integrate the costs of memory accesses into this model without representing cache accesses in the IR, and important determinants of cache behavior, such as whether loops are fused, are not available until later phases.

In contrast, MORELLO guarantees a global optimum with respect to its cost model but, of course, the cost model may be inaccurate with respect to the hardware being targeted. Its reduction is component-wise monotonic: replacing a child with a lower-cost implementation cannot increase the cost of its parent.

6.2. FIREIRON

The MORELLO IR is most directly inspired by Hagedorn et al.'s FIREIRON [29], which does not address synthesis but does explicitly represent memory movement and decomposes program specifications into smaller sub-specifications at each scheduling step. FIREIRON demonstrates that syntactic, top-down expansion of specifications into partial programs with nested sub-specifications is a simple and expressive way to schedule tensor programs for GPUs.

6.3. DATABASE QUERY OPTIMIZATION

Scheduling optimal plans using dynamic programming is not a new idea, especially in the databases community.

Relational query optimizers translate a logical query into a physical execution plan. The seminal SYSTEM R optimizer [57], the later VOLCANO optimizer [28], and VOLCANO's successor CASCADES [27] all synthesized execution plans using dynamic programming.

7. CONCLUSION

This dissertation explores a novel point in the design space of compilers and auto-schedulers: evaluation of every expressible implementation of a program in our IR. While the rapid combinatorial explosion in the number of implementations suggests that this approach is intractable, we have shown that practical synthesis of small operators is possible by exploiting optimal substructure and storing results in a database that can efficiently compact 'adjacent', same-value optima. We summarize the contributions of this dissertation in Section 7.1.

While synthesizing entire DNNs as large as VGG-19 remains prohibitive in terms of time and space, there are many opportunities to further improve the scalability of our approach, ranging from changes to the Spec language (e.g., our data layout abstraction is likely too precise) to substantial changes to the search algorithm (e.g., search could work directly on the database rectangles instead of individual Specs). These optimizations have the potential to increase the synthesis rate by one or two orders of magnitude. We elaborate on possible future directions in Section 7.2.

7.1. SUMMARY

Recall that DNN performance depends on many interacting choices, including fusion, tiling, memory placement, layout, data-type conversion, vectorization, and microkernel selection. These choices are dependent: fusing two pipeline stages can change the best tile shape or whether recomputation is worthwhile, and a microkernel choice may depend on a compatible memory layout and data movement schedule. These interactions make fixed phase orders and local search brittle, and motivate this dissertation's approach: exhaustive search of a joint space of implementations.

Three key ideas make this exhaustive search practical: an IR expressive enough to jointly represent important optimization decisions, a cost model and Spec-indexed dynamic program for optimizing those choices, and a spatial memoization database for storing the resulting optima compactly. We reified these ideas as a new compiler named Morello.

The rest of this section reviews the core contributions of this dissertation: these key ideas and an evaluation of Morello and its synthesized code.

IR & Scheduling. In Chapter 2, we introduced an expressive, hierarchical IR for tensor programs. A key feature of the IR, which was heavily inspired by that of fireiron [29], is that programs may contain Specs, and those Specs denote sub-programs still in need of implementation. This forms a natural interface for composing programs in the IR: if a program implements a Spec s and another program contains a nested application of Spec s, then the program can be substituted for the nested s (i.e., β-reduction). Scheduling applies rewrites that perform these substitutions, reducing until no nested Specs remain. This term rewriting system provides the interface for both users and our synthesizer to construct programs.

Synthesis. Chapter 3 defines the dynamic program used to search the space of implementations, treating Specs as sub-problems. Each scheduling rewrite of a Spec introduces a partial implementation containing zero or more nested dependent Specs. Synthesis proceeds by recursively solving these dependent Specs and selecting the cost-minimizing implementation. The cost model is defined to satisfy the optimal substructure property: the total cost of a candidate can be computed as the root node's cost combined with the optimal costs of its dependent Specs. The result is an exponentially smaller optimization problem that scales with the number of reachable Specs rather than, as is the case in most related work, the number of rewrite sequences (Section 3.3).

In addition to scaling to larger problems, a key usability benefit of exhaustive search is that it provides a diagnostic capability that heuristic auto-schedulers generally lack. Heuristic auto-schedulers can usually be run longer in the hope of finding a better schedule; there is no guaranteed termination condition for optimality. In contrast, because our approach is exhaustive, an unsatisfactory result has a clear explanation: either the desired implementation is outside the search space, or the cost model is inaccurate. This greatly improves a performance engineer's ability to reason about the results.

Since tiling, memory placement, packing, widening, pipeline decomposition, and microkernel selection are all optimization decisions with corresponding rewrite rules in Morello, synthesis explores all combinations of those optimizations and can compare their combined effects. The cost of, for instance, selecting a microkernel is evaluated together with that of any data movement needed to feed it.

As we showed in Section 3.3, our approach to synthesis scales well across large tensor dimensions and small pipelines. Increasing an existing dimension's size increases asymptotic complexity by only a polylogarith-

mic factor. Pipeline length contributes a factor equal to the number of unique sub-pipelines (at most quadratic), plus a linear split-enumeration factor. Increasing the number of Spec dimensions or adding modeled memories presents a greater scalability challenge. Adding a new dimension is asymptotically more expensive than scaling an existing one: scaling an existing dimension merely increases its associated logarithmic factor, whereas a new dimension introduces an additional multiplicative polylogarithmic factor. New dimensions are often a consequence of increasing pipeline length, as new inputs can introduce new Spec dimensions. Adding a modeled memory (e.g., L2 or L3 caches) is more expensive than increasing an existing memory's capacity. Doing so increases total synthesis cost polynomially with respect to the number of modeled memories—with the degree determined by the number of tensor parameters—and introduces an additional memory-limit axis. Conversely, increasing a memory's capacity only expands its existing limit axis: linearly for register files, and logarithmically for caches and main memory. The range-filling optimization described in Section 3.2.2 mitigates this overhead by reusing an optimal solution across the component-wise range between an implementation's peak memory footprint and the queried limits.

Evaluation of Synthesized Code. Chapter 4 shows that Morello, with minimal manual scheduling, generates competitive implementations of matrix multiplication and softmax on both AVX2 Zen 1 and AVX-512 Zen 5 systems. Morello's float32 matrix-multiplication implementations approach the performance of OpenBLAS and often outperform AOCL and Intel MKL. Its softmax implementations are competitive with XNNPACK on Zen 1 and in some Zen 5 configurations, and they broadly outperform PyTorch. Ultimately, these results confirm that Morello's search space is expressive enough and its cost model accurate enough to generate practically useful implementations of important kernels.

Spatial Database Representation. Finally, while Morello's synthesis algorithm eliminates the combinatorial explosion of rewrite sequences typical of auto-schedulers, representing every sub-problem explicitly requires an impractical amount of storage. Chapter 5 introduced a novel memoization database representation that maps Specs to integer coordinates in high-dimensional tables and stores adjacent identical decisions as hyperrectangles. This representation exploits the regularity of optimal decisions across nearby tensor shapes and memory limits: many adjacent Specs share the same decision, similar cost intensity, and similar memory behavior. The result is an order-of-magnitude reduction in storage and

a corresponding increase in the size of problems that can be solved. In addition, a precomputed database is not just a per-run cache; it is a compact artifact that can be reused by later searches or even shipped with the compiler.

Taken together, these pieces demonstrate that a tensor compiler can search a much richer implementation space while still retaining exact optimality with respect to a stated cost model. The key is to decompose programs into reusable Specs, define a cost model with optimal substructure over those Specs, and represent the memoization table in a way that takes advantage of the optimization problem's regularity. With that structure, a complete search can be applied to implementation spaces that jointly include fusion, tiling, memory movement, layout, SIMD, and microkernel choices.

7.2. FUTURE WORK

While there are opportunities to improve Morello's expressiveness and programmability,[20] the most important opportunities for future work are those that further improve the scalability of synthesis and the accuracy of its cost model. In particular, the cost model could be extended to model latency hiding, and the synthesis algorithm could take better advantage of the underlying data representation (rectangles).

[20] As discussed in Section 2.6, it would be useful to replace the `Compose` with a more general means of representing graphs of primitive Specs. It would also be useful to provide additional, user-facing control of synthesis spaces beyond scheduling a partial implementation manually before invoking synthesis, as described in Section 2.8.1.

7.2.1. LATENCY HIDING AND MULTI-RESOURCE COST MODELING

A limitation of Morello's scalar, additive cost model is that it does not explicitly represent how work from different program nodes overlaps on a superscalar, out-of-order processor. Consequently, it cannot compositionally model independent instructions executing while a load is pending, prefetches overlapping later computation, or contention among execution units and memory bandwidth. The same limitation arises when estimating asynchronous accelerator offloading, where host computation, data transfers, and accelerator execution may overlap.

Morello captures some of these effects locally through the opaque scalar costs assigned to microkernels. On x86, these costs summarize instruction dependencies, out-of-order execution, and processor-resource pressure within each microkernel. When a microkernel accesses an L1-resident tensor through a memory operand, that access is integrated into this local estimate. However, a scalar microkernel cost cannot represent overlap or interference across adjacent program nodes.

Future work could associate each implementation with a vector of costs, such as critical-path latency, execution-unit occupancy, and memory bandwidth demand at each level of the memory hierarchy. Synthesis could retain the Pareto frontier of non-dominated implementations, preserving those that trade pressure among different resources. Such an extension would increase synthesis time and database size, but the magnitude of the increase is difficult to predict. For a Spec with N feasible implementations, a Pareto frontier contains N implementations in the worst case; all N candidates can be mutually non-dominating. Whether frontiers remain small in practice—and whether they become small as relaxation factor ε is increased—is best answered empirically.

7.2.2. RECTANGLE-/DATA-DRIVEN SEARCH

Second, while Morello compresses its memoization database, the search algorithm is not designed to take advantage of this spatial representation. Rewrite dependencies and goal Specs are visited individually, whether or not those goal or dependency Specs map to the same underlying rectangle. A more efficient algorithm would visit each rectangle a small number of times, rather than repeatedly visiting each point inside that rectangle.

Batching Dependencies. To achieve this, future work could represent a set of scheduling rewrites with a pair of functions. The first would map an individual goal Spec to rectangles of dependency Specs. Together, those rectangles would cover the dependencies of every candidate rewrite in the set. A spatial join would then find the stored database rectangles that intersect those dependency rectangles. The second function would map those intersections back to the cost-minimal rewrite for the goal Spec. For example, the first function for tiling could map a matrix-multiplication Spec to a rectangle covering all valid smaller matrix-multiplication Specs. Given the intersections containing solutions for those smaller Specs, the second function could evaluate the corresponding tiling rewrites and select the one with the lowest cost. (A challenge here will be preserving simple dependency geometry; canonicalization, especially of layouts, could require many rectangles or more complex geometry to cover the correct dependencies.)

Batching Goals. The preceding synthesizer could also batch goal Specs. Instead of operating on a single goal Spec at a time, a batched interface would accept a rectangle describing many goal Specs. For example, it could accept matrix multiplications that vary only in their shapes and materialize all tiling dependencies in one step. This could amortize the work of

computing and resolving rewrite dependencies over multiple goal Specs: rather than rediscovering heavily overlapping dependency sets for each point in the rectangle, the synthesizer could compute dependency rectangles once and reuse them across the whole batch. The interface would also let synthesis pass misses from the preceding spatial join—the dependency rectangles left uncovered by stored database rectangles—directly into the functions that accept rectangles of goal Specs, rather than immediately lowering each miss to individual Specs.

Supporting Sparse Databases. An additional benefit of resolving dependencies with a spatial join is that it enables the efficient use of sparse databases: if a dependency rectangle intersects a database with holes, the join naturally returns only the stored solutions for dependencies present in the database. If synthesis treats the missing dependencies as unavailable, this would enable it to take advantage of arbitrary pruning of the database, sacrificing completeness and optimality to scale synthesis beyond what can be stored.

7.3. Use of AI Coding Assistants in Development

We developed Morello largely by hand, without the assistance of LLMs. Anecdotally, we found that coding assistants powered by models such as OpenAI's GPT-5.5 struggled to generate correct code in this codebase and consequently did not improve our productivity.

In particular, we used OpenAI's Codex, configured with GPT-5.5 and "Extra High" reasoning, to implement microkernels, including both their code generators and cost models. We also equipped the coding assistant with a detailed skill that described the components to implement, explained how to use llvm-mca to validate the code generator and cost model, provided an annotated example of a well-implemented microkernel, described the testing process, and identified common failure modes to avoid. Because many of our microkernels correspond to common, well-understood patterns in high-performance computing—particularly those found in BLAS implementations—we expected this approach to work well.

Unfortunately, several problems occurred frequently, despite being explicitly discussed in the skill:

- Generated microkernels were usually too broad, typically subsuming the functionality of one or more other microkernels as well as numerous special cases.
- Microkernels were generalized to support such a wide range of shapes that it implemented, within the microkernel code gen-

erators, tiling transformations nearly as general as Morello's general-purpose tiling rewrites.

- Memory pressure and residency were frequently conflated with the requirements of the throughput cost model. The generated code often introduced penalties intended to account for spills, even though memory was modeled separately in a way that made spills impossible.
- The agent was largely unable to reason through Clang's lowering to machine code. It usually designed the peak-memory and throughput models around a literal and unoptimized interpretation of the C code. For example, it rarely accounted correctly for the live registers introduced by the evaluation of nested expressions.

Perhaps future LLM-based coding assistants will be more helpful.

REFERENCES

[1] Adams, A. et al. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Transactions on Graphics*. 38, 4 (July 2019). https://doi.org/10.1145/3306346.3322967.

[2] Aminabadi, R.Y. et al. 2022. DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis* (2022), 1–15. https://doi.org/10.1109/SC41404.2022.00051.

[3] Anderson, L. et al. 2021. Efficient Automatic Scheduling of Imaging and Vision Pipelines for the GPU. *Proceedings of the ACM on Programming Languages*. 5, (Oct. 2021), 1–28. https://doi.org/10.1145/3485486.

[4] Ansel, J. et al. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (2024), 929–947. https://doi.org/10.1145/3620665.3640366.

[5] Baghdadi, R. et al. 2018. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. *https://arxiv.org/abs/1804.10694*.

[6] Bastoul, C. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques* (USA, 2004), 7–16.

[7] Beckmann, N. et al. 1990. The R*-tree: an efficient and robust access method for points and rectangles. *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data* (Atlantic City, New Jersey, USA, 1990), 322–331. https://doi.org/10.1145/93597.98741.

[8] Chelini, L. et al. 2019. Declarative Loop Tactics for Domain-specific Optimization. *ACM Transactions on Architecture and Code Optimization*. 16, 4 (Dec. 2019). https://doi.org/10.1145/3372266.

[9] Chellapilla, K. et al. 2006. High Performance Convolutional Neural Networks for Document Processing. *Tenth International Workshop on Frontiers in Handwriting Recognition* (Oct. 2006).

[10] Chen, C. et al. 2008. *CHiLL: A Framework for Composing High-Level Loop Transformations*. Technical Report #08–897.

[11] Chen, T. et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), 578–594.

[12] Chen, T. et al. 2018. Learning to Optimize Tensor Programs. *Advances in Neural Information Processing Systems* (2018).

[13] Contributors, O. OpenBLAS. *https://www.openblas.net/*. Accessed: 2025-01-12.

[14] Contributors, O. OpenBLAS's sgemm_kernel_16x4_skylakex_3.c. *https://github.com/OpenMathLib/OpenBLAS/blob/a47b45d4ebb020b61a269271556e60b6d6f0461d/kernel/x86_64/sgemm_kernel_16x4_skylakex_3.c#L451-L515*. Accessed: 2026-05-14.

[15] Contributors, O. OpenBLAS's sgemm_kernel_8x4_haswell_2.c. *https://github.com/OpenMathLib/OpenBLAS/blob/a47b45d4ebb020b61a269271556e60b6d6f0461d/kernel/x86_64/sgemm_kernel_8x4_haswell_2.c#L292-L421*. Accessed: 2026-05-14.

[16] Contributors, T. 2020. dmlc/HalideIR Git Repository. *https://github.com/dmlc/HalideIR*. Accessed: 2025-01-23.

[17] Contributors, X. XNNPACK. *https://github.com/google/XNNPACK*. Accessed: 2026-07-10.

[18] Cottier, B. et al. 2024. The rising costs of training frontier AI models. *https://arxiv.org/abs/2405.21015*.

[19] Dao, T. et al. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *Advances in Neural Information Processing Systems* (2022), 16344–16359.

[20] Dielissen, V.J. and Kaldewaij, A. 1991. Rectangular Partition Is Polynomial in Two Dimensions but NP-Complete in Three. *Information Processing Letters*. 38, 1 (1991), 1–6. https://doi.org/https://doi.org/10.1016/0020-0190(91)90207-X.

[21] Elango, V. et al. 2018. Diesel: DSL for linear algebra and neural net computations on GPUs. *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Philadelphia, PA, USA, 2018), 42–51. https://doi.org/10.1145/3211346.3211354.

[22] Fatahalian, K. et al. 2006. Sequoia: Programming the Memory Hierarchy. *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (2006).

[23] Frigo, M. and Johnson, S.G. 1998. FFTW: an adaptive software architecture for the FFT. *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing* (1998), 1381–1384. https://doi.org/10.1109/ICASSP.1998.681704.

[24] Frigo, M. and Johnson, S.G. 2005. The Design and Implementation of FFTW3. *Proceedings of the IEEE*. 93, 2 (2005), 216–231. https://doi.org/10.1109/JPROC.2004.840301.

[25] Georganas, E. et al. 2018. Anatomy of high-performance deep learning convolutions on SIMD architectures. *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas, 2018).

[26] Goto, K. and Geijn, R.A. van de. 2008. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*. 34, (2008), 12:1–12:25.

[27] Graefe, G. 1995. The Cascades Framework for Query Optimization. *IEEE Bulletin of the Technical Committee on Data Engineering*. 18, (1995), 19–29.

[28] Graefe, G. and McKenna, W.J. 1993. The Volcano optimizer generator: extensibility and efficient search. *Proceedings of IEEE 9th International Conference on Data Engineering* (1993), 209–218. https://doi.org/10.1109/ICDE.1993.344061.

[29] Hagedorn, B. et al. 2020. Fireiron: A Data-Movement-Aware Scheduling Language for GPUs. *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event, GA, USA, 2020), 71–82. https://doi.org/10.1145/3410463.3414632.

[30] Henighan, T. et al. 2020. Scaling Laws for Autoregressive Generative Modeling. *https://arxiv.org/abs/2010.14701*.

[31] Ikarashi, Y. et al. 2022. Exocompilation for productive programming of hardware accelerators. *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA, 2022), 703–718. https://doi.org/10.1145/3519939.3523446.

[32] The IREE Authors. IREE: An MLIR-based Compiler and Runtime for Machine Learning Models. *https://iree.dev/*. Accessed: 2026-05-20.

[33] Ivanov, A. et al. 2021. Data Movement Is All You Need: A Case Study on Optimizing Transformers. *Proceedings of Machine Learning and Systems* (2021).

[34] Jia, Z. et al. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada, 2019), 47–62. https://doi.org/10.1145/3341301.3359630.

[35] Jia, Z. et al. 2019. Optimizing DNN Computation with Relaxed Graph Substitutions. *Proceedings of the Second Conference on Machine Learning and Systems* (2019).

[36] Kaufman, S.J. 2024. Factor out deinterleaving of bf16 vectors for MatVecs. *https://github.com/google/gemma.cpp/pull/166*. Accessed: 2026-07-17.

[37] Kim, J. et al. 2025. The Cost of Dynamic Reasoning: Demystifying AI Agents and Test-Time Scaling from an AI Infrastructure Perspective. *https://arxiv.org/abs/2506.04301*.

[38] Kokkila-Schumacher, S. et al. 2024. Triton Kernel Compilation Stages. *https://pytorch.org/blog/triton-kernel-compilation-stages/*. Accessed: 2026-05-20.

[39] Lai, R. et al. 2025. Relax: Composable Abstractions for End-to-End Dynamic Machine Learning. *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (New York, NY, USA, 2025), 998–1013. https://doi.org/10.1145/3676641.3716249.

[40] Lamprakos, C. et al. 2025. Futureproof Static Memory Planning. *https://arxiv.org/abs/2504.04874*.

[41] Lattner, C. et al. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2021), 2–14. https://doi.org/10.1109/CGO51591.2021.9370308.

[42] Lattner, C. et al. 2023. A Unified, Extensible Platform to Superpower Your AI. *https://www.modular.com/blog/a-unified-extensible-platform-to-superpower-your-ai*. Accessed: 2026-05-20.

[43] Li, T.-M. et al. 2018. Differentiable programming for image processing and deep learning in Halide. *ACM Transactions on Graphics*. 37, 4 (July 2018). https://doi.org/10.1145/3197517.3201383.

[44] Milakov, M. and Gimelshein, N. 2018. Online normalizer calculation for softmax. *https://arxiv.org/abs/1805.02867*.

[45] MLIR. Bufferization. *https://mlir.llvm.org/docs/Bufferization/*. Accessed: 2026-04-12.

[46] MLIR. Linalg Dialect Rationale: The Case For Compiler-Friendly Custom Operations. *https://mlir.llvm.org/docs/Rationale/RationaleLinalgDialect/*. Accessed: 2026-05-18.

[47] OpenXLA Project. StableHLO Roadmap. *https://openxla.org/stablehlo/roadmap*. Accessed: 2026-05-20.

[48] OpenXLA Project. XLA Architecture. *https://openxla.org/xla/architecture*. Accessed: 2026-05-20.

[49] Pitchanathan, A. et al. 2021. FPL: fast Presburger arithmetic through transprecision. *Proc. ACM Program. Lang.* 5, OOPSLA (Oct. 2021). https://doi.org/10.1145/3485539.

[50] Project, O. XLA. *https://openxla.org/xla*. Accessed: 2026-07-17.

[51] Pugh, W. 1991. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing* (Albuquerque, New Mexico, USA, 1991), 4–13. https://doi.org/10.1145/125826.125848.

[52] Püschel, M. et al. 2005. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE*. 93, 2 (2005), 232–275. https://doi.org/10.1109/JPROC.2004.840306.

[53] Ragan-Kelley, J. et al. 2015. Halide CVPR 2015 Tutorial. *http://halide-lang.org/cvpr2015.html*. Accessed: 2025-01-23.

[54] Ragan-Kelley, J. et al. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *ACM SIGPLAN Notices*. 48, 6 (June 2013), 519–530. https://doi.org/10.1145/2499370.2462176.

[55] Roesch, J. et al. 2018. Relay: A New IR for Machine Learning Frameworks. *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Philadelphia, PA, USA, 2018), 58–68. https://doi.org/10.1145/3211346.3211348.

[56] Rotem, N. et al. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. *https://arxiv.org/abs/1805.00907*.

[57] Selinger, P.G. et al. 1979. Access Path Selection in a Relational Database Management System. *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts, 1979), 23–34. https://doi.org/10.1145/582095.582099.

[58] Sevilla, J. et al. 2022. Compute Trends Across Three Eras of Machine Learning. *2022 International Joint Conference on Neural Networks (IJCNN)* (July 2022), 1–8. https://doi.org/10.1109/IJCNN55064.2022.9891914.

[59] Shaikhha, A. et al. 2017. Destination-passing style for efficient memory management. *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing* (Oxford, UK, 2017), 12–23. https://doi.org/10.1145/3122948.3122949.

[60] Steiner, B. et al. 2021. Value Learning for Throughput Optimization of Deep Neural Networks. *Proceedings of Machine Learning and Systems* (2021).

[61] Steiner, B. et al. 2022. OLLA: Optimizing the Lifetime and Location of Arrays to Reduce the Memory Usage of Neural Networks. *https://arxiv.org/abs/2210.12924*.

[62] Steuwer, M. et al. 2015. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada, 2015). https://doi.org/10.1145/2784731.2784754.

[63] Tillet, P. 2022. Merge triton-mlir Branch—Complete Rewrite of the Backend from Scratch. *https://github.com/triton-lang/triton/pull/1004*. Accessed: 2026-05-20.

[64] Tillet, P. et al. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (2019), 10–19. https://doi.org/10.1145/3315508.3329973.

[65] Tollenaere, N. et al. 2023. Autotuning Convolutions Is Easier Than You Think. *ACM Transactions on Architecture and Code Optimization*. 20, 2 (Mar. 2023). https://doi.org/10.1145/3570641.

[66] Vasilache, N. et al. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *https://arxiv.org/abs/1802.04730*.

[67] Verdoolaege, S. 2010. isl: An Integer Set Library for the Polyhedral Model. *Mathematical Software – ICMS 2010* (Berlin, Heidelberg, 2010), 299–302.

[68] Wang, H. et al. 2021. PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections. *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)* (July 2021), 37–54.

[69] Willsey, M. et al. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, (Jan. 2021). https://doi.org/10.1145/3434304.

[70] Yang, Y. et al. 2021. Equality Saturation for Tensor Graph Superoptimization. *Proceedings of Machine Learning and Systems* (2021), 255–268.

[71] Van Zee, F.G. and Geijn, R.A. van de. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Transactions on Mathematical Software*. 41, 3 (June 2015). https://doi.org/10.1145/2764454.

[72] Zhao, J. et al. 2021. AKG: Automatic Kernel Generation for Neural Processing Units Using Polyhedral Transformations. *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada, 2021), 1233–1248. https://doi.org/10.1145/3453483.3454106.

[73] Zheng, L. et al. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation* (Nov. 2020), 863–879.

[74] Zheng, S. et al. 2020. FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System. *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2020), 859–873.

[75] Zhu, H. et al. 2022. ROLLER: Fast and Efficient Tensor Compilation for Deep Learning. *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA, July 2022), 233–248.