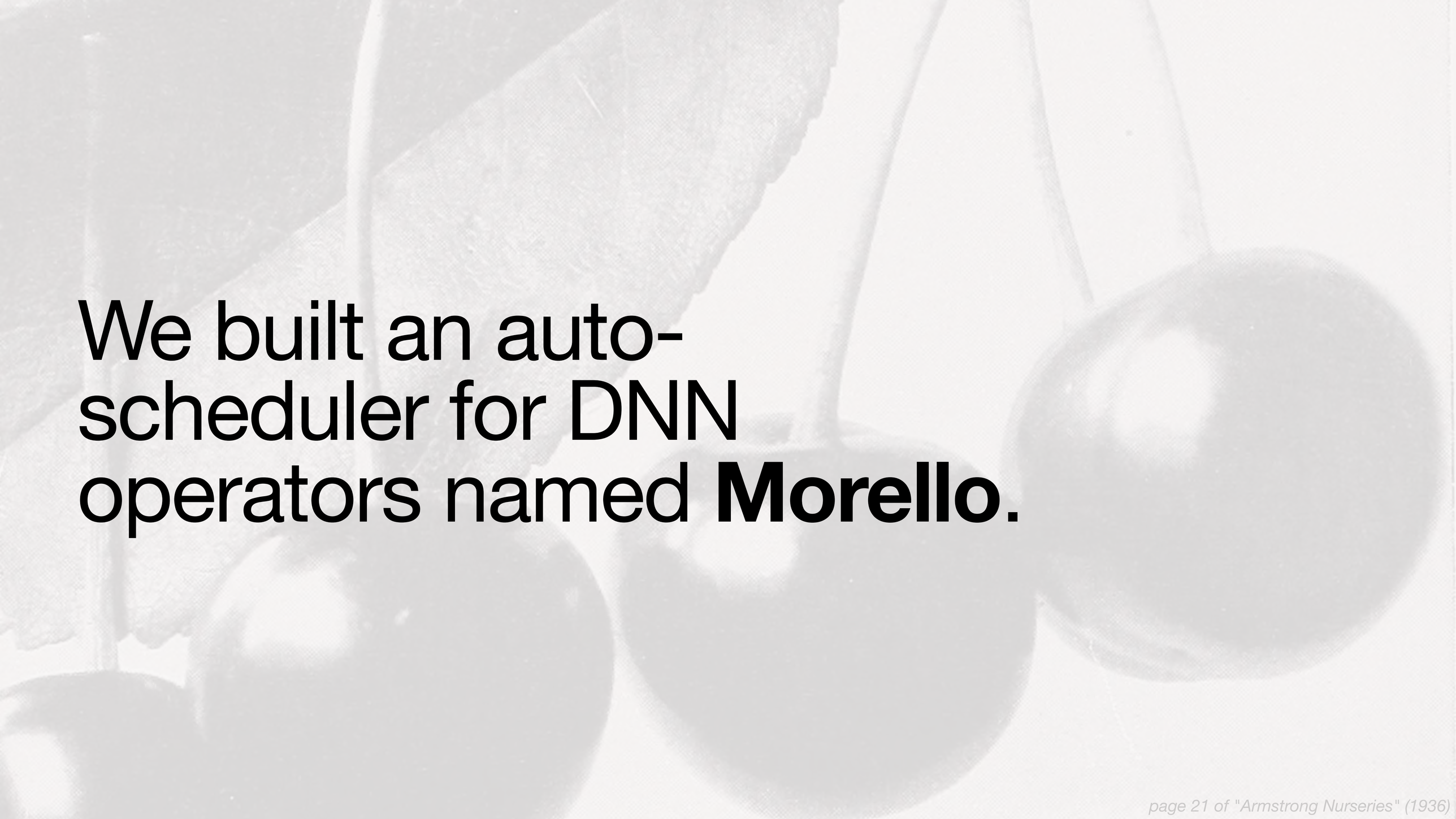
Joint Optimization of Deep Neural Networks with Dynamic Programming and Spatial Compression

Sam Kaufman · Final Exam · 17 August 2026



We built an auto-scheduler for DNN operators named **Morello**.

Auto-Schedulers

Specification
(e.g., a matmul
of some size)

Auto-Scheduler

Optimized Program

Broadly speaking…

- Inherit an IR from their underlying compiler (TVM, XLA, …)
- Search **stochastically** for a sequence of rewrites.
  - Guide search with heuristic.
    - Usually a combination of model learned from previous jobs and real benchmarking.

Set of Specifications

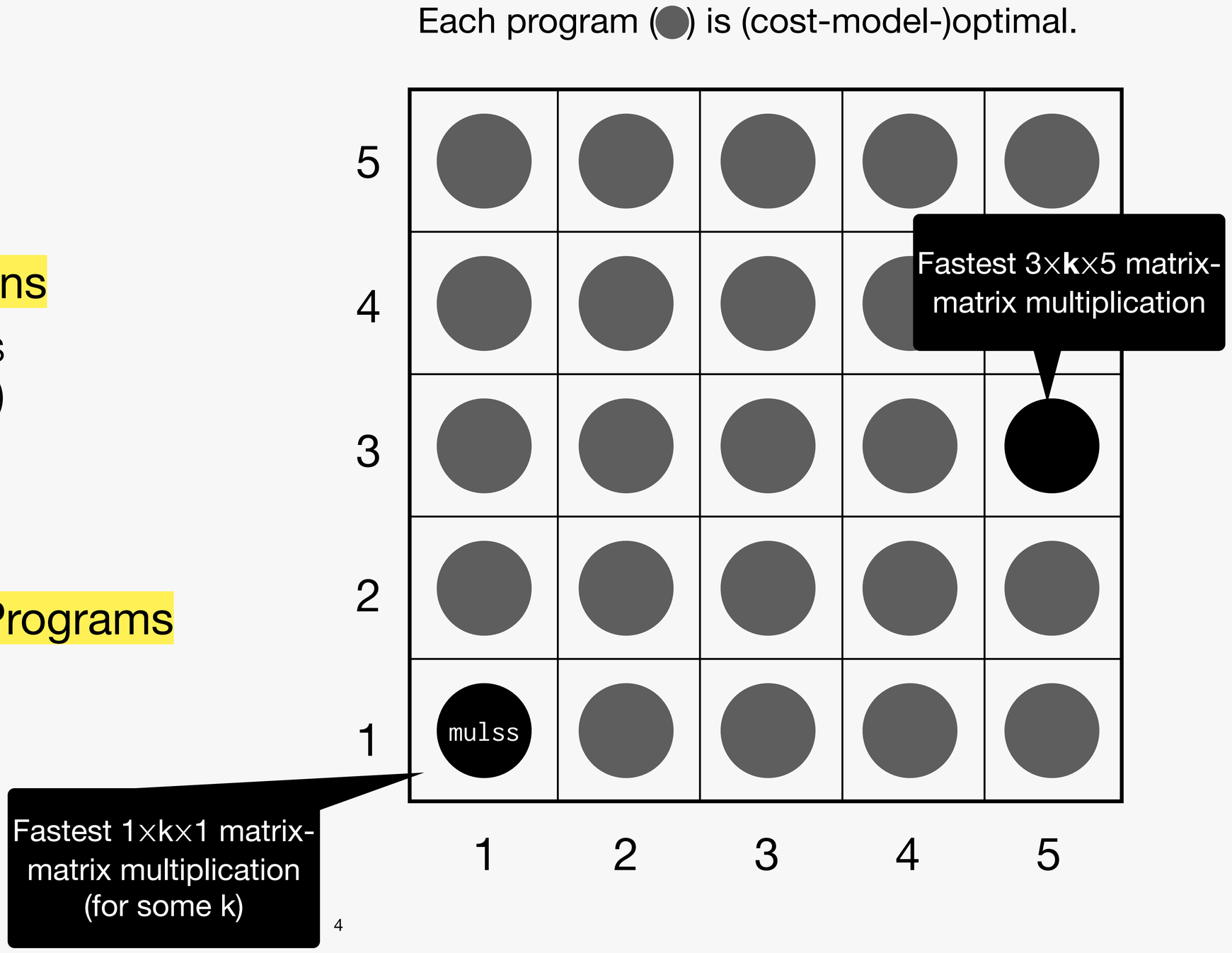
(e.g., all matmuls through size 512)

|

Morello

↓

Database of Optimized Programs

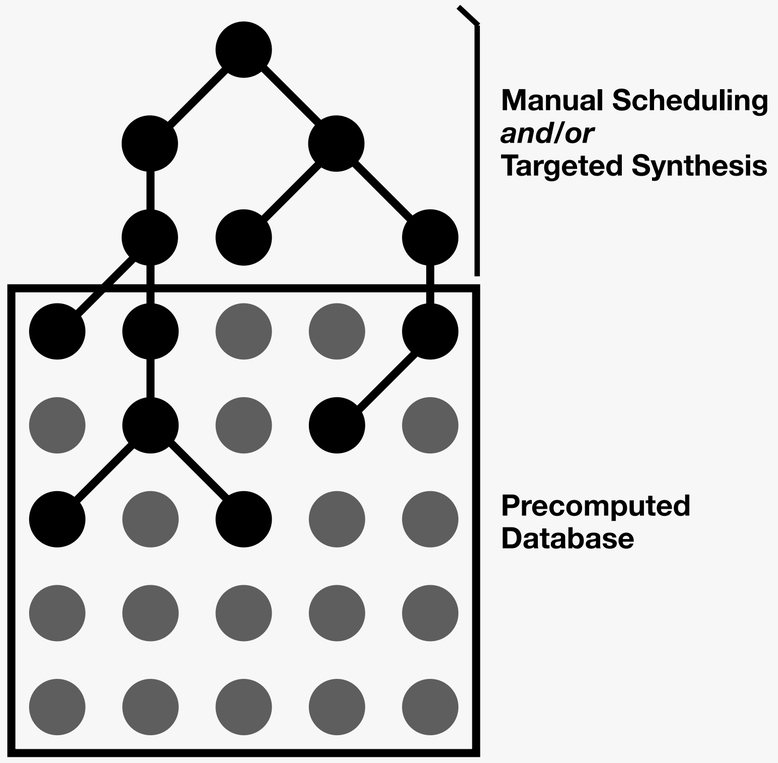


Benefits

Compute once, store on a laptop or data center

Immediately reference fast implementations (like a library!)

- Each implementation (●) is runnable.
- Can use as a base to accelerate subsequent auto-scheduling



Demo

github.com/samkaufman/morello

Key Challenge: Tables are high-dimensional and large.

Specifications (table keys)
have many features.

Many tables
One for each operator type–arity

30+ dimensions

b	m	k	n	data type of matrix A	layout of matrix A	resident memory of A	…	accumulate into output?	single thread only?	L1 capacity	…	Implementation
1	1	1	1	f32	row-major	L1 cache	..	no	yes	4 lines	..	mulss
..	1	1	2	..	..	..	..	..	..	..	..	
..	2	1	2	..	..	..	..	..	..	..	..	
..	2	2	2	..	col-major	registers	..	..	..	..	..	

2×10^{14} entries for 2048-square matmul (pow2 only)

Sub-Challenges

Programs must be fast

Answer: One IR exposing all critical optimization decisions; and search **exhaustively.**

Compute implementations quickly

Many implementations.

Answer: Develop a cost model with optimal substructure. This shares work between implementations.

Store implementations efficiently

At ~10 bytes per table entry, 2x10^14 entries is ~1.8 PiB for a matmul!

Answer: Compress matching optima using a spatial database.

1. IR & Cost Model

2. Search

3. Spatial Database

4. Evaluation

1. IR & Cost Model

2. Search

3. Spatial Database

4. Evaluation

Hierarchical Tiling

Example

```
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);
      }
    }
  }}
}
```

Specifications / Table Keys

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)

Built around hierarchical tiling.

Decompose via:

- tiling,
- data movement,

to smaller, faster sub-programs.

Hierarchical Tiling

Example

```
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);
      }
    }
  }}
}
```

Specifications / Table Keys

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)

Built around hierarchical tiling.

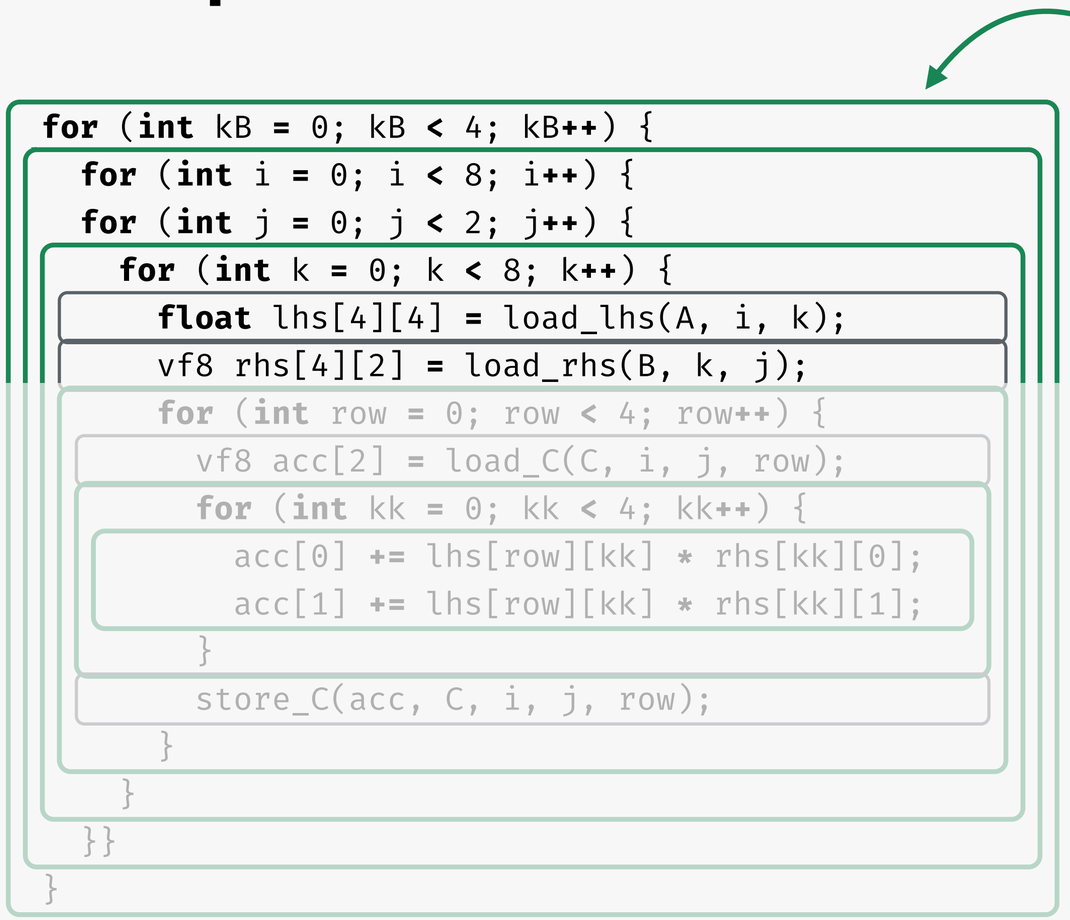
Decompose via:

- tiling,
- data movement,

to smaller, faster sub-programs.

Hierarchical Tiling

Example



```
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);
      }
    }
  }}
}
```

Specifications / Table Keys

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)

Built around hierarchical tiling.

Decompose via:

- tiling,
- data movement,

to smaller, faster sub-programs.

Hierarchical Tiling

Example

```
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);
      }
    }
  }}
}
```

Specifications / Table Keys

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)

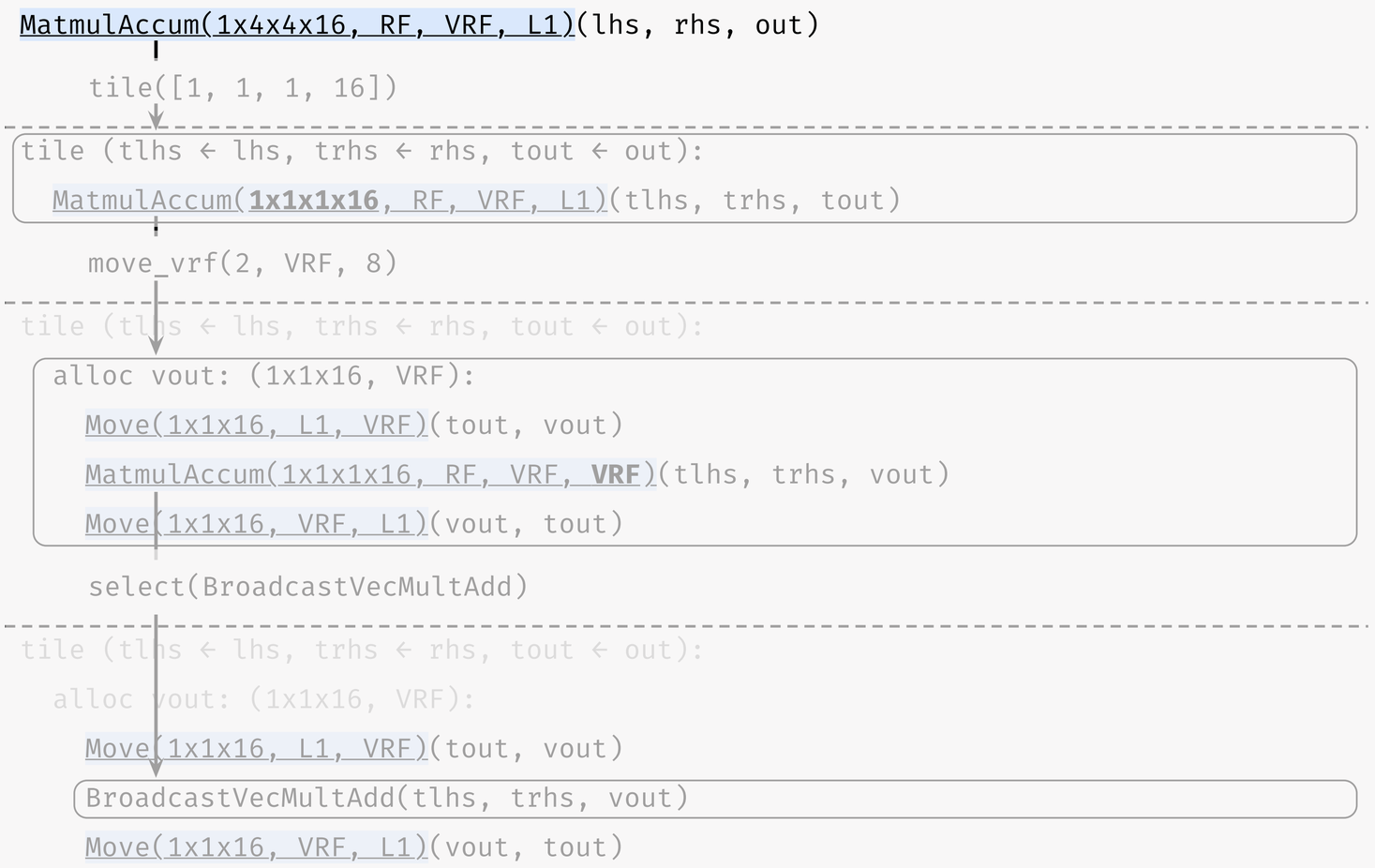
Built around hierarchical tiling.

Decompose via:

- tiling,
- data movement,

to smaller, faster sub-programs.

Building a Morello Implementation

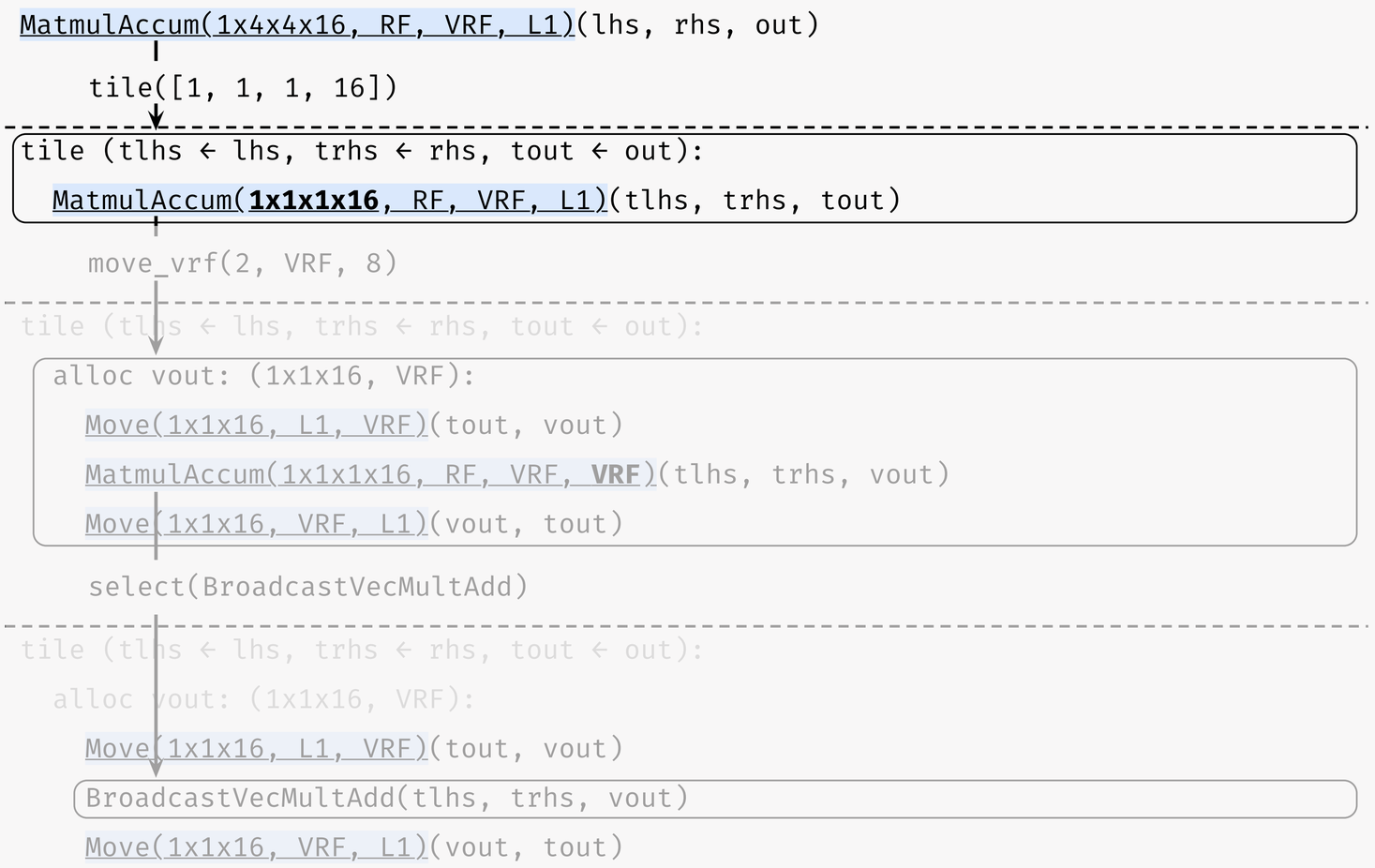


Lower via a terminating rewrite system: *Spec* – rewrite → partial program containing Specs.

- Incrementally fixes optimization decisions.
- IR is based on that of Fireiron (PACT ’20).

Building a Morello Implementation

Tilings



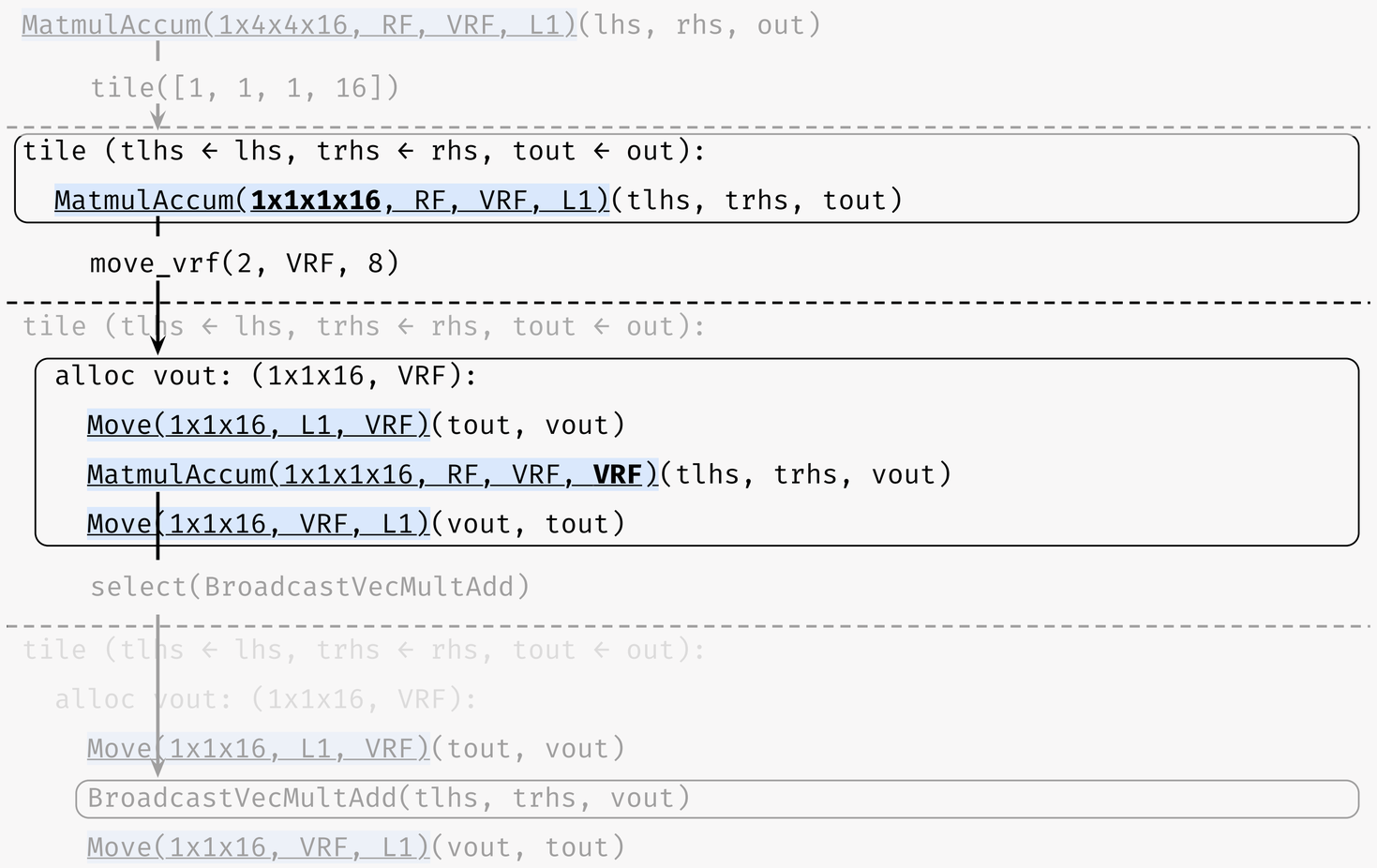
Tiling produces a loop.

The loop body is a new Spec (application).

Specs are references into the database.

Building a Morello Implementation

Moves



alloc scopes track lifetimes.

They can represent:

- Copies.
- Data layout changes.
- **Hardware cache misses.**
  - Avoids unexpected spills.
  - Planning cache lifetimes is critical to writing high-performance code.

Building a Morello Implementation

Microkernels (Leaves)

Complete by selecting microkernels (`BroadcastVecMultAdd` here) for the leaves.

Microkernels are small C codelets.

Scheduling terminates when no Specs remain.

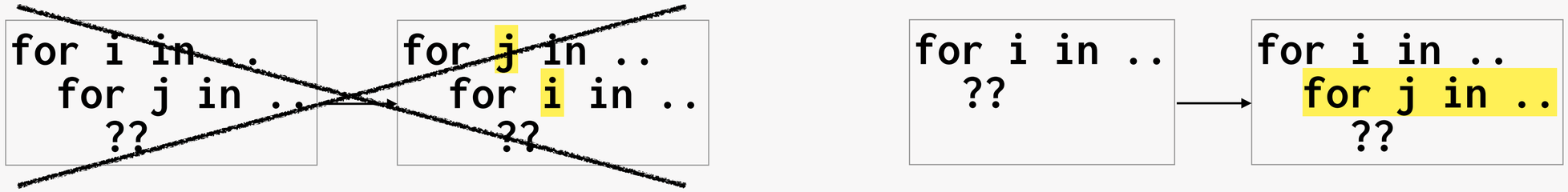
Later lowering to x86 C is essentially syntax-directed translation.

Building a Morello Implementation

Strictly Top-Down

Building strictly top-down…

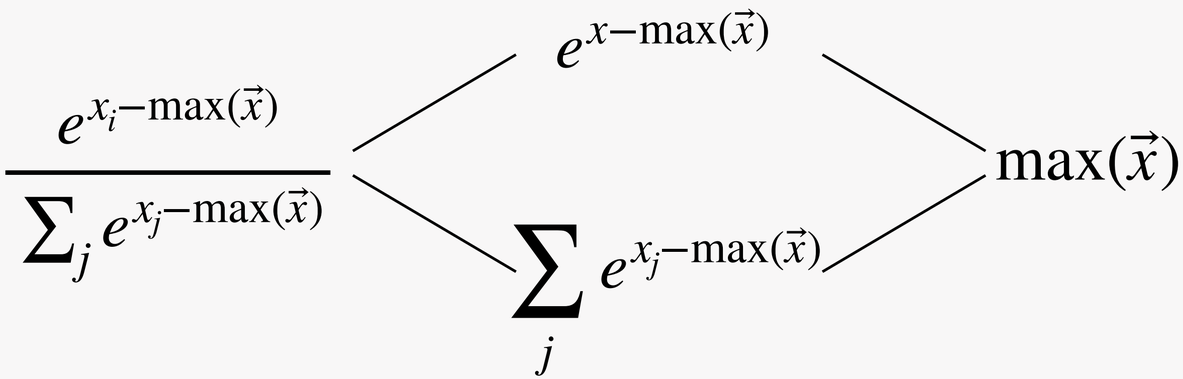
- terminates,
- enables an optimal substructure we'll introduce later, but
- precludes loop reordering and other interior node mutation.



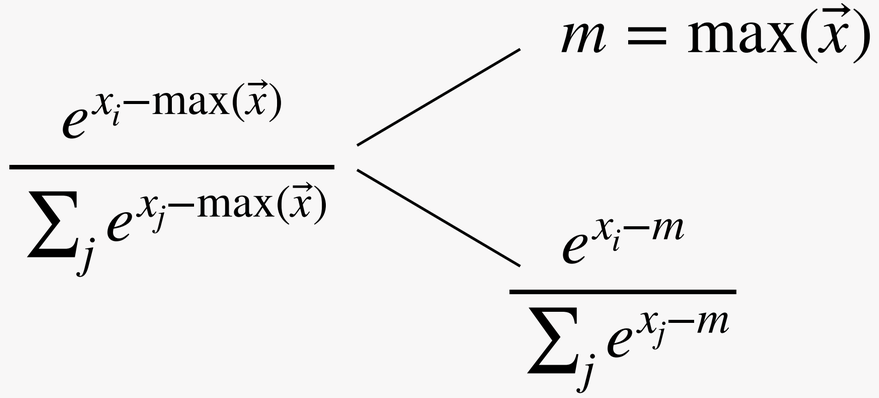
Softmax & Other Computation Graphs

Specs can also name computation graphs (compositions of Specs).

- Rewrites can decompose graphs into sub-graphs with intermediate buffers.
- e.g., Softmax might be decomposed like this:

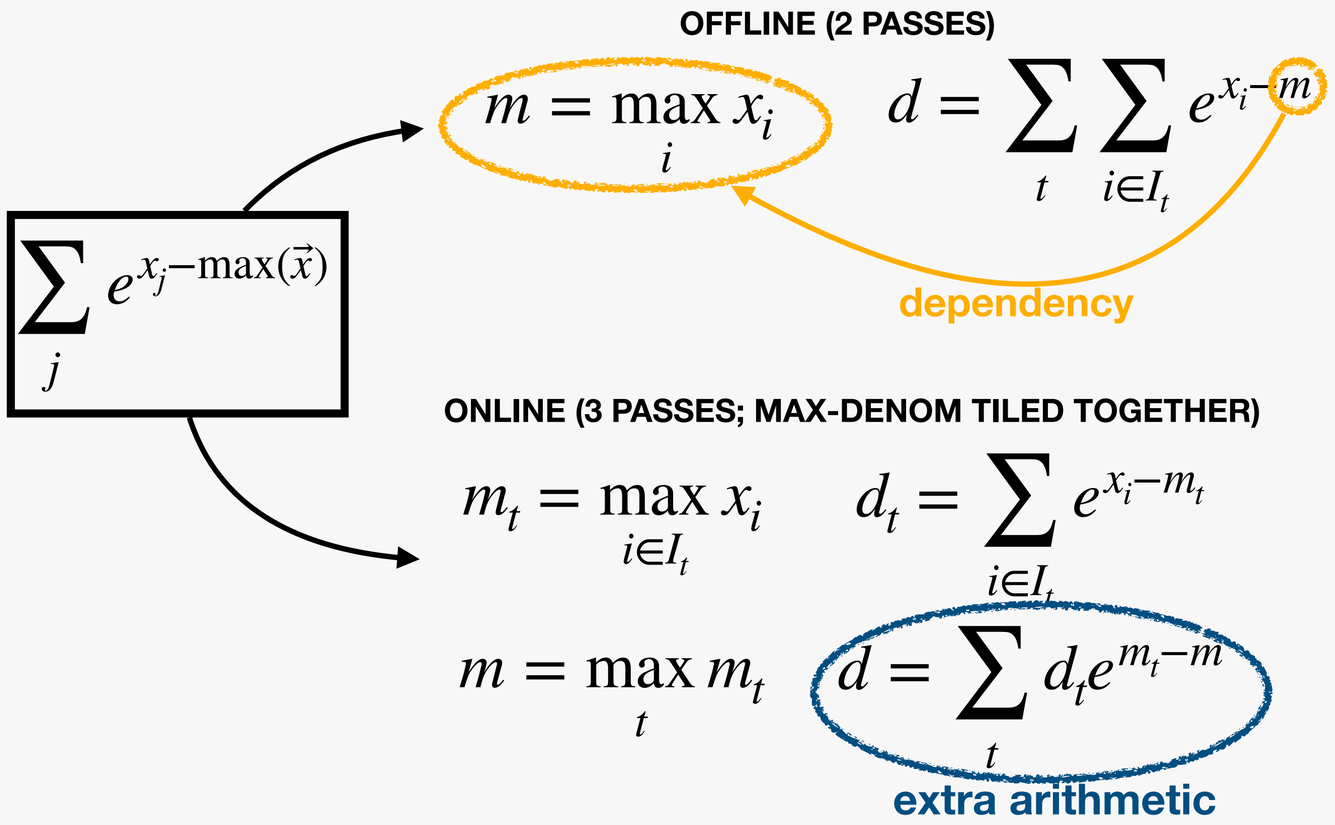


or



Softmax & Other Computation Graphs

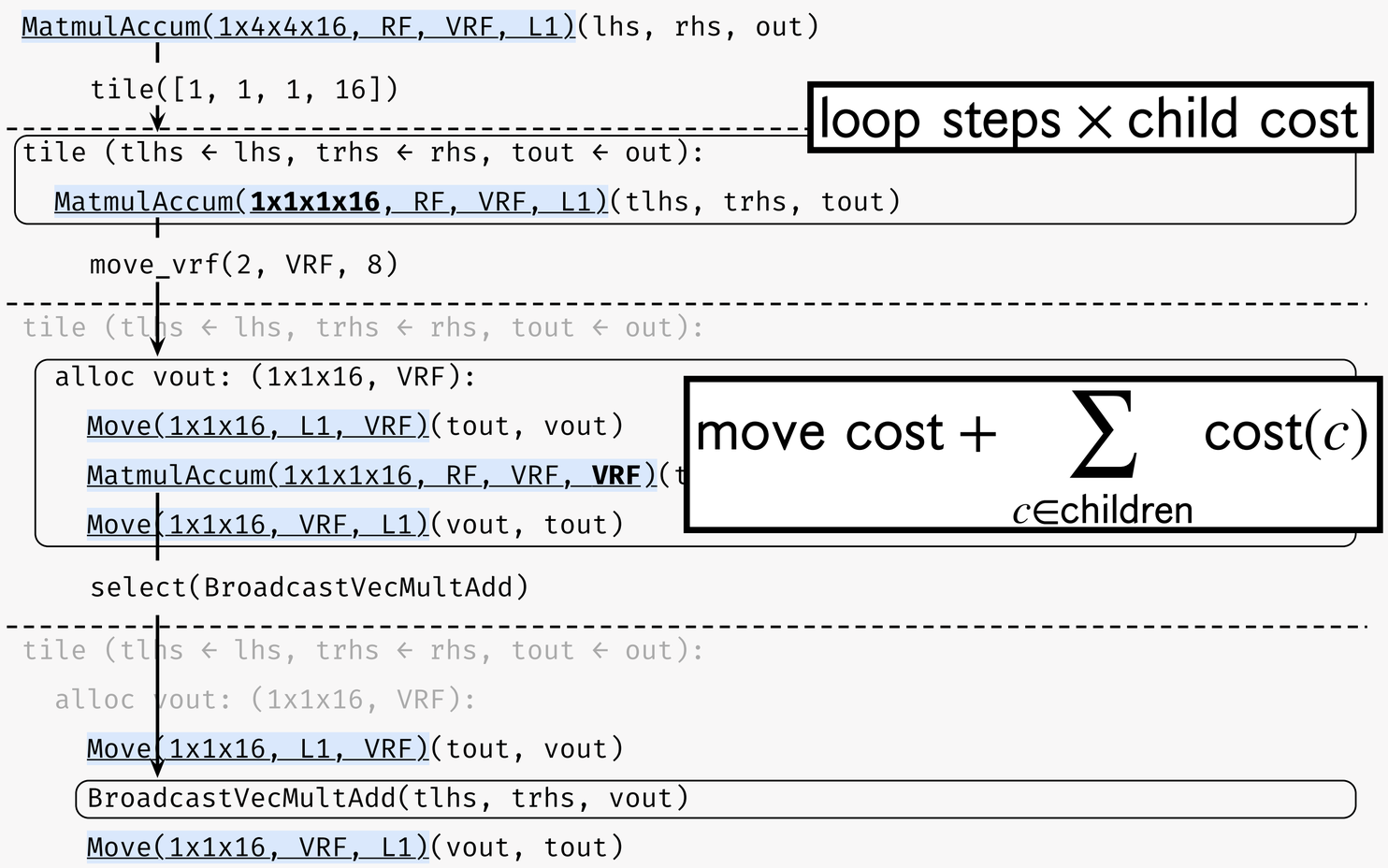
Lower a Softmax Spec to either “online” or “offline” algorithms.



Single IR for Joint Optimizations

- A single IR exposes critical optimization decisions and machine behavior (e.g. caches).
  - For example, how memory layout, tile size, and microkernel selection interact.
- A cost model defined over this IR allows for **joint** evaluation of these optimizations.
  - No guessing about what happens in a later phase or lower-level IR.

Cost Model



We define a compositional cost model over programs:

$$c : \mathscr{P} \to \mathbb{R}^+$$

Costs estimate **reciprocal throughput** (i.e., initiation interval).

- Affine in the costs of sub-trees:
  $c_{\text{node}}(\vec{x}) = A_{\text{node}}\vec{x} + b_{\text{node}}$

Spec contains everything needed to describe performance “context.”

- Most important: tensor residency and available memory (e.g. unused cache lines).

Minimization has an **optimal substructure**.

Cost Model

- **Fast:** Constant-time per program, given sub-program costs.
  - Avoids simulation and hardware benchmarking.
- **Simple:** Simple equations defined over program nodes are easy to read.
  - Performance engineers can straightforwardly debug unexpected cost rankings.
- **Enables dynamic programming** by having an optimal substructure.

1. IR & Cost Model

2. Search

3. Spatial Database

4. Evaluation

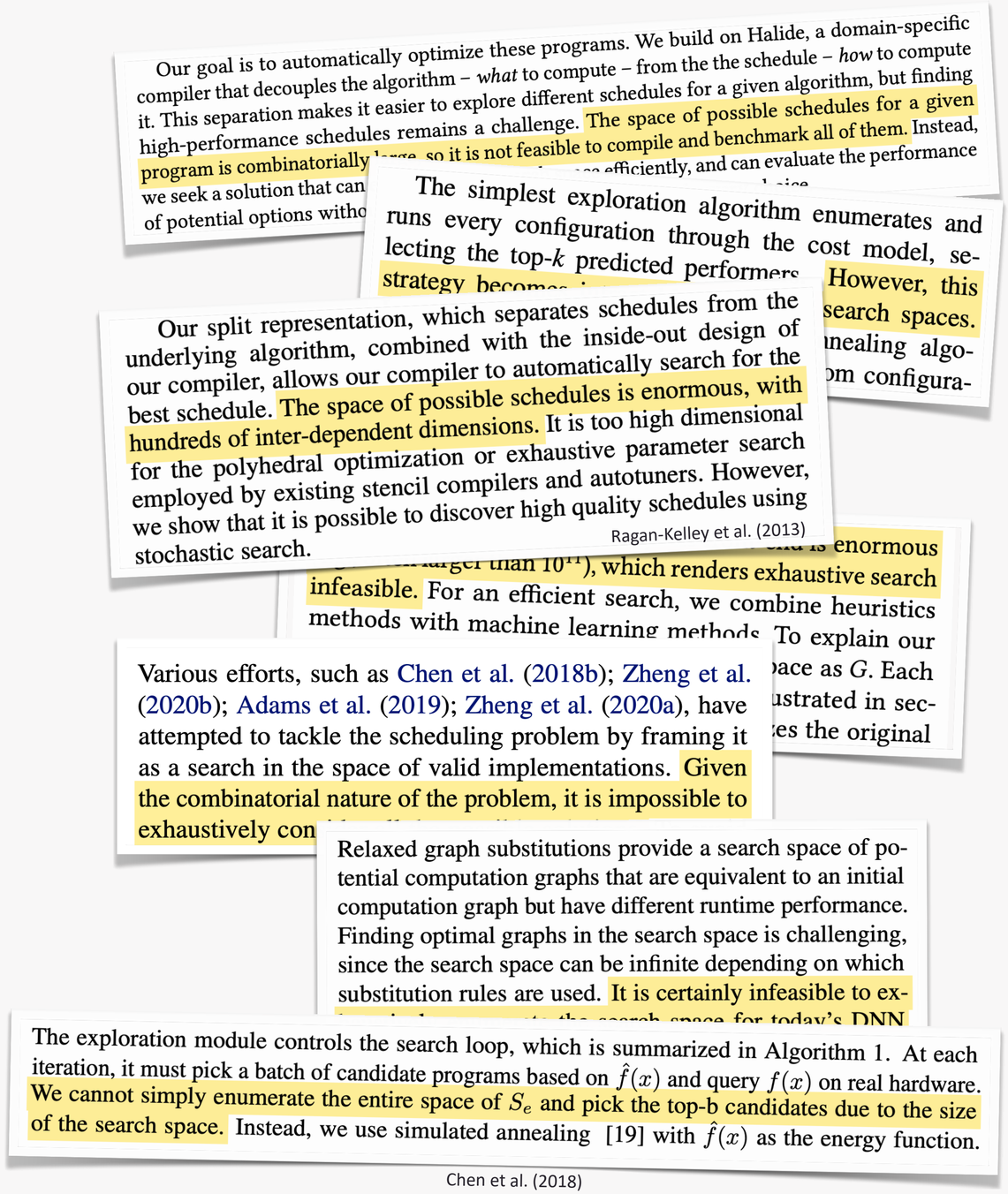
Exhaustive Search

Our cost model's optimal substructure assumption makes exhaustive search feasible.

- "Exhaustive" subject to optimal substructure assumption.

No heuristics are used.

In contrast, most prior work uses stochastic search because exploring the combinatorial space seems impossible.



Our goal is to automatically optimize these programs. We build on Halide, a domain-specific compiler that decouples the algorithm – *what* to compute – from the the schedule – *how* to compute it. This separation makes it easier to explore different schedules for a given algorithm, but finding high-performance schedules remains a challenge. The space of possible schedules for a given program is combinatorially large, so it is not feasible to compile and benchmark all of them. Instead, we seek a solution that can efficiently, and can evaluate the performance of potential options witho

The simplest exploration algorithm enumerates and runs every configuration through the cost model, selecting the top-k predicted performers. However, this strategy becomes search spaces. nealing algo- om configura-

Our split representation, which separates schedules from the underlying algorithm, combined with the inside-out design of our compiler, allows our compiler to automatically search for the best schedule. The space of possible schedules is enormous, with hundreds of inter-dependent dimensions. It is too high dimensional for the polyhedral optimization or exhaustive parameter search employed by existing stencil compilers and autotuners. However, we show that it is possible to discover high quality schedules using stochastic search.

Ragan-Kelley et al. (2013)

enormous larger than 10^{11}), which renders exhaustive search infeasible. For an efficient search, we combine heuristics methods with machine learning methods. To explain our pace as G. Each ustrated in sec- es the original

Various efforts, such as Chen et al. (2018b); Zheng et al. (2020b); Adams et al. (2019); Zheng et al. (2020a), have attempted to tackle the scheduling problem by framing it as a search in the space of valid implementations. Given the combinatorial nature of the problem, it is impossible to exhaustively con

Relaxed graph substitutions provide a search space of potential computation graphs that are equivalent to an initial computation graph but have different runtime performance. Finding optimal graphs in the search space is challenging, since the search space can be infinite depending on which substitution rules are used. It is certainly infeasible to ex-

The exploration module controls the search loop, which is summarized in Algorithm 1. At each iteration, it must pick a batch of candidate programs based on $\hat{f}(x)$ and query $f(x)$ on real hardware. We cannot simply enumerate the entire space of S_e and pick the top-b candidates due to the size of the search space. Instead, we use simulated annealing [19] with $\hat{f}(x)$ as the energy function.

Chen et al. (2018)

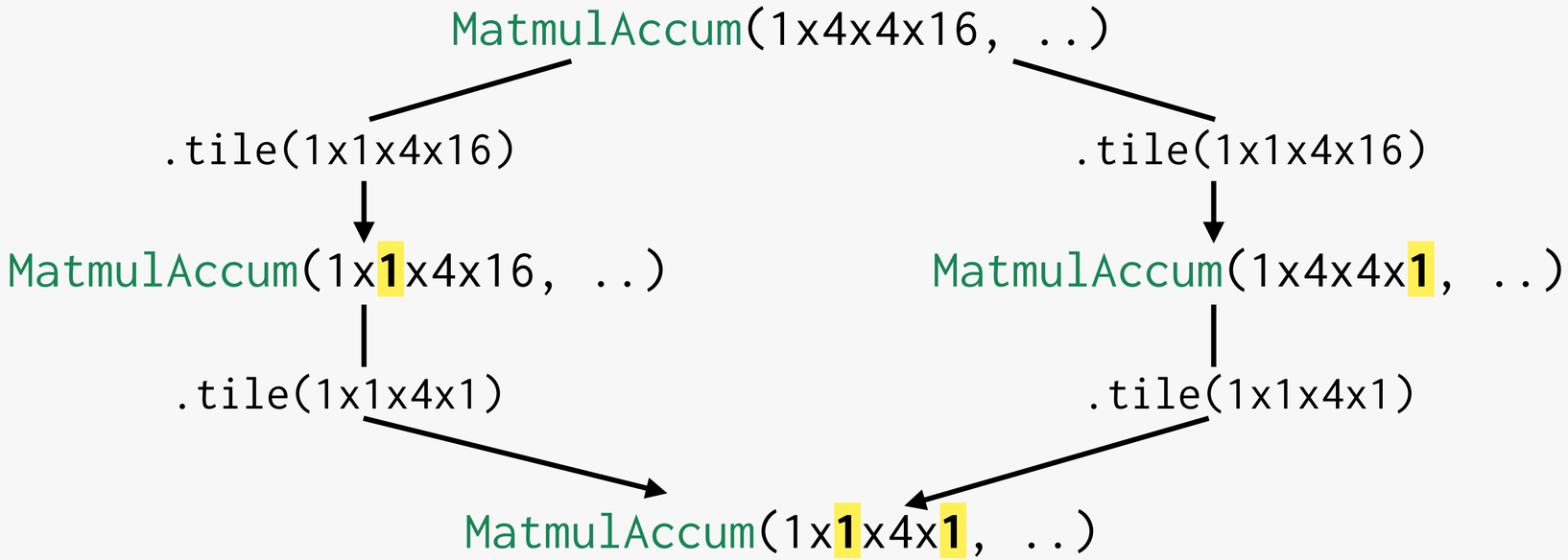
Two Search Algorithms

We implement two exhaustive search algorithms:

- **Top-Down**. This prunes some dependencies, but is serial.
- **Bottom-Up**. Cannot prune dependencies, but is highly parallel.

Top-Down Search

Apply depth-first enumerative search with memoization.



Top-Down Search

Memory-Limits Monotonicity

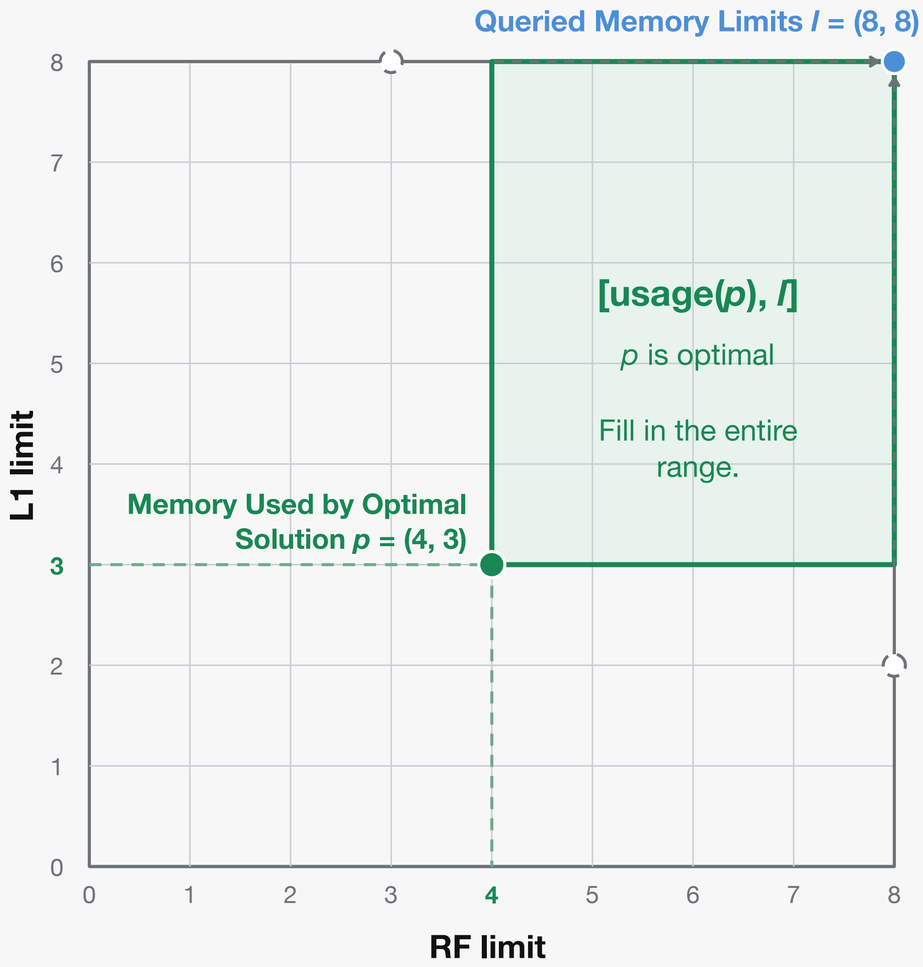
If a solution *p* is optimal below memory limits *l*, it is also optimal at all lower limits g.t.e. the actual memory used by *p*.

Correct because:

- *p* is valid at and above usage(*p*).
- *p* stays optimal: anything better in the range would have ‘won’ at the queried limits.

When storing to the database, we store *p* as the optima for all Specs with limits through used.

Reduces visited Specs by ~10x.

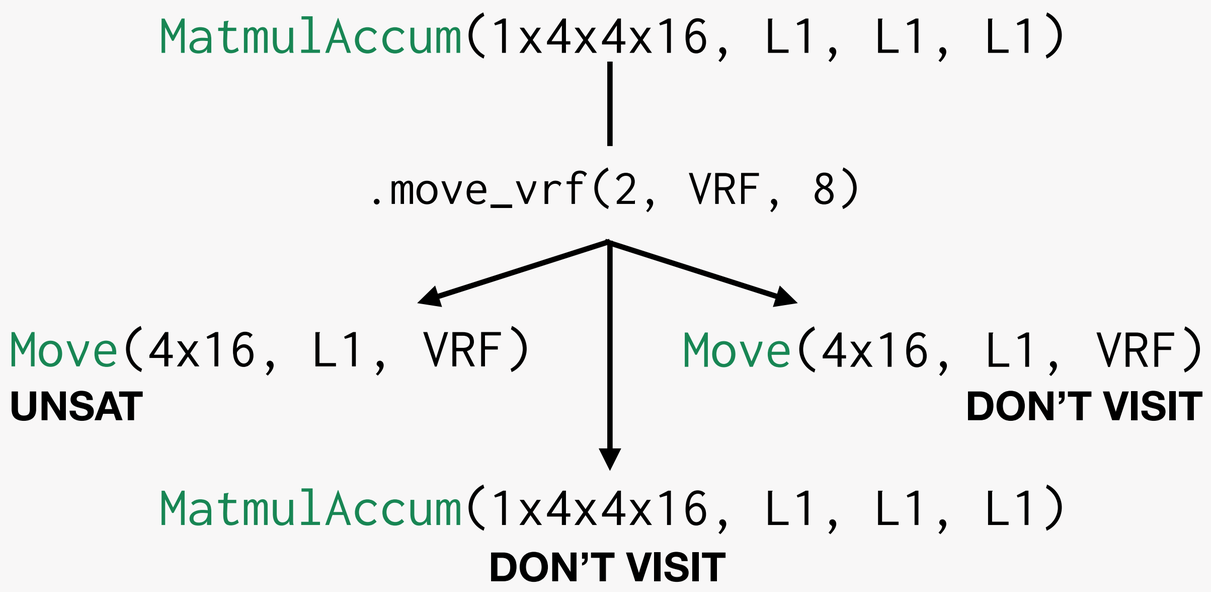


Top-Down Search

Short-Circuits

Top-down skips dependencies if pruned by short-circuiting.

- Dependencies in each reduct are visited depth-first.
- If the dependency Spec has no valid implementation, later dependencies are never visited.

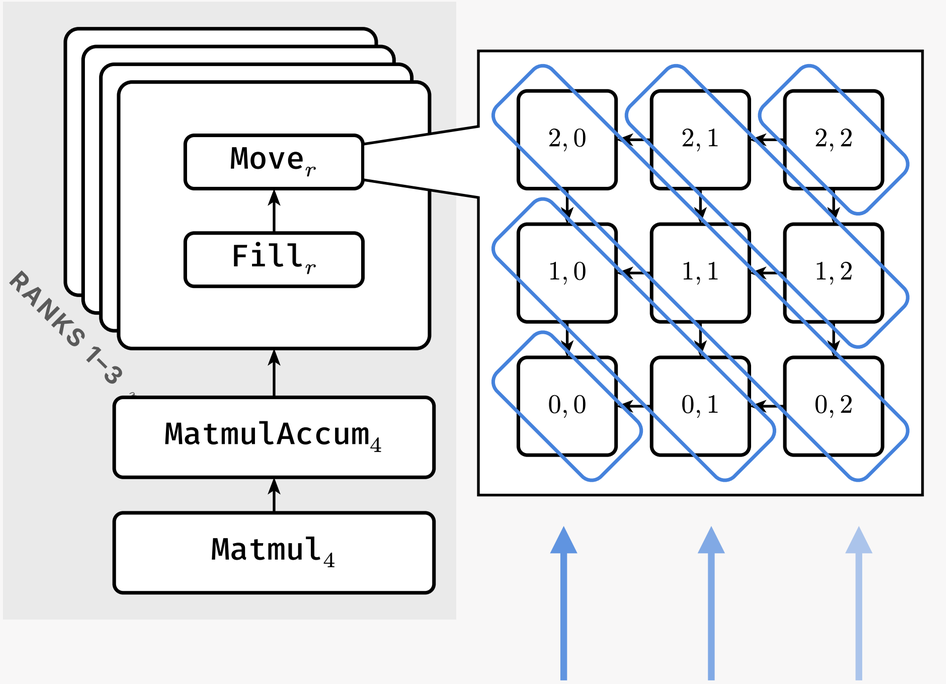


Bottom-Up Search

To compute a **whole database in parallel**:

- Define a visit order over Specs.
  - Visit order respects data dependencies.
- Implemented by applying the top-down search procedure for each Spec.
  - Dependencies are available, so this will not recurse

This is fast: computes a database for 22 Spec kinds through size 4 (incl. batch size) in ~6 hours.



Compute Specs in each diagonal in parallel.

1. IR & Cost Model

2. Search

3. Spatial Database

4. Evaluation

Storage Challenge

A naive memo. table from Spec → (cost, optimal rewrite) uses too much space.

- Values take 11 bytes each.
- **Example:** 11 bytes × | dependencies of Matmul(1 × 4 × 4 × 4) | = ~15.3 GiB

~1.5 billion

Worse if you include the keys!

We would like to scale to much larger operators.

Spatial Compaction

Motivating Insight

Perturbing a single Spec parameter is unlikely to change the optimal decision.

Example: optimal size-1024 and size-512 matrix multiplications tile to the same size.

Spatial Database

Representation

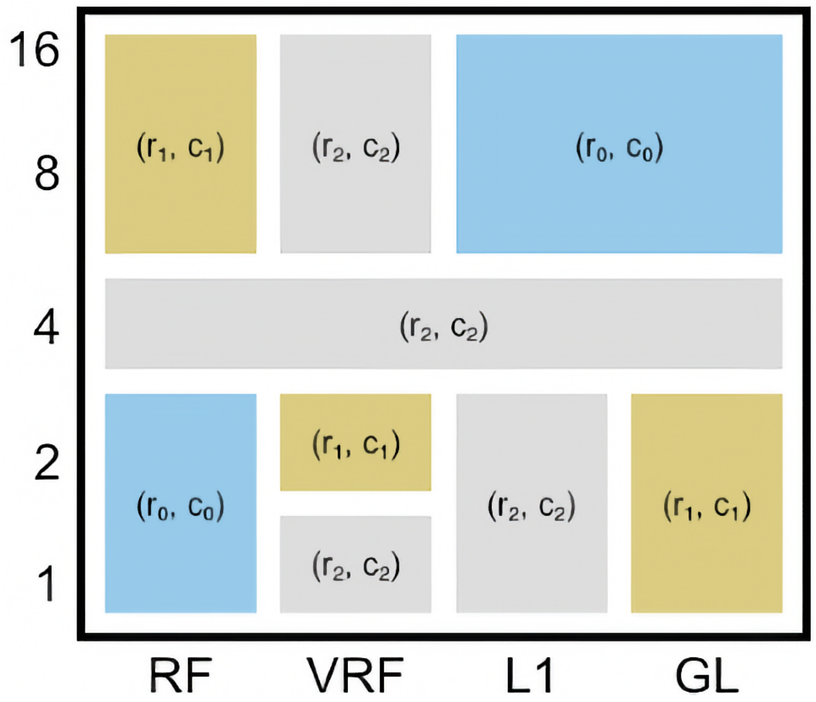
Map Specs to a point in a multi-dimensional space.

Database is a set of axis-aligned rectangles.

Map Spec-coordinates to rectangles covering intersecting those coordinates (no overlap)..

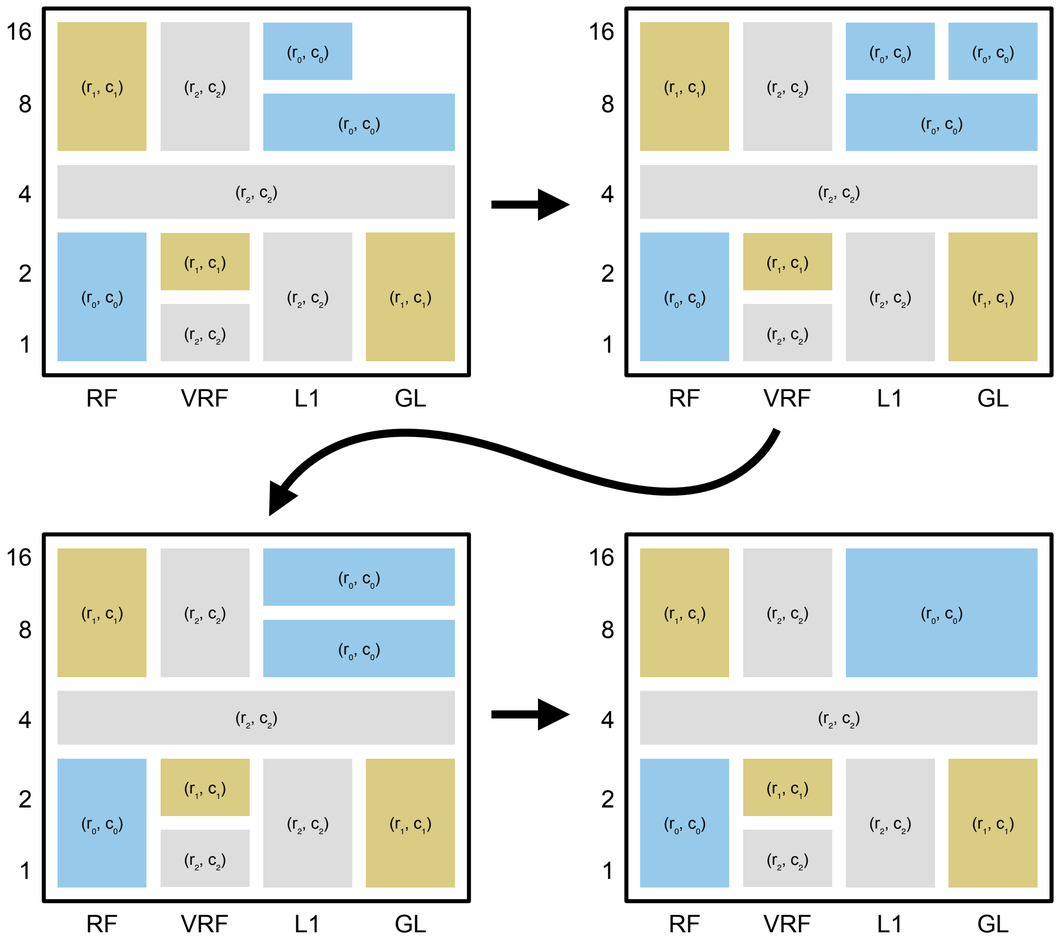
- Rectangles store **cost intensity (cost divided by Spec volume)** and the optimal rewrite.
  - **Cost intensity** removes a predictable scaling effect; e.g., doubling an operator's volume, all else being equal, doubles its cost.

Use R*-Tree to index.



Building the Database

- Optimally partitioning values into same-value rects. is intractable.
- Morello uses a greedy approximation on every insertion
  - Combine “adjacent” rectangles when values match.
  - Greedily merge with adjacent rectangles until a fix point



DB Evaluation

- In practice, each rectangle stores **1e4–1e6** (inclusive) entries.
  - This can be interpreted as the compression ratio over using a hash map.
- **This dramatically changes what can be stored.**
  For example, under a 16 GiB budget, spatial compaction allows us to store a table of matrix multiplications through size M=K=N=32,768, not M=K=N=4.
  - Also makes practical the sharing of reusable databases (e.g., shipping with Morello).

1. IR & Cost Model

2. Search

3. Spatial Database

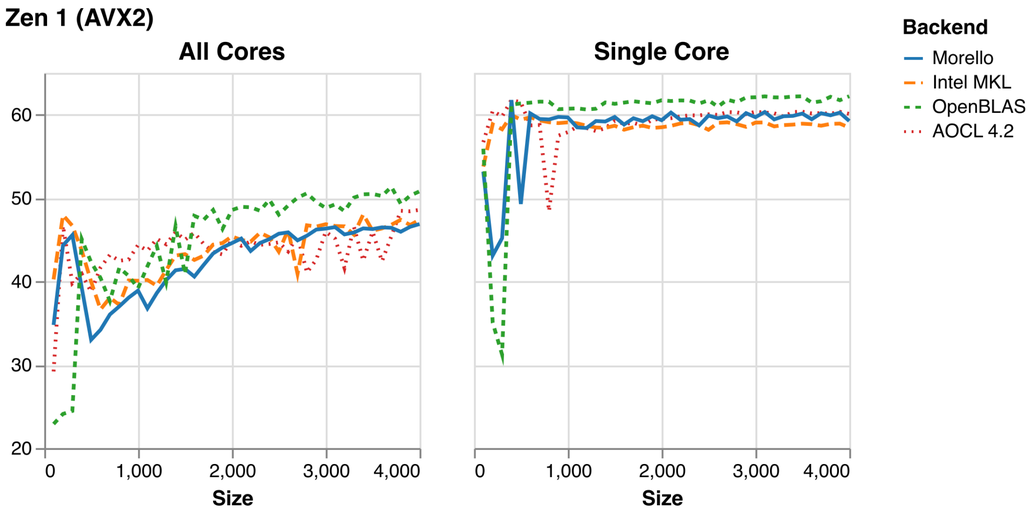
4. Evaluation

Experiments

- We test Morello's ability to generate matrix-multiplication and softmax implementations.
  - Common operators in DNNs.
  - Vary in typical arithmetic intensity and number of reduction axes.
- We target two very different CPUs:
  - An AVX2 AMD CPU from 2017
  - An AVX-512 AMD CPU from 2024
- We achieve competitive or SOTA performance on each.
- We developed for AVX2 and, with little work, retargeted for AVX-512.

	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

Experiment: Matrix Multiplication, float32



Manually scheduled some macro decisions:

- Outer tiling to M×K×N = 528×528×1056.
- The packed layouts of A and B.
- Inner tiling to pack size: M×N = 4×16 (AVX2); M×N = 8×48 (AVX-512).

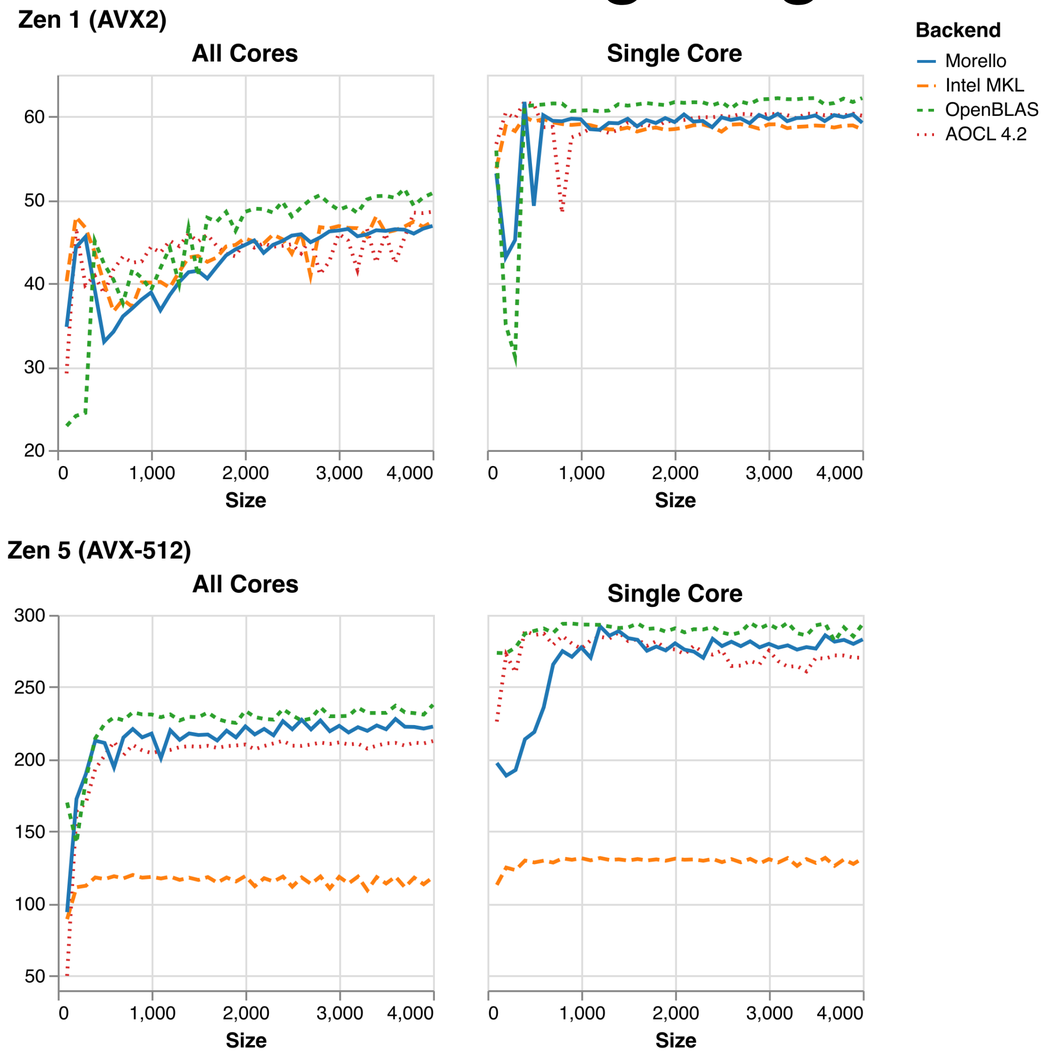
Then **synthesize:** 25m on AVX2 and 1.5 hours on AVX-512.

Competitive performance on Zen 1.

- Intel MKL, OpenBLAS, and AOCL are highly optimized baselines.

Experiment: Matrix Multiplication, float32

AVX-512 & Retargeting

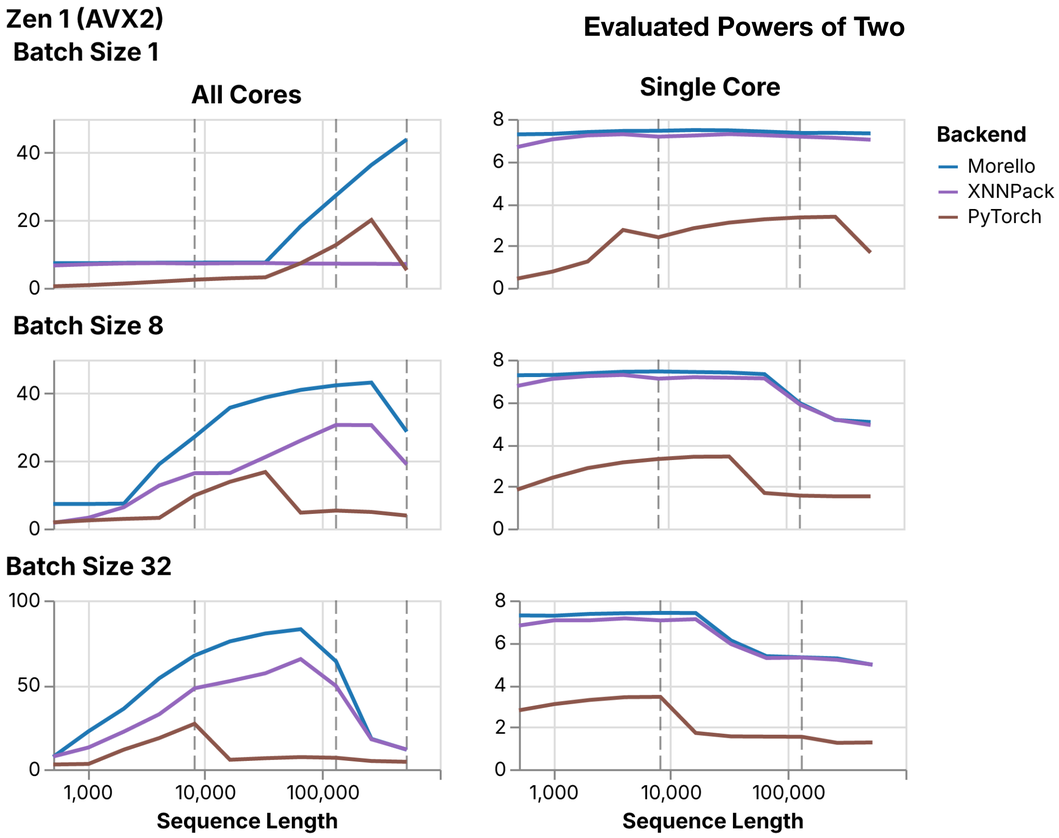


Outperform AOCL (and Intel MKL) at most sizes.

Retargeting to AVX-512 required few changes.

- **Add new instructions.**
- **Update the machine model:** add thirty-two 512-bit vectors, update the number of cores and cache throughput, and calibrate parallel loop overhead.
- **Add the manual tile size in the manual schedule mentioned previously.**

Experiment: Softmax, float32



Manual decisions made to accelerate search are:

- Force a top-level tiling over the batch dimension.
- Within the tiling, first stage computation of the denominator.

Then **synthesize:** 12m on AVX2 and 3.8 hours on AVX-512.

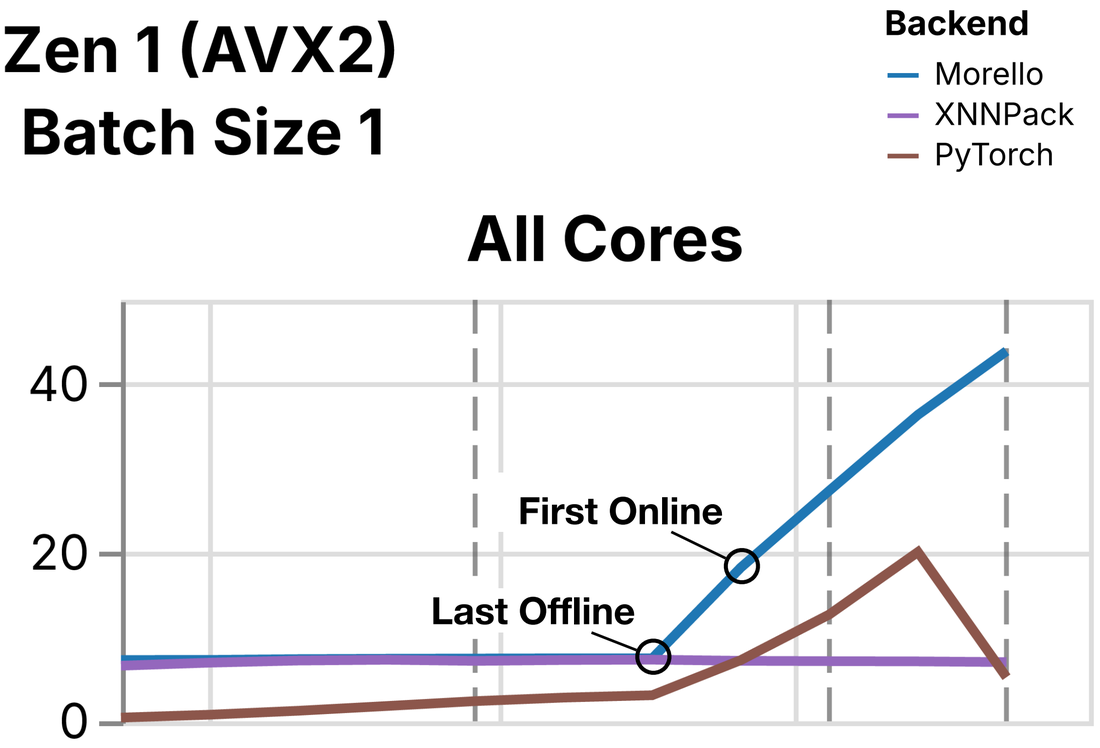
Morello's implementations outperform baselines in most configurations.

- Strongest baseline is XNNPACK, which backs XLA/TensorFlow on x86.

Experiment: Softmax, float32

Morello swaps to online when profitable

Zen 1 (AVX2)
Batch Size 1



At 64k, Morello preferred additional arithmetic to additional memory traffic.

OFFLINE (2 PASSES)

$$m = \max_i x_i \qquad d = \sum_t \sum_{i \in I_t} e^{x_i - m}$$

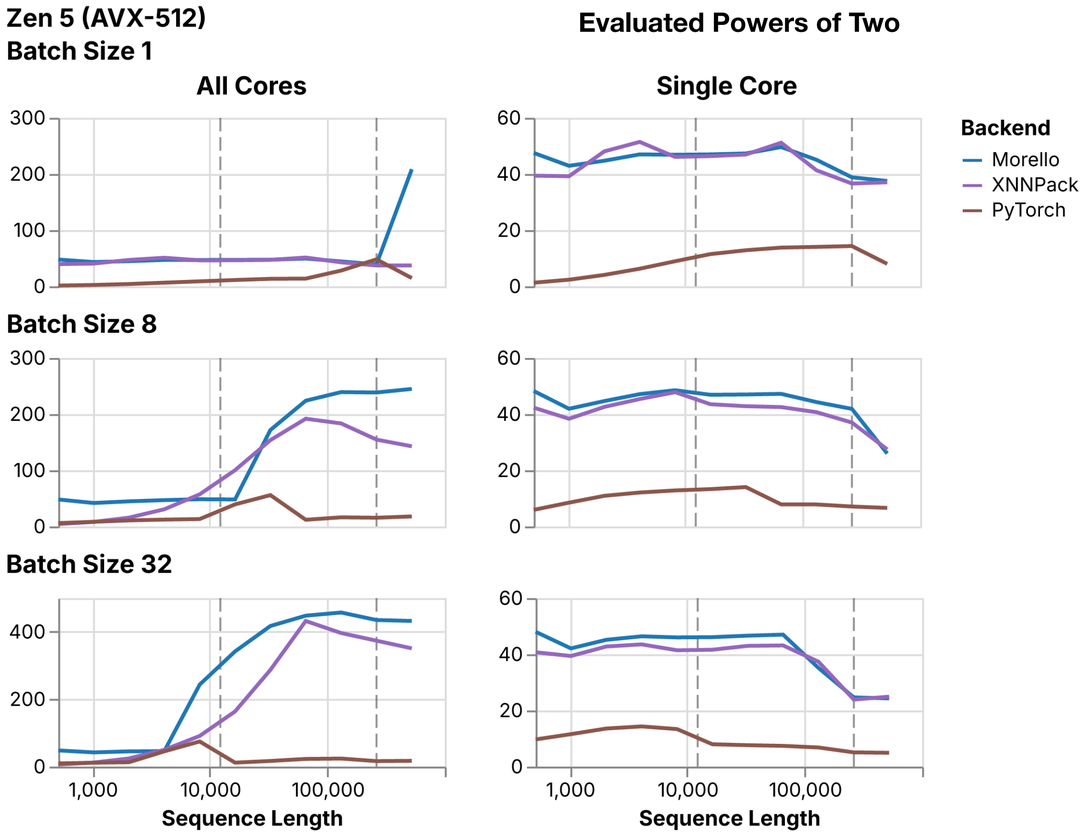
ONLINE (3 PASSES; MAX-DENOM TILED TOGETHER)

$$m_t = \max_{i \in I_t} x_i \qquad d_t = \sum_{i \in I_t} e^{x_i - m_t}$$

$$m = \max_t m_t \qquad d = \sum_t d_t e^{m_t - m}$$

Experiment: Softmax, float32

AVX-512 & Retargeting



As with matmul, we targeted AVX2 and later added AVX-512 support.

Retargeting required one thing: adding an AVX-512 version of our exponentiation helper.

As with AVX2, in most configurations, Morello is the fastest.

These experiments support Morello's design.

- **Competitive on two CPU generations.**
  - Matrix multiplication approaches OpenBLAS performance
  - Softmax matches and frequently outperforms XNNPACK
    - Cost model prefers different algorithms when arithmetic cost is outweighed by data movement cost.
- **Retargeting** to AVX-512 needed few changes and achieved strong performance.

Summary

Programs must be fast

Generated matrix-multiplication and softmax implementations are fast. Morello can select these because merging important optimization decisions (e.g., hardware cache behavior) into one IR allows for jointly evaluating their cost.

Compute implementations quickly

Many implementations.

Dynamic programming completed our matrix-multiplication and softmax implementations in 12–25 mins. (AVX2) or 1.5–3.8 hours (AVX-512).

Store implementations efficiently

At ~10 bytes per table entry, 2x10^14 entries is ~1.8 PiB for a matmul!

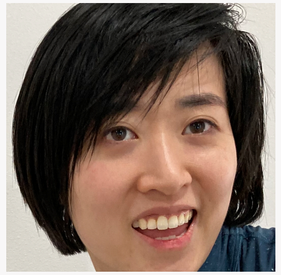
Our novel spatial memoization database reduces storage requirements by 4–6 orders of magnitude. Without the spatial database, we could store results for a M=K=N=4 matmul in a 16 GiB budget. With it, we can store M=K=N=32k.

Thanks to…

A Learned Performance Model for Tensor Processing Units

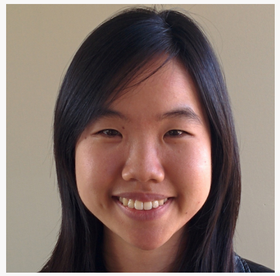
Samuel J. Kaufman [1 2 *] Phitchaya Mangpo Phothilimthana [1 *] Yanqi Zhou [1] Charith Mendis [1] Sudip Roy [1]
Amit Sabne [1] Mike Burrows [1]











Sampling Invariants from Frequency Distributions

Grigory Fedyukovich Samuel J. Kaufman Rastislav Bodík
University of Washington Paul G. Allen School of Computer Science & Engineering
{grigory, kaufmans, bodik}@cs.washington.edu

Prioritizing Mutants to Guide Mutation Testing

Samuel J. Kaufman
kaufmans@cs.washington.edu
University of Washington
Seattle, Washington, USA

Ryan Featherman
feathr@cs.washington.edu
University of Washington
Seattle, Washington, USA

Justin Alvin
jalvin@umass.edu
University of Massachusetts
Amherst, Massachusetts, USA

Bob Kurtz
rkurtz22@gmu.edu
George Mason University
Fairfax, Virginia, USA

Paul Ammann
pammann@gmu.edu
George Mason University
Fairfax, Virginia, USA

René Just
rjust@cs.washington.edu
University of Washington
Seattle, Washington, USA

github.com/samkaufman/morello