Pushing Tensor Accelerators beyond MatMul in a User-Schedulable Language

Yihong Zhang
University of Washington, USA

Derek Gerstmann
Adobe, USA

Andrew Adams
Adobe, USA

Maaz Bin Safeer Ahmad
Adobe, USA

***Abstract*— Tensor accelerators now represent a growing share of compute resources in modern CPUs and GPUs. However, they are hard to program, leading developers to use vendor-provided kernel libraries that support tensor accelerators. As a result, the usage of tensor accelerators is limited to the provided interface, mainly designed for traditional ML and scientific computing workloads.**

In this paper, we show that tensor accelerators can improve the performance of applications beyond simple variants of MatMul. For example, many image processing pipelines are linear transformations over matrices in disguise and can therefore utilize such specialized hardware. This is nonetheless hindered by the difficulties in programming tensor accelerators. We tackle this problem with compiler-based techniques. We use the Halide user-schedulable language and express operations as Halide algorithms succinctly. To this end, we implement a flexible tensor instruction selector based on equality saturation. The tensor instruction selector supports both CPU- and GPU-attached tensor accelerators and works with existing scheduling operations (e.g., producer-consumer fusion). Together, this enables developers to write diverse accelerator-leveraging applications in a few dozen lines.

Using our system, we demonstrate the potential of tensor accelerators beyond their traditional domains. We implement several image processing pipelines (e.g., filtering, resampling, and denoising) in our system and evaluate them against non-accelerator-leveraging baselines. We show that these pipelines can achieve significant speedups. For example, a downsampling routine is sped up by $6.1\times$ by utilizing Tensor Cores on an Nvidia RTX 4070 GPU.

***Index Terms*—Program optimization, tensor accelerators, user-schedulable languages, equality saturation.**

I. Introduction

The unrelenting demand for performance in AI and scientific computing has driven hardware vendors to develop highly efficient domain-specific accelerators for matrix operations, with considerable success. In fact, the ability to leverage such hardware accelerators has become essential to extracting peak performance. For example, the Tensor Cores in Nvidia's RTX 4090 deliver roughly $2\times$ throughput of its general-purpose CUDA cores, while datacenter GPUs like the H100 offer 8-16$\times$ more potential. Yet such architectures are highly specialized, constantly evolving, and hard to program. Instead of programming against specialized hardware, developers use vendor-provided kernel libraries that consist of hardware-specific routines.

These kernel libraries support common math operations like matrix multiplication (MatMul), but the tensor accelerators could be useful for many other tasks. For example, many operations on signals are linear transformations over inputs and can be modeled as MatMul. However, these tasks remain inaccessible due to the narrow, rigid interfaces of kernel libraries. As a result, existing state-of-the-art implementations for these operations fail to utilize specialized hardware at all. Even in cases where application developers can contort their operations and data layouts around the provided API, the rigid interface also limits their ability to perform program optimizations such as code fusion, data layout reorganization, and workload scheduling, leading to inferior implementations.

In this paper, we study performance-critical applications where tensor accelerators are underused. We argue user-schedulable languages [17, 5] like Halide present a promising approach to expand the reach of hardware accelerators beyond the capability of kernel libraries. Halide is a domain-specific language for high-performance applications based on the idea of decoupling algorithms from schedules. Developers can succinctly describe their applications in a few lines of Halide algorithms, and use schedules for fine-grained controls over machine-specific execution strategies (e.g., loop tiling, vectorization). Ideally, developers would develop their tensor-accelerator-empowered kernels by focusing on the high-level algorithms and schedules, without worrying too much about the mapping to specific accelerator calls.

Current user-schedulable language implementations lack flexible support for tensor accelerators. State-of-the-art implementations hard-code supported syntactic patterns and use pattern-based rewriting rules. This approach is brittle and inflexible and takes excessive developer cycles. For example, it took Halide developers seven months to add initial support for Intel's Advanced Matrix Extensions (AMX) accelerator [20]. However, despite the considerable effort, it only recognizes a very limited syntactic form. A later pull request that generalized this support [6] took another six months to merge. Yet we found the existing implementation contained a number of bugs.

We revamp Halide's support for tensor accelerators with a new instruction selector, dubbed HardBoiled. The goal of HardBoiled is to support flexible use of tensor accelerators for domain applications. To this end, we use equality saturation (EqSat), a technique for rewrite-based program optimization that is robust to syntactic variations in the source program [24]. We repurpose Halide's existing vector abstraction for representing tensor programs and express tensor identities as rewrite rules in equality saturation. This provides a consistent user

interface, as from the user's perspective, tensor instructions are similar to vectorized instructions—just with more than one dimension.

Using our system, we explore the extent to which applications in signal and image processing can be sped up by tensor accelerators. As a case study, we consider the following applications: 1D/2D convolution, integer-factor resampling, non-integer-factor resampling, recursive filtering, and denoising using discrete cosine transform. We tuned and evaluated these applications on Tensor Cores and show that they can achieve up to 6.1× speedup on convolution-like kernels, and 1.1× to 1.4× speedup on end-to-end applications. As an additional validation, we have also implemented several kernels representative of classical ML workloads in our system and show that they are comparable to library implementations.

To summarize, we make two contributions in this work. First, we implement HARDBOILED, a tensor instruction selector for Halide. HARDBOILED can identify and compile diverse workloads for tensor accelerators and works seamlessly with Halide schedules. It supports both Intel AMX and Nvidia Tensor Cores. Second, using our system, we implement a set of classical image processing workloads in ways that effectively utilize tensor accelerators. We show that they achieve significant speedups compared to non-accelerator-leveraging baselines under appropriate settings.

II. BACKGROUND

A. Tensor Accelerators

Modern hardware has increasingly added specialized AI accelerators to offload math-heavy tensor computations like MatMul. For example, Nvidia's Tensor Cores are dedicated accelerators that support fast matrix operations for specific sizes and layouts. Similarly, Intel's Advanced Matrix Extensions (AMX) support matrix operations between tile registers with sizes up to 16×32. The ability to leverage these accelerators can lead to significant speedups.

On the other hand, tensor accelerators are underutilized. Their uses are limited primarily to traditional ML/AI and scientific computing workloads. As we will show in this paper, many operations over signals are linear tensor operations and can leverage these tensor accelerators. Nonetheless, state-of-the-art kernel implementations of these operations often fail to use such specialized hardware. This is unfortunate, given the massive industrial investment in improving their throughput.

This underutilization can be partially attributed to the difficulty in programming hardware accelerators, which forces developers to use vendor's implementations of a few common high-level operations. Because library interfaces are often coarse-grained, they prevent optimizations such as producer–consumer fusion, making them ill-suited for many domain-specific applications. A more effective approach is to provide developers with a user-friendly interface for programming tensor accelerators.

B. User-Schedulable Languages

In a user-schedulable language like Halide, developers express the desired computations as functional definitions of arrays, without regard to how those computations are executed. At this stage, the focus is purely on the high-level computation. For example, MatMul may be expressed in Halide as follows:

```
mm(y, x) = 0.f;
mm(y, x) += A(r, x) * B(y, r);
```

In a separate stage, developers write *schedules* that describe the optimization details for the computation, such as tiling, loop orders, and vectorization.

For example, the following schedule breaks down the computation of `mm` into 8×2 tiles. Within each tile, the y axis is vectorized while the x axis is unrolled.

```
Var xi, yi;
mm.in().tile(x, y, xi, yi, 2, 8)
  .vectorize(yi).unroll(xi);
mm.compute_at(mm.in(), y);
mm.vectorize(yi).unroll(xi);
mm.update().vectorize(yi).unroll(xi);
```

With this schedule, our matrix multiplication compiles to the following low-level code.

```
for x in range(0, N, 16):
 for y in range(0, M, 2):
  float mm0[8][2] # local 8x2 buffer
  mm0[0:8, 0] = 0;  mm0[0:8, 1] = 0
  for r in range(0, K):
   mm0[0:8, 0] += A[r, x]   * B[y:y+8, r]
   mm0[0:8, 1] += A[r, x+1] * B[y:y+8, r]
  mm[y:y+8, x]   = mm0[0:8, 0]
  mm[y:y+8, x+1] = mm0[0:8, 1]
```

C. Tensor Accelerators in User-Schedulable Languages

User-schedulable languages like Halide present a promising paradigm to host developments of tensor-accelerator-based applications. Halide enables advanced optimizations, such as tiling and fusion, to be expressed in just a few lines, and it generates efficient low-level implementations for various hardware platforms. This allows developers to prototype their applications for various hardware quickly.

However, languages like Halide have only limited support for hardware accelerators. To add support, compiler developers must match all possible syntactical patterns and handle different corner cases. This typically results in fragmented, imperative implementations and special-case handling. Currently, Halide's support for Intel AMX is limited to a small set of patterns and contains multiple bugs, and no support for Nvidia Tensor Cores exists.

An alternative approach is taken by TVM [5], which provides a user-extensible pattern-matching system for accelerator intrinsics via its tensorize directive. While this places higher demands on the user, with this system, a user can explicitly tell the compiler how some portion of their algorithm maps to an accelerator intrinsic, without having to wait for official compiler support. In practice, programs in TVM are often auto-scheduled with works like TensorIR [8] and Ansor [36].

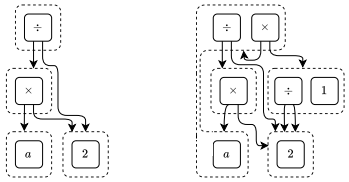

Fig. 1: Two example E-graphs [31]

We focus on scenarios where application developers write schedules manually, as is typical in Halide's production usage. We compare the two approaches further in Section VI.

D. Equality Saturation

Our goal is to build a robust and flexible tensor instruction selector to support our exploration of accelerator-driven domain-specific applications. To this end, we use equality saturation (EqSat) [31, 24]. EqSat is a program optimization technique that efficiently explores program spaces defined by rewrite rules. It explores combinations of different rewrite orders to find the optimal program. To mitigate the combinatorial blowup, it uses a data structure called E-graphs to compactly represent the equivalence classes of programs. Figure 1 shows two example E-graphs. The E-graph on the left represents the program $(a \times 2) \div 2$, and on the right is the E-graph after rules $(a \times 2) \div 2 \rightarrow a \times (2 \div 2)$, $2 \div 2 \rightarrow 1$, and $a \times 1 \rightarrow a$. Nodes grouped by the same dashed boxes (called *E-classes*) belong to the same equivalence class. For instance, the E-graph on the right shows $(a \times 2) \div 2$ is equivalent to a.

In this paper, we use egglog [34], a recent system that combines equality saturation and deductive reasoning. Two kinds of rules are supported in egglog: **`rewrite`** and **`rule`**. For example, `(rewrite (Add x y) (Add y x))` defines commutativity of addition. A **`rule`** consists of a query and an action. For instance, the following rule defines the type derivation for function application.

```
(rule ((= e (App e1 e2))
       (has-type e1 (Arrow t1 t2))
       (has-type e2 t1))
      ((has-type e t2)))
```

In its query, this rule looks for the pattern `(App e1 e2)` in the E-graph satisfying `e1` has type `t1`→`t2` and `e2` has type `t1`. For each match of this pattern, the rule asserts that `(App e1 e2)` should have type `t2`. In fact, a **`rewrite`** is a special form of **`rule`** and can also be desugared using the **`union`** keyword, e.g., `(rule ((= e (Add x y))) ((union e (Add y x))))`.

III. Writing Tensor Kernels with HardBoiled

In this section, we present the design of HardBoiled and the workflow of using HardBoiled-enabled Halide to write kernels that use tensor accelerators.

HardBoiled works as a compiler pass in Halide's optimization pipeline. To compile statements to take advantage of tensor accelerators, a user specifies which buffers are stored in tensor accessible memory via Halide scheduling primitives.

```
A[ramp(ramp(0, 8, 4), x4(1), 8)]
vector_reduce_add(
    A[x8(ramp(0, 1, 3))]
        * B[ramp(ramp(0, 1, 3), x3(1), 8)],
    3)
vector_reduce_add(
    A[ramp(x6(ramp(0, 1, 3)), x18(3), 4))]
        * B[x4(ramp(ramp(0, 6, 3), x3(1), 6))],
    3)
```

Fig. 2: The fully-vectorized representation in Halide IR of a 4x8 transpose, a 3-tap convolution of a signal of size 8, and the multiplication of a 4x3 matrix with a 3x6 matrix, respectively.

HardBoiled relies on Halide's vectorization pass to lower a user's program to a vectorized representation. The tile extractor annotates the IR program with data movements, generates an EqSat program, and sends the program to egglog. After egglog finishes optimizing, the tile extractor parses the optimized program back into Halide and performs post-processing. Finally, the optimized program is passed back to resume Halide's compilation passes, which generate the final machine code.

HardBoiled uses combinations of three existing Halide IR nodes to represent linear tensor operations. All three nodes arise naturally from vectorizing loops in Halide. The first is `Ramp`, which takes a base, a stride, and a length, and represents a vector containing a linear sequence. When a loop is vectorized, instances of the loop variable are replaced with a `Ramp` representing all values of the loop index. The base and stride may themselves be vectors, in which case `Ramp` is the flattened concatenation of a linear sequence of vectors. This occurs when two nested loops are both vectorized. The second IR node is `Broadcast`, which can be thought of as a `Ramp` with a stride of zero. Like `Ramp`, its argument may itself be a vector, where it represents the concatenation of copies of that vector. A broadcast of a value v by a factor of n is written tersely as $\texttt{x}n(v)$. To load and store small tensor tiles, nested combinations of ramps and broadcasts are used as indices to vector load and store instructions. The third relevant IR node is `vector_reduce_add`, which sums fixed-size groups of adjacent scalars to produce a smaller vector. These appear when a reduction is vectorized along the reduction dimension. Some combinations of these IR nodes that represent common tensor operations are shown in Figure 2.

In the rest of this section, we demonstrate HardBoiled by exploring MatMul in detail. Note that a core goal of HardBoiled is to handle more complex computational patterns that are not literal MatMul (e.g., resampling). However, even straightforward MatMul is already non-trivial to support in compilers.

A. Scheduling MatMul

Consider the following MatMul in Halide[1]:

```
ImageParam A(BFloat(16), 2), B(BFloat(16), 2);
```

[1]Halide follows the indexing convention of OpenGL, where the innermost dimension is written first (the opposite of C-like languages).

```
2  Var x, y; RDom r(0, 16);
3  mm(y, x) = 0.f;
4  mm(y, x) += cast<float>(A(r, x))
5              * cast<float>(B(y, r));
6
7  mm.in().bound(x, 0, 32).bound(y, 0, 16);
```

Line 1-6 describes a MatMul program in Halide, and the last line constrains the computation to be between a 16x32 matrix and a 32x16 matrix. The user then writes the following schedule to compile this MatMul program to the AMX accelerator. The schedule (1) uses the `store_in` directive to ask `mm` to be stored in AMX tile register and (2) vectorizes relevant computation:

```
1  mm.store_in(MemoryType::AMXTile)
2    .compute_at(mm.in(), x);
3  mm.vectorize(x, 16).vectorize(y, 16);
4  mm.update().atomic()
5    .vectorize(x, 16).vectorize(y, 16)
6    .vectorize(r, 32);
7  mm.in().vectorize(x, 16).vectorize(y, 16);
```

This schedule has four statements. The first statement defines the storage of `mm` and creates a wrapper buffer `mm.in()` (in memory) that loads the computed matrix from `mm`. The rest of the statements `vectorize` the three stages in the computation with corresponding tile sizes: initialization (`mm`), MatMul (`mm.update()`), and loading to the wrapper buffer (`mm.in()`).

At this step, the user specifies the target hardware of each computation stage using Halide scheduling directives. This defines the specifications that the instruction selection needs to follow. For example, it is the instruction selector's job to find available AMX instructions that compute *mm* in AMX tile registers. In our case, This can be mapped to Intel AMX's `TDPBF16PS` instruction, which does exactly $AB + C$ for `BFloat16` matrices of this shape.

B. Matching MatMul

Guided by the provided schedule, Halide lowers the MatMul program to Halide's intermediate representation using a series of compiler passes including simplification and vectorization. The resulting IR is shown in Figure 3 (minus the highlighted data movement nodes).

HARDBOILED injects data movement nodes `mem_to_amx` and `amx_to_mem` to indicate program locations where data movements between hosts and accelerators happen. While Halide's IR does not distinguish between computations in different memories and implicit inject data movement instructions during codegen, HARDBOILED needs to explicitly differentiate them: The E-graph should not treat a MatMul computed in memory as being equivalent to a MatMul in a tensor register, since using the result of the latter requires first loading the data to memory.

HARDBOILED prints this program as egglog expressions and sends them to egglog along with a set of predefined egglog rules. To identify accelerator-leveraging opportunities, egglog looks for the following pattern:

```
(= e (Add C (VectorReduceAdd 512
  (Mul (Cast (Float32 8192) lhs)
       (Cast (Float32 8192) rhs)))))
(= lhs (Load (BFloat16 8192) A
  (Ramp (Broadcast (Ramp A-base 1 32) 16)
        (Broadcast A-stride 512) 16))))
(= rhs (Load (BFloat16 8192) B
  (Broadcast (Ramp (Ramp B-base B-stride 32)
                   (Broadcast 1 32) 16) 16)))
```

It looks for a pattern `e` with the form $C + \sum lhs \cdot rhs$, where *lhs* and *rhs* are expressions that load matrices A and B with certain access patterns. The access patterns are expressions above with nested `Ramp`s and `Broadcast`s.

The access patterns for *lhs* and *rhs* can be read as follows: both matrices are enumerated with three loop variables (corresponding to the three nesting levels of `Ramp`/`Broadcast`). The outermost loop dimension enumerates over `A` and is constant over `B`, which corresponds to variable x in the source program. The second loop enumerates over `B` and is constant over `A`, so it corresponds to `y`. The innermost loop co-iterates `A` and `B` and corresponds to the reduction domain r. The (pointwise) multiplication between `lhs` and `rhs` creates a vector with 8192 lanes, which is then reduced to 512 lanes via `VectorReduceAdd`. This is exactly the semantics of a MatMul.

While the pattern we try to match directly follows MatMul, Halide lowers the input program to a different shape. In Figure 3, matrix A is loaded with index `ramp(broadcast(..), ..) + broadcast(ramp(..), ..)`, while the expected pattern has shape `ramp(broadcast(ramp(..), ..), ..)`. Similarly, the B matrix is loaded with `ramp(ramp(..), ..)` and then broadcasted, again not matching our pattern. This mismatch is caused by Halide's aggressive simplification passes that run throughout compilation. While a natural pattern for matrix A would expect three levels of nesting, Halide un-nests it during simplification, resulting in two shallower terms. For matrix B, Halide's simplifier has converted a load of a broadcast index to broadcasting the loaded value instead, since this is cheaper. In other words, the local optimization performed by Halide's simplifier obscures the computation pattern, making tensor computation less recognizable. This is an instance of the compiler phase-ordering problem.

Halide's existing support for AMX accelerator implements complex logic to cover each individual post-simplification pattern for programs of interest. Its pattern matcher is thus obscure and not scalable.

HARDBOILED solves this problem by introducing another set of axiomatic rules. For instance, to push `broadcast` inside `load`, it defines the following egglog rule:

```
(rewrite
  (Broadcast (Load type name index) lanes)
  (Load (MultiplyLanes type lanes) name
        (Broadcast index lanes)))
```

`MultiplyLanes` computes the correct type for the new load, and it will be rewritten to a concrete type via supporting rules like the following:

```
(rewrite (MultiplyLanes (Float32 l) x)
         (Float32 (* l x))
```

```
matmul[ramp(0, 1, 512)] = mem_to_amx( x512(0.f) );
matmul[ramp(0, 1, 512)] = mem_to_amx(
    (float32x512)vector_reduce_add(
        (cast<float32x8192>(A[ramp(x512(0), x512(32), 16) + x256(ramp(0, 1, 32))])
        * x16(cast<float32x256>(B[ramp(ramp(0, 16, 32), x32(1), 16)])))
    ) + matmul[ramp(0, 1, 512)] );
matmul_wrapper[ramp(0, 1, 512)] = amx_to_mem( matmul[ramp(0, 1, 512)] );
```

Fig. 3: Halide IR of the example MatMul before equality saturation. Highlighted in purple are the data movement nodes HARDBOILED has injected. The three statements can be mapped to AMX instructions `tile_zero`, `tile_matmul`, and `tile_load`, respectively.

EqSat automatically finds the right order to apply these rules so that our MatMul pattern gets matched.

Similarly, HARDBOILED has a set of axioms between nested `Ramp`/`Broadcast` expressions and their unnests. Thanks to EqSat, we can express target applications using principled patterns and, independent of application patterns, define a set of axiomatic rules, without worrying about optimizations interfering with one another.

C. Lowering MatMul to Hardware

So far we have successfully recognized the MatMul pattern, but there is a caveat. AMX expects matrix B to be in the VNNI format, a specific data layout that groups B's rows by 2 and interleaves elements from rows in a group. Indeed, to use AMX for our MatMul example requires data shuffling:

```
B_vnni(x%2, y, x/2) = B(y, x);
```

Halide's existing AMX support assumes the data is already stored in the VNNI format, so it actually cannot lower our MatMul example. In fact, the AMX applications it supports must have the following form:

```
mm(y, x) += cast<float>(A(r, x))
          * cast<float>(B_vnni(r%2, y, r/2));
```

Yet there are many applications that do not directly fit this pattern. As a result, application developers need to rewrite their computation to account for accelerator-specific data layouts and swizzling, failing Halide's promises of decoupling algorithms and schedules.

HARDBOILED solves this problem with a set of application-specific rules. Application-specific rules encode domain knowledge about MatMul-mappable optimization patterns. To support different layouts for matrix B, HARDBOILED uses a binary predicate relation `amx-B-tile`.

```
(relation amx-B-tile (Expr Expr))
```

The first argument to `amx-B-tile` is an AMX-mappable matrix, and the second argument is an expression that loads this matrix to AMX tile registers (with potential shuffling). The full rule for discovering matrices in the standard layout as AMX-mappable is shown in the Appendix (first rule, Figure 10b [33]). In its query, this rule looks for matrix B's loading pattern in the standard layout. For each matched loading pattern, the rule runs its action that (1) constructs an expression that shuffles B matrix, stores the result in a temporary buffer, and loads the buffer to memory, and (2) declares a fact of the predicate relation `amx-B-tile` that the constructed matrix holds the same data as the original matrix but in the desired layout.

Finally, HARDBOILED uses lowering rules to emit AMX instructions. The full rule is shown in the Appendix (first rule, Figure 10a [33]). It looks for expressions of the form $C + \sum A \times B$, queries relations `amx-A-tile` and `amx-B-tile` for matrices A and B in the expected layout, and constructs an equivalent MatMul expression that uses AMX.

D. Discussion

So far, we have shown an end-to-end walkthrough of HARDBOILED's flexible support of tensor operation patterns via EqSat. A more detailed description of the intermediate representation and rules is presented in the Appendix [33]. In this section, we discuss several practical considerations in the implementation of HARDBOILED.

1) Supporting Nvidia Tensor Cores: Besides Intel AMX, HARDBOILED also supports Warp-level Matrix Multiply-Accumulate (WMMA) instructions of Nvidia Tensor Cores. Different from CPUs, Nvidia's CUDA programming model organizes computations into GPU blocks and threads, and a WMMA instruction executes synchronously across a warp of 32 threads. To support WMMA instructions, the tile extractor wraps each WMMA-related statement with a warp-level parallel loop `for_gpu_lanes (thread_id_x, 0, 32) { }`. HARDBOILED relies on Halide's `FuseGPUThreadLoop` pass to fuse multiple warp-level loops into a single loop.

Under WMMA's semantics each thread holds a fragment of the tile, while Halide by default stores allocations outside the warp-level execution, so it is shared across threads in a warp. Therefore, HARDBOILED also pushes the WMMA buffer allocation inside the warp-level execution and scales down the allocation size to be per thread.

2) Rule Schedule: A *rule schedule* in egglog controls when and how often rules run. Currently, HARDBOILED uses a simple rule schedule, where it runs a fixed number of iterations of axiomatic, app.-specific, and lowering rules, and between each iteration, runs supporting rules to fixpoints. This is because supporting rules like type analysis always saturate in finite steps.

3) Cost Model: After equality saturation, egglog needs to extract the optimal program from the E-graph, which requires

a cost model. Unlike previous work on auto-scheduling [36, 8], which use sophisticated cost models, our cost model is simple, AST-size based. This is sufficient because, in our setting, the schedule is explicitly provided by the developer (e.g., with primitive `store_in(AMXTile)`). This makes instruction selection a hit-or-miss problem.

4) Supporting More Flexible Patterns in HARDBOILED*:* Thanks to the rule-based design, it is easy to extend HARDBOILED with support for new lowering patterns, and with EqSat, we do not have to worry about the interaction of rules leading to the phase-ordering problem.

HARDBOILED has implemented two categories of rules that lower to hardware: MatMul-like rules, which support both the standard and VNNI layouts, and convolution-like rules, which use a generalized Toeplitz transformation (Section V-A) to lower convolution-like patterns, including single-channel convolution, upsampling and downsampling, to MatMul hardware intrinsics. We found these two categories of rule sufficient to cover a range of unconventional workloads, like those considered in this paper.

5) Implementing Tensor Accelerator Kernels Using HARDBOILED*:* With HARDBOILED's flexible pattern support, users often do not need to contort their Halide programs (the algorithms) into MatMul patterns that are in direct correspondence to the hardware's capability; instead, they can express the algorithms in their natural forms. Users then update schedules to be accelerator-efficient and use the scheduling primitive `store_in` to determine which part of the computation to run on tensor accelerators.

All HARDBOILED schedules in the end-to-end case studies, except resampling (Section V-C), are implemented in this way. Expressing resampling as linear matrix operations requires global transformations, while, as an instruction selector, HARDBOILED only emits local, vector-lane-level shuffling instructions. Interestingly, re-expressing resampling in a form more amenable to tensor accelerators also provides speedups on non-tensor hardware by increasing arithmetic intensity.

IV. EVALUATION ON ML WORKLOADS

Before diving into applications that do not typically use tensor accelerators, we first evaluate HARDBOILED on traditional ML workloads as a sanity check of our system. In particular, we focus robustness and performance.

For performance measurements on Tensor Cores, we use an AWS EC2 p4de.24xlarge instance with Nvidia A100 80GB SXM[2]. Since we did not have access to a CPU that supports Intel's AMX, we used the Intel Software Development Emulator (version 9.38) to validate the correctness of HARDBOILED-generated code, focusing on functional correctness rather than performance.

Robustness: We use a collection of MatMul implementations from Intel's Optimization Reference Manual [10, Chapter 20.5.5] and reimplement them as HARDBOILED schedules. Table I summarizes each schedule and its compatibility with HARDBOILED. The reference manual assumes inputs are stored in the VNNI format expected by AMX instructions and that elements are pre-swizzled. In addition to evaluating this layout, we also test the more conventional "standard" layout, where operands are stored in row-major order. In this setting, HARDBOILED discovers when swizzling is required and injects the necessary swizzling code without user annotations or schedule changes.

Across the VNNI-format schedules, HARDBOILED supports all cases listed in the manual except software pipelining. Software pipelining requires fine-grained interleaving of load/store and compute instructions that cannot be expressed in Halide's scheduling model. Under the standard layout, HARDBOILED supports all from VNNI except the one that preloads matrix B. In this case, it is ambiguous whether the preloaded data should be swizzled.

Performance: To validate that our system produces high-quality code for standard workloads, we benchmark three canonical operators: matrix multiplication, convolutional layer, and scaled dot-product attention. For each workload we compare against strong Tensor Core baselines and Halide implementations that target CUDA without Tensor Cores. Our goal here is not to exhaustively tune Halide schedules or to outperform vendor libraries, but rather to demonstrate that HARDBOILED can work with provided Halide schedules to yield competitive performance on well-understood problems. The results are summarized in Figure 4.

For reference, we have also added the optimal runtime for each benchmark based on A100 Tensor Core's theoretical TFLOPs (156 TFMAs for FP16) and bandwidth (2 TB/s) [13]. Each bar is labeled with either (C) or (M) to indicate whether it is compute-bound or memory-bound.

We evaluated GEMM on 1024×1024 matrices. Our baselines are `cuBLASLtMatmul`, NVIDIA's optimized GEMM routine (v12.6.4), and a CUDA-only schedule from Halide's repository[3]. Our Tensor Core-enabled schedule reduced runtime from 223 us to 66 us, a $3.36\times$ speedup. Our generated code is about $1.5\times$ slower compared to `cuBLASLtMatmul`.

We next benchmarked a standard convolutional layer with bias and activation, matching the forward-propagation primitive commonly used in neural networks. We compare against two vendor baselines: `torch::nn::Conv2d` (PyTorch v2.5.1), and `cudnnConvolutionBiasActivationForward` (Nvidia's cuDNN v9.1.0), as well as a CUDA-only Halide schedule drawn from Halide's repository[4]. On input size $4096\times64\times64\times16$, our Tensor Core schedule achieved 1.05 ms, outperforming PyTorch by $3.7\times$, and $1.6\times$ faster than

[2]We use a data center GPU here as it is representative of large-scale ML workloads. In Section V, we evaluate our image processing case studies using a desktop with an RTX 4070 SUPER, since image processing workloads are typically run on consumer hardware.

[3]The official Halide repository has a set of manually-tuned example applications. The Halide GEMM implementation is available at https://github.com/halide/Halide/tree/main/apps/cuda_mat_mul.

[4]The implementation is available at https://github.com/halide/Halide/tree/main/apps/conv_layer and only supports 32 channels and beyond.

cuDNN. On input size 4096×64×64×32, our schedule ran in 5.34 ms, making it 3× faster than the CUDA-only baseline, 1.2× faster than PyTorch and 1.8× slower than cuDNN.

Finally, we benchmarked scaled dot-product attention, which consists of three stages: (1) computing QK^T and scaling, (2) applying a softmax across keys, and (3) multiplying the resulting probabilities with V. We considered two vendor baselines: a PyTorch implementation using `torch.nn.functional` primitives, and a composed version combining cuBLAS for MatMuls, cuDNN softmax for normalization, and a lightweight custom CUDA kernel for scaling. We do not compare against highly optimized kernels such as FlashAttention, since they incorporate algorithmic and data layout optimizations that are orthogonal to Tensor Core uses; our Halide implementation follows the naïve attention algorithm for an apples-to-apples comparison, focusing on proper Tensor Core utilization.

On input size $N = 64$, $L = 4096$, $D = 64$, our Tensor Core implementation achieved 27.8 ms, outperforming the PyTorch baseline by 1.2×, while trailing the composed cuBLAS+cuDNN baseline by about 1.3×.

These results demonstrate that our system can produce hardware-accelerated kernels that are competitive with vendor baselines. We emphasize that our goal is not to beat cuBLAS or cuDNN but to show that our support for tensor accelerators is effective.

V. Case Study

Using Halide equipped with HardBoiled for tensor instruction selection, we are able to map many classical operations in signal and image processing to tensor accelerators. In this section, we present a series of case studies, focusing on mapping to the Nvidia Tensor Cores in a commodity desktop GPU, the GeForce RTX 4070 SUPER. We study the following applications: ① 1D/2D convolution, ② upsampling by 2, ③ downsampling by 2, ④ resampling by a non-integer factor, ⑤ recursive filtering, and ⑥ DCT-based denoising. Source code for all case studies is available online[5].

Among them, the first three applications are micro benchmarks that can be reformulated as linear matrix operations via local transformations on the kernel matrix, so HardBoiled emits shuffling instructions to transform it into Tensor Cores MatMul instructions.

The last three applications are end-to-end applications that can benefit from the use of tensor accelerators for small matrix multiplications or convolutions in key places within larger pipelines. In the interest of space we will describe these at a higher level. Readers are encouraged to see the source code for the differences between the CUDA-only schedules and the tensor core schedules.

In addition to proving the generality and usefulness of HardBoiled, our main research question in these case studies is whether tensor accelerators can provide meaningful speedups for these kinds of applications, so our primary baselines will be our best-effort optimized Halide CUDA schedules that do not use Tensor Cores.

[5]https://github.com/yihozhang/cgo2026-hardboiled-artifact

A. Convolution

We first consider 1D convolution of the form $O(x) = \sum_{0 \le r_x < l} I(x + r_x) \cdot K(r_x)$ and 2D convolution of the form $O(x, y) = \sum_{0 \le r_x < l, 0 \le r_y < m} I(x + r_x, y + r_y) \cdot K(r_x, r_y)$, where the kernel K has size l and $m \times l$ respectively. Unlike multi-channel convolutions used in convolutional layers, they cannot be mapped to MatMul by e.g., im2col [4], which would degenerate to a matrix-vector multiplication.

Let us first review how to reformulate 1D convolutions into linear operations. Consider values of O at positions $(x, y), \ldots, (x, y + k)$. It can be computed as

$$\big[O(x), \ldots, O(x + k - 1)\big] = \big[I(x), \ldots, I(x + k + l - 1)\big] \times A_K$$

where A_K is a $k \times (k + l)$ Toeplitz matrix defined by

$$A_K(x, y) = \begin{cases} K(y - x) & \text{if } 0 \le y - x < l \\ 0 & \text{otherwise} \end{cases}$$

To turn this vector-matrix multiply into a matrix-matrix multiply, we can load h (overlapped) vectors of length $k + l$ from I:

$$\begin{bmatrix} O(x) & \ldots & O(x+k-1) \\ O(x+k) & \ldots & O(x+2k-1) \\ & \vdots & \\ O(x+(h-1)k) & \ldots & O(x+hk-1) \end{bmatrix} = \begin{bmatrix} I(x) & \ldots & I(x+k+l-1) \\ I(x+2k) & \ldots & I(x+2k+l-1) \\ & \vdots & \\ I(x+(h-1)k) & \ldots & I(x+hk+l-1) \end{bmatrix} \tag{1}$$

$$\times A_K \tag{2}$$

HardBoiled automatically constructs the Toeplitz matrix for suitable patterns, so 1D convolution can be naturally expressed as the following Halide algorithm:

```
ImageParam K(Float(16), 1), I(Float(16), 1);
RDom rx;
conv(x) = 0.f;
conv(x) += cast<float>(K(rx))
           * cast<float>(I(x + rx));
output(x) = conv(x);
```

Both the kernel and the input are in `float16`, and the output is in `float32`, a layout that WMMA instructions on Tensor Cores support.

To direct HardBoiled to lower the convolution to Tensor Cores, the user writes a schedule that divides the computation into 256-pixel segments and a reduction axis of 8, and requests each segment to be computed on WMMA accumulator registers. In code, this looks as follows:

```
output.split(x, x, xi, 256)
      .vectorize(xi);
conv.compute_at(output, x)
    .store_in(WMMAAccumulator)
    .split(x, x, xi, 256)
    .vectorize(xi);
conv.update()
    .split(x, x, xi, 256).split(rx, rx, rxi, 8)
    .reorder(rxi, xi, rx, x)
    .atomic().vectorize(xi).vectorize(rxi);
```

The computation of each segment is equivalent to multiplying a 16×32 input matrix consisting of 32 (overlapped) vectors with an 8×16 kernel matrix, producing an 32×8 tile as output. This maps to WMMA's $32 \times 16 \times 8$ MatMul. More details

TABLE I: Support for MatMul schedules from Intel's Optimization Reference Manual in two layouts.

Implementation	VNNI	Standard
Reference impl.	✓	✓
Loop reordering	✓	✓
Preloading matrix A	✓	✓
Preloading matrix B	✓	×
Software pipelining	×	×

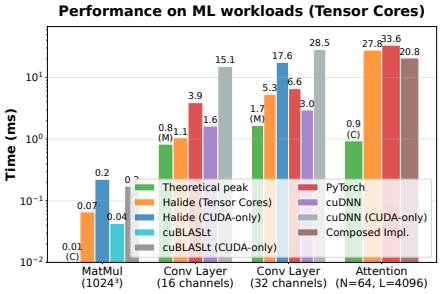

Fig. 4: Performance comparison on ML workloads.

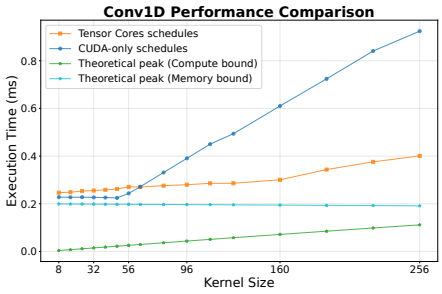

Fig. 5: Performance comparison on 1D convolution.

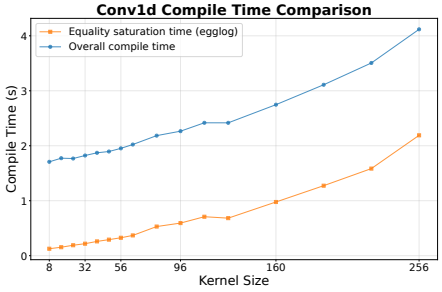

Fig. 6: Kernel compilation time of 1D convolution.

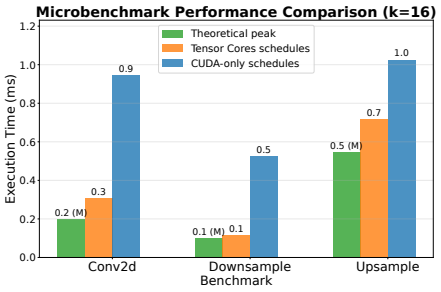

Fig. 7: Performance comparison on microbenchmarks with kernel size 16.

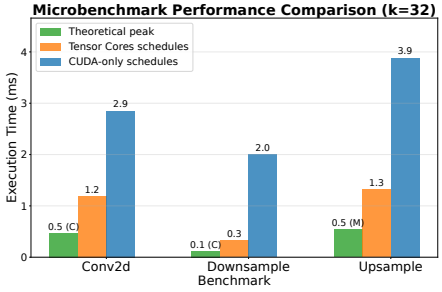

Fig. 8: Performance comparison on microbenchmarks with kernel size 32.

on how HARDBOILED maps this 1D convolution pattern to hardware can be found in Appendix B [33].

To tensorize a 2D convolution, we can parametrize over one reduction axis, which reduces the problem to 1D convolutions: define function $O_{r_y}(x, y)$ parametrized by r_y as $\sum_{0 \le r_x < l} I(x + r_x, y + r_y) \cdot K(r_x, r_y)$ where r_y and $y + r_y$ are effectively constants, and $O(x, y) = \sum_{0 \le r_y < m} O_{r_y}(x, y)$. This parametrization step, when reflected in Halide schedules, is equivalent to leaving r_y as a serial outer loop.

These schedules produce inner loops that perform two matrix loads, one matrix multiply, and one matrix store. To achieve a higher arithmetic intensity, we can additionally unroll along any axis on which one of the operands depends but the other does not. For convolution on an image, we unroll over multiple rows of the output to amortize the cost of loading the convolution kernel, which does not vary along that axis.

Figure 5 shows the result for 1D convolution, comparing a manually tuned Halide CUDA schedule. Starting at around $k = 64$, the CUDA-only schedule transitioned from bandwidth-limited to compute-limited, while the Tensor Core schedule remained bandwidth-limited, culminating in a speedup of 2.3× at $k = 256$. For 2D convolution, the Tensor Core schedule achieves a 3.1× speedup at $k = 16$ (Figure 7) and 2.4× speedup at $k = 32$ (Figure 8).

The figures also show the theoretical performance limits based on RTX 4070 SUPER's theoretical TFLOPs and bandwidth[6]. This is computed using the algorithmic description of the benchmark and is oblivious to the redundant and unnecessary computations introduced by the use of accelerators (such as Toeplitz transformations)[7], to avoid being dependent on the specific schedules and transformations used. As a result, even schedules that use Tensor Cores perfectly cannot reach the performance limit shown in the figures.

We also measured run time spent in equality saturation for 1D convolution. Since we unroll along the reduction dimension, larger kernel sizes mean longer programs. Unlike many EqSat applications that suffer from blowups, HARDBOILED exhibits a manageable run time because (1) individual tensorized statements are usually small and (2) HARDBOILED's search for tensor-leveraging opportunities is guided by user-provided schedules and requires a smaller search space than traditional super-optimizers.

B. *Downsampling and Upsampling by an Integer Factor*

We consider the task of downsampling (strided convolution) by a factor of 2:

$$\sum_{r_x, r_y} I(2x + r_x, 2y + r_y) \cdot K(r_x, r_y).$$

[6] For RTX 4070 SUPER, we use a peak theoretical Tensor Core compute of 36 TFMAs, obtained by scaling the numbers advertised for RTX 4090 [14] proportionately to the reduced number of Tensor Cores, since they have the same Tensor Core specification. The peak bandwidth of RTX 4070 SUPER is advertised to be 504.2 GB/s.

[7] For example, the theoretically optimal amount of compute for 1D convolution is calculated as $(4096 - k) \cdot 4096 \cdot k$ for kernel size k, and the optimal IO is calculated as the sum of input size and output size.

By parameterizing over r_y via loop reordering, we are left with downsampling along one axis: $\sum_{r_x} I(2x + r_x, y_1) \cdot K(r_x, y_2)$. This requires constructing the following Toeplitz-like coefficient matrix of size $k \times (2k + l)$:

$$A_K^{down}(x, y) = \begin{cases} K(y - 2x) & \text{if } 0 \le y - 2x < l \\ 0 \end{cases}$$

On the other hand, upsampling by a factor of 2 has the following form:

$$O(x, y) = \sum_{r_x, r_y} I\left(\left\lfloor \frac{x}{2} \right\rfloor + r_x, \left\lfloor \frac{y}{2} \right\rfloor + r_y\right) \cdot K(2r_x + x\%2, 2r_y + y\%2)$$

We can simplify the indexing by separating it into phases and treating this as a multiphase filter:

$$K^{phase}(x, y, d_x, d_y) = K(2x + d_x, 2y + d_y)$$
$$O^{phase}(x, y, d_x, d_y) = \sum_{r_x, r_y} I(x + r_x, y + r_y) \cdot K^{phase}(r_x, r_y, d_x, d_y)$$
$$O(x, y) = O^{phase}\left(\left\lfloor \frac{x}{2} \right\rfloor, \left\lfloor \frac{y}{2} \right\rfloor, x\%2, y\%2\right)$$

Note that O^{phase} is already in the form of a convolution that HARDBOILED supports, but this would be memory inefficient since the output is stored in an interleaved form in O^{phase}. To be more cache-friendly, we store O^{phase} compactly by reordering the storage of d_x, d_y as the innermost dimensions:

```
O_phase.reorder_storage(dx, dy, x, y);
```

An $h \times k \times 2 \times 2$ tile of O^{phase} with a reduction axis of length l can be computed as

$$\begin{bmatrix} O^{phase}(x, y, 0, 0) & \dots & O^{phase}(x + k - 1, y, 1, 1) \\ O^{phase}(x + k, y, 0, 0) & \dots & O^{phase}(x + 2k - 1, y, 1, 1) \\ & \vdots & \\ O^{phase}(x + (h-1)k, y, 0, 0) & \dots & O^{phase}(x + hk - 1, y, 1, 1) \end{bmatrix}$$
$$= \begin{bmatrix} I(x, y) & \dots & I(x + \lfloor \frac{k}{4} \rfloor + l - 1, y) \\ I(x + 2k, y) & \dots & I(x + 2\lfloor \frac{k}{4} \rfloor + l - 1, y) \\ \vdots & & \\ I(x + (h-1)k, y) & \dots & I(x + h\lfloor \frac{k}{4} \rfloor + l + l - 1, y) \end{bmatrix} \times A_{K,y}^{up}$$

where $A_{K,w}^{up}$ is a $\lfloor \frac{k}{4} \rfloor \times \left(\lfloor \frac{k}{4} \rfloor + l\right)$ matrix:

$$A_{K,w}^{up}(x, y) = \begin{cases} K^{phase}(y - \lfloor \frac{x}{4} \rfloor, w, x\%2, \lfloor \frac{x}{2} \rfloor \%2) \\ \qquad\qquad \text{if } 0 \le y - \lfloor \frac{x}{4} \rfloor < l \\ 0 \quad \text{otherwise} \end{cases}$$

HARDBOILED recognizes both downsampling and upsampling patterns, where it would generate shuffling instructions that construct the matrix A_K^{down} and $A_{K,w}^{up}$.

As in convolution, we added a small amount of additional unrolling to increase the number of matrix multiply instructions per matrix load and store instruction. This results in our downsample and upsample implementations, which achieve 4.6× and 1.4× speedups respectively at $k = 16$, and 6.1× and 2.9× speedups at $k = 32$.

C. Resampling by a Non-Integer Factor

In some settings, it is necessary to quickly resample an image down to a smaller fixed size, regardless of its original size. This happens when generating thumbnails for a gallery, or when preparing inputs to an ML model trained at a fixed size. Unlike the integer-factor resizing above, there is no single filter bank that is used per output pixel. Mathematically, resampling is done by convolving the high-resolution samples with a continuous pre-filter to make a continuous signal, then sampling that continuous signal at a lower rate. The pre-filter is sized according to the output sampling rate and designed to reject any frequencies that cannot be represented at reduced resolution, so its footprint on the input can be quite large. A popular choice for a high-quality pre-filter is a three-lobed Lanczos [26], defined as $sinc(x)sinc(x/3)$ for $x \in [-3, 3]$, where $sinc(x) = sin(\pi x)/(\pi x)$.

TABLE II: Runtimes for resizing a 2048×2048 image down by a non-integer factor using a three-lobed Lanczos pre-filter. The geomean speedup is 1.47×.

Output size	CUDA-only (us)	Tensor core (us)
143×143	111	79
245×245	110	73
450×450	113	74
921×921	145	102

Evaluating this Lanczos filter repeatedly is expensive. Fortunately, resizing can be separated into vertical followed by horizontal resizing. When resizing an image vertically, each column undergoes the same linear transformation. This linear transformation can be precomputed as a single sparse matrix applied to all columns, amortizing the cost of evaluating the Lanczos kernel across columns. The same can be done for horizontal resizing. This sparse matrix is rectangular, with a non-zero band running down the diagonal with a known bounded width. An easy representation for this sparse matrix is the index of the first non-zero followed by the non-zero entries, and this is the one used by the `resize` application in Halide's repository[8].

However, we can also treat this matrix as block-sparse: we can give entire groups of rows the same starting index, and expand the width of the band as necessary. This introduces some unnecessary multiplications by zero. Still, it is an overall win even without tensor accelerators because it lets us tile the computation in a memory-efficient way, reducing the number of repeated loads of the same input values. We found aligning groups of 16 rows in this way sped up the original `resize` application by about 3×.

We modified the `resize` application to be block-sparse in this way, further optimized its CUDA schedule, and benchmarked resampling a four-megapixel RGB image by several non-integer factors. Results are shown in Table II under the column "CUDA-only". These kernels achieve high utilization of both compute and memory bandwidth, both ranging between 60% and 90% depending on the size and resampling direction.

We then changed the schedule to perform the small matrix multiplication stages on Tensor Cores. Adding compute using Tensor Cores makes the kernels bandwidth-limited. This schedule achieves only 10% utilization of the Tensor Cores, but it is enough to yield a geometric mean overall speedup of 1.47× across the output sizes tested.

[8]https://github.com/halide/Halide/tree/main/apps/resize

This application demonstrates that low-arithmetic-intensity workloads such as resizing images can still benefit from Tensor Cores, despite using them at a fraction of their peak throughput.

D. Recursive Filtering

In audio processing, recursive filters are used as a core primitive for a variety of effects. A simplified second-order recursive filter on an input signal X can be expressed as:

$$Y_t = X_t + \alpha Y_{t-1} + \beta Y_{t-2}$$

Each output is a linear combination of the next input and the previous two outputs. A direct implementation is necessarily entirely serial. Audio signals may have multiple channels that can be processed in parallel, but for stereo audio this number is merely two. Fortunately, a variety of techniques exist for transforming recursive filters into forms that admit parallel computation. First, they can be computed in parallel tiles using the method of Hoppe et al. [12]. This technique tiles the domain and computes the original filter on each tile in parallel. As additional passes, it then fixes up each tile using the contribution from previous tiles. This provides a parallel outer loop.

We can also achieve a parallel inner loop using a technique called Scattered-lookahead interpolation (SLA) by Parhi et al. [15]. Given a dilation factor d, it gives a recipe for decomposing the recursive filter into a non-recursive convolution of size $2d + 1$ followed by a dilated recursive filter of the form:

$$Y_t = X_t + \alpha' Y_{t-d} + \beta' Y_{t-2d}$$

In this form, the recursive step can operate in parallel over groups of d outputs. We found that a combination of these two techniques was most effective for implementing a recursive filter on GPU, with inter-block parallelism provided by the first technique, and intra-block parallelism provided by the second. Transformed in this way, our optimized Halide CUDA implementation takes 67.5 us to process 2^{21} stereo samples (about 50 seconds of audio). The runtime is roughly two thirds in the recursive step and one third in the fix-up step required by Hoppe et al. The convolution is fused into the recursive step. The fix-up step is limited by global memory bandwidth, achieving 90% of peak, while the recursive step is limited by bandwidth from L1, achieving 92% of peak. We found performance was maximized at $d = 8$ and tiles of size 1024.

Without changing the algorithm, we changed the Halide schedule to use Tensor Cores for the convolution: we vectorize across the reduction in the convolution and across groups of 256 outputs, and place the results in shared memory to be immediately used by the recursive step. HARDBOILED can recognize the convolution in this form, similar to Section V-A, and constructs a Toeplitz matrix for our convolution kernel ahead of time. This schedule reduced overall runtime to 58 us, with all the savings coming from the recursive step (the fix-up kernel is unchanged). As the recursive kernel was L1-bandwidth-limited, the speedup is not due to harnessing additional compute, but rather because the Tensor Core schedule is easier on the memory subsystem. The Tensor Cores are at a mere 8% utilization.

This application demonstrates how ad hoc usage of Tensor Cores inside non-ML pipelines can provide meaningful speedups, despite only using a small fraction of their peak capabilities.

E. DCT-Based Denoising

A traditional approach to reduce noise in images is transform-domain coring. An image is divided into overlapping tiles, with a soft windowing function applied. Each tile then undergoes some linear transformation designed to concentrate the signal into a few bins while leaving the noise spread across many bins. Any bin with a small magnitude is then set to zero. The transform is then inverted, and overlapping tiles are blended into a single output. In our case, we will use a discrete cosine transform (DCT), in line with the most notable use of transform-domain coring—the JPEG file format [29]. We will use both a direct DCT, and the fast DCT described by Plonka et al. [16]. The tile size is set to be 16×16.

For the direct variant, the DCT is performed by treating each tile as a 16×16 matrix, and multiplying it by the DCT matrix to transform down each column. The result is transposed, and multiplied by the DCT matrix again to transform along each row. After coring, the same is done in reverse along rows and columns using the inverse DCT matrix, resulting in four MatMuls per tile. These are all fused into a single kernel. The fast DCT variant has the same structure, but the MatMuls are replaced by a fully unrolled 16-point fast DCT run along the rows or columns. After the transform kernel, a second blending kernel is launched to blend overlapping output tiles and undo the windowing.

Our optimized Halide CUDA implementation takes 277 us and 76 us to denoise a one megapixel three-channel image using the direct and fast approaches respectively. Focusing on the fast variant, time is split roughly equally between the transform kernel and the blending kernel. The transform kernel is balanced, achieving 75% of peak compute and 77% of peak memory bandwidth, while the blending kernel is entirely bandwidth-limited.

We rescheduled the direct variant to use Tensor Cores for the MatMuls. This yields a total runtime of 68 us. By using Tensor Cores, the brute-force DCT is four times faster than using CUDA cores alone, and is about 10% faster than the fast DCT despite doing $3.6\times$ more floating-point operations. The transform kernel is bandwidth-limited, achieving 85% of peak bandwidth from DRAM and 88% of peak bandwidth from L1. Tensor Core utilization is 28%.

This transform kernel fuses four matrix multiplications with a non-linear operation (the coring) in between, yet it is still bandwidth-limited. If implemented using separate calls to a library to perform the matrix multiplication steps, this fusion would be lost, and the runtime would scale up according to the number of kernel launches required. Being able to ad hoc schedule individual pipeline stages onto Tensor Cores in a fused kernel is more efficient than having to break out of Halide and call a library routine as a separate kernel launch.

VI. Related Work

A. Program Optimization Using Equality Saturation

Several works have been proposed on targeting domain-specific accelerators with EqSat. For example, Diospyros [28] and JOKer [35] use EqSat-based rewrite rules to select vector instructions for digital signal processors (DSPs). The most relevant to us is Glenside [23, 9], which uses low-level rewrite rules to map tensor programs to DL accelerators. Both Glenside and HardBoiled work at the instruction selection level, but Glenside is a standalone project, while HardBoiled works with Halide. The integration with Halide allows users to develop hardware-accelerated kernels in a high-level scheduling DSL.

Similar to HardBoiled, which considers how to retrofit EqSat to existing compilers, DialEgg[32] integrates egglog with MLIR in a dialect-agnostic way. Caviar [11] boosts the optimizing rewrite rules in Halide with EqSat and shows it can prove pre-conditions that Halide cannot. LIAR [27] rewrites program patterns in a minimalistic DSL into hardware-accelerated instructions using EqSat.

B. Compilers for Hardware Accelerators

HalideTCU [22] was an independent attempt to add Tensor Cores support to Halide. It added a high-level `tensor_core` method to the IR that just passes parameters to WMMA instructions, but does not support scheduling. Many other frameworks support Tensor Cores, especially within the ML programming community. TVM [5] exposes Tensor Core intrinsics but requires the developer to manually allocate memory in the correct format and write adapter methods to target the intrinsics directly [7]. TileLang [30] extends TVM with high-level abstractions for MatMul and other common ML operations, but it still requires the developer to allocate memory in the correct format and to manually copy data around. Triton [25] is a Python DSL for developing GPU kernels and provides a library of primitives that map to hand-written CUDA code. When the library does not match their needs, developers have to write their own intrinsic (e.g., to fuse data movement and computation). The RISE language for rewrite-based performance tuning [21] was recently extended to target Tensor Cores.

Recent work like Ansor and TensorIR has focused on auto-scheduling for user-schedulable languages. TensorIR [8] is especially relevant to us, since it targets Tensor Cores and applies a similar tile abstraction to TVM's IR. HardBoiled is different from this work in several aspects: first, auto-scheduling aims to find good schedules that automatically map high-level workloads onto tensor cores. In contrast, we want to enable tensor core usage from human-written schedules, which means HardBoiled must correctly map code onto tensor cores in precisely the way it is told to by a human, rather than making a best effort at finding *any* good mapping to tensor instructions. The additional constraints make the problem easier in some respects and harder in others. Moreover, work like TensorIR focuses on traditional ML tasks such as GEMM and convolution layers, while we focus on workloads that may leverage tensor accelerators but have not done so (potentially due to a lack of suitable tooling).

HardBoiled can be viewed as solving the lifting problem from lower-level tensor operations to high-level primitives like MatMul. Tensorize [3] uses symbolic traces to lift algorithms in legacy codebases to high-level tensor programs. Rake [1] improves instruction selection for DSPs by first lifting Halide IR to a higher-level Uber-Instruction IR, and Pitchfork [19] takes a similar lift-then-lower approach but uses offline rewrite rule synthesis.

VII. Conclusion

In this work, we add support for tensor accelerators to the Halide scheduling language in the HardBoiled system. HardBoiled gives Halide users fine-grained direct control over how portable algorithm code is mapped to tensor accelerators in a way that composes cleanly with the existing scheduling language. It builds on Halide's existing vector IR, and uses equality saturation to select tensor instructions from Halide expressions that have been vectorized along multiple axes at once. We have shown that HardBoiled enables ad hoc use of tensor accelerators in signal processing applications that have not traditionally used them, and that this provides meaningful speedups.

Acknowledgment

We would like to thank the anonymous reviewers and shepherd for their valuable feedback and to friends who share their insights during this work, including Shoaib Kamil, Alex Reinking, Zachary Tatlock, and Zihao Ye. This material is based upon work supported by the National Science Foundation under Grant No. 2232339 and 2312195.

Data Availability

Additional data related to this publication may be found in the repository at https://doi.org/10.5281/zenodo.17810573 [2].

References

[1] Maaz Bin Safeer Ahmad, Alexander J. Root, Andrew Adams, Shoaib Kamil, and Alvin Cheung. Vector instruction selection for digital signal processors using program synthesis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 1004–1016, New York, NY, USA, 2022. Association for Computing Machinery.

[2] Maaz Bin Safeer Ahmad, Yihong Zhang, Andrew Adams, and Derek Gerstmann. Artifact of cgo 2026 "pushing tensor accelerators beyond matmul in a user-schedulable language", December 2025.

[3] Alexander Brauckmann, Luc Jaulmes, José W de Souza Magalhães, Elizabeth Polgreen, and Michael FP O'Boyle. Tensorize: Fast synthesis of tensor programs from legacy code using symbolic tracing, sketching and solving. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, pages 15–30, 2025.

[4] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High Performance Convolutional Neural Networks for Document Processing. In Guy Lorette, editor, *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule (France), October 2006. Université de Rennes 1, Suvisoft. http://www.suvisoft.com.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 579–594, USA, 2018. USENIX Association.

[6] Frederik Engels. Add amx instructions with k dimension larger than 4 bytes. https://github.com/halide/Halide/pull/6582, August 2022. Pull request #6581, Halide.

[7] Siyuan Feng. How to optimize convolution using tensorcores? https://tvm.apache.org/docs/v0.15.0/how_to/optimize_operators/opt_conv_tensorcore.htm, 2025. [Accessed: 2025-05-23].

[8] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 804–817, New York, NY, USA, 2023. Association for Computing Machinery.

[9] Bo-Yuan Huang, Steven Lyubomirsky, Yi Li, Mike He, Gus Henry Smith, Thierry Tambe, Akash Gaonkar, Vishal Canumalla, Andrew Cheung, Gu-Yeon Wei, Aarti Gupta, Zachary Tatlock, and Sharad Malik. Application-level validation of accelerator designs using a formal software/hardware interface. *ACM Trans. Des. Autom. Electron. Syst.*, 29(2), February 2024.

[10] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, 2024. Published: 2024-05-22.

[11] Smail Kourta, Adel Abderahmane Namani, Fatima Benbouzid-Si Tayeb, Kim Hazelwood, Chris Cummins, Hugh Leather, and Riyadh Baghdadi. Caviar: an e-graph based trs for automatic code optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, CC 2022, page 54–64, New York, NY, USA, 2022. Association for Computing Machinery.

[12] Diego Nehab, André Maximo, Rodolfo S. Lima, and Hugues Hoppe. Gpu-efficient recursive filtering and summed-area tables. *ACM Trans. Graph.*, 30(6):1–12, December 2011.

[13] NVIDIA. NVIDIA A100 Tensor Core GPU Architecture. White paper, NVIDIA Corporation, 2020. Version 1.0.

[14] NVIDIA. NVIDIA Ada GPU Architecture. White paper, NVIDIA Corporation, 2022. Version 2.02, Updated 2023.

[15] K.K. Parhi and D.G. Messerschmitt. Pipeline interleaving and parallelism in recursive digital filters. i. pipelining using scattered look-ahead and decomposition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(7):1099–1117, 1989.

[16] Gerlind Plonka and Manfred Tasche. Fast and numerically stable algorithms for discrete cosine transforms. *Linear algebra and its applications*, 394:309–345, 2005.

[17] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Halide: decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, 61(1):106–115, December 2017.

[18] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, and Clark Barrett. Quantifier instantiation techniques for finite model finding in smt. In Maria Paola Bonacina, editor, *Automated Deduction – CADE-24*, pages 377–391, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[19] Alexander J Root, Maaz Bin Safeer Ahmad, Dillon Sharlet, Andrew Adams, Shoaib Kamil, and Jonathan Ragan-Kelley. Fast instruction selection for fast digital signal processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ASPLOS '23, page 125–137, New York, NY, USA, 2024. Association for Computing Machinery.

[20] Thales Sabino. Add support for amx instructions. https://github.com/halide/Halide/pull/5818, October 2021. Pull request #5818, Halide.

[21] Lukas Siefke, Bastian Köpcke, Sergei Gorlatch, and Michel Steuwer. Systematically extending a high-level code generator with support for tensor cores. In *Proceedings of the 14th Workshop on General Purpose Processing Using GPU*, GPGPU '22, New York, NY, USA, 2022. Association for Computing Machinery.

[22] Savvas Sioutas, Sander Stuijk, Twan Basten, Lou Somers, and Henk Corporaal. Programming tensor cores from an image processing dsl. In *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '20, page 36–41, New York, NY, USA, 2020. Association for Computing Machinery.

[23] Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. Pure tensor program rewriting via access patterns (representation pearl). In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, PLDI '21, page 21–31. ACM, June 2021.

[24] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, page 264–276, New York, NY, USA, 2009. Association for Computing Machinery.

[25] Philippe Tillet, H. T. Kung, and David Cox. Triton:

an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 10–19, New York, NY, USA, 2019. Association for Computing Machinery.

[26] Ken Turkowski. *Filters for common resampling tasks*, page 147–165. Academic Press Professional, Inc., USA, 1990.

[27] Jonathan Van der Cruysse and Christophe Dubach. Latent idiom recognition for a minimalist functional array language using equality saturation. In *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '24, page 270–282. IEEE Press, 2024.

[28] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. Vectorization for digital signal processors via equality saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 874–886, New York, NY, USA, 2021. Association for Computing Machinery.

[29] Gregory K Wallace. The jpeg still picture compression standard. *Communications of the ACM*, 34(4):30–44, 1991.

[30] Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. Tilelang: A composable tiled programming model for ai systems, 2025.

[31] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021.

[32] Abd-El-Aziz Zayed and Christophe Dubach. Dialegg: Dialect-agnostic mlir optimizer using equality saturation with egglog. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '25, page 271–283, New York, NY, USA, 2025. Association for Computing Machinery.

[33] Yihong Zhang, Derek Gerstmann, Andrew Adams, and Maaz Bin Safeer Ahmad. Pushing tensor accelerators beyond matmul in a user-schedulable language. https://arxiv.org/abs/2512.02371, 2025.

[34] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. Better together: Unifying datalog and equality saturation. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[35] Xiaolei Zhao, Zhaoyun Chen, Yang Shi, Mei Wen, and Chunyun Zhang. Automatic end-to-end joint optimization for kernel compilation on dsps. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2023.

[36] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

$$
\begin{aligned}
\text{Stmt } s &::= \texttt{store}(str, e_{idx}, e_{val}) \mid \texttt{eval}(e) \\
\text{Expr } e &::= \texttt{load}(str, t, e_{idx}) \mid \texttt{cast}(t, e) \mid \texttt{call}(f, \vec{e}) \\
&\quad \mid op(\vec{e}) \mid \texttt{ramp}(e_{ini}, e_{inc}, n) \mid \texttt{broadcast}(e, n) \\
&\quad \mid \texttt{vector\_reduce\_add}(n, e) \\
&\quad \mid \texttt{loc\_to\_loc}(l_1, l_2, e) \mid x \mid v \\
\text{Location } l &::= \texttt{Mem} \mid \texttt{AMX} \mid \texttt{WMMA} \\
\text{Type } t &::= \texttt{bfloat16}^n \mid \texttt{float32}^n \mid \cdots \\
\text{Arith. Op. } op &::= + \mid - \mid * \mid \ / \ \mid \cdots \\
\text{Intrinsics } f &::= \texttt{tile\_load} \mid \texttt{tile\_matmul} \mid \cdots \\
\text{Variable } x &::= a \mid b \mid \cdots \mid \texttt{ExprVar}(e) \\
\text{Value } v &\in [\![\texttt{bfloat16}]\!] \cup [\![\texttt{float32}]\!] \cup \cdots
\end{aligned}
$$

Fig. 9: Syntax of HARDBOILED's internal representation

APPENDIX

A. Implemenetation of HARDBOILED

Our IR is shown in Figure 9. A program is a sequence of statements. A `store` statement takes a buffer name, an index pattern, and a vector expression and stores the vector in the buffer according to the index. An `evaluate` statement evaluates a given expression and is useful for intrinsics with side effects, e.g., `tile_store`.

A `load` operator takes a buffer name, the result type, and an index pattern and does a vectorized load similar to `store`. A `cast` operator converts a value to a compatible type. A `call` operator describes a (tensor) intrinsic call, and arithmetic operators *op* are interpreted using standard arithmetic over numbers and pointwise over vectors.

Both `broadcast` and `ramp` expressions are primitives for building indexing vectors. Semantically, $\texttt{ramp}(e_{ini}, e_{inc}, n)$ concatenates vectors $e_{ini}, e_{ini} + e_{inc}, \ldots, e_{ini} + (n-1)e_{inc}$, and $\texttt{broadcast}(e, n)$ represents a concatenation of n copies of e. A `vector_reduce_add` operator[9] takes an aggregation factor n and a vector expression e with k lanes. It requires k to be divisible by n and partitions vector e into k subvectors, and sums within each subvector to derive an aggregated vector with k lanes. `loc_to_loc` marks data movement between memories and accelerator registers.

Besides standard variables, HARDBOILED also uses expression variables to represent temporary buffers. `ExprVar` takes a vector expression and denotes a pointer to a buffer that stores the result of evaluating that expression. HARDBOILED uses `ExprVar` to store swizzled matrices.

The heavy lifting of optimization is performed via rewrite rules in egglog. We classify rules into four categories: lowering rules, application-specific rules, axiomatic rules, and supporting rules.

1) Lowering rules: Lowering rules lower high-level tensor patterns to accelerator intrinsics. Figure 10a presents the lowering rules for MatMul and storing the results to memory. The lowering rule captures the MatMul pattern $AB + C$ while abstracting away the representation details with `amx-*-tile` relations. HARDBOILED uses additional application-specific rules to populate `amx-*-tile` relations (Section A2).

The lowering rule for AMX MatMul does not assume the result is already marked as being stored in AMX registers. Instead, it unions the original expression with `(AMX2Memnew-e)`, so even if the user does not indicate the result should be stored in AMX registers (e.g., intermediate computations that users do not get a handle to schedule), HARDBOILED may still leverage AMX when possible and load the result to memory. In the case the user has required the result to be stored in memory, HARDBOILED cancels out `AMX2Mem` with rule `(rewrite (Mem2AMX (AMX2Mem e)) e)`.

The second rule in Figure 10a eliminates `AMX2Mem` by looking for `store`s from AMX to memory and union them with a call to the `tile_store` intrinsics.

2) Application-specific rules: Application-specific rules extend and complement lowering rules with diverse accelerator-leveraging MatMul patterns. Figure 10b shows two rules supporting different B matrix patterns. The first rule searches for the access pattern of the B matrix in the standard, dense layout. For each such B, it uses the `KWayInterleave` intrinsics to shuffle the B matrix, materializes the result in memory, and `tile_load` is to AMX tile register. The second rule searches for the loading pattern of the VNNI layout verbatim. The outer `broadcast` corresponds to the `i` dimension of `A(k, i) * B(k%2, j, k/2)` (where the A matrix is enumerated while the index over B stays the same), and the three inner `ramp`s correspond to `k/2`, `j`, `k` respectively.

Both rules use the `amx-B-tile` relation that abstracts over the application-specific details, which gets queried by the lowering rules for AMX. This allows us to avoid rule duplication by implementing only one rule for lowering to hardware. HARDBOILED also has a number of rules that combine the application-specific patterns and hardware-specific lowering. Section V will showcase a rule that converts an image convolution into MatMul by constructing a Toeplitz matrix.

3) Axiomatic rules: HARDBOILED uses axiomatic rules to make pattern matching robust to syntactic variations. Some vector-related axiomatic rules are shown in Figure 10c. For instance, the first rule reads that two consecutive `broadcast`s can be flattened into one. HARDBOILED also implements additional arithmetic axioms such as commutativity. We do not implement associativity as it can make the E-graph grow exponentially and our use case does not yet require associativity.

One consideration in writing these rules is to decide what equivalent expressions to populate, similar to the quantifier instantiation problem in SMT solving [18]. For instance, the inverse of the `broadcast`-flattening rule above, which creates nested broadcasts, is also a valid rule, and it is helpful in several settings. But the inverse rule is not immediately applicable: we cannot infer what `(* l1 l2)` is before we know what `l1` and `l2` is! HARDBOILED uses two heuristics. First, we use sibling terms as hints. For example, HARDBOILED uses the

[9]Halide supports vector reduction of any associative-commutative operators. For this paper we only need + as the aggregation operator.

```
(rule ( ;; lowering matmul
  (= e (Add C (VectorReduceAdd 512
    (Mul (Cast (Float32 8192) A)
         (Cast (Float32 8192) B)))))
  (amx-A-tile A amx-A)
  (amx-B-tile B amx-B)
) (
  (let new-e (Call "tile_matmul"
    (vec-of (Mem2AMX C) amx-A amx-B)))
  (union e (AMX2Mem new-e))
))
(rule ( ;; lowering tile store
  (= s (Store buffer (AMX2Mem tile) index))
  (= index (Ramp (Ramp base 1 16)
                 (Broadcast stride 16)
                 16))
) (
  (let new-s (Evaluate
    (Call "tile_store" (vec-of
      (Var buffer) base stride tile))))
  (union s new-s)
))
```

(a) Lowering rules

```
(rewrite (Broadcast (Broadcast x l1) l2)
         (Broadcast x (* l1 l2)))
(rewrite (Broadcast x 1) x)
(rewrite (Broadcast (Load type name index)
                    lanes)
         (Load (MultiplyLanes type lanes)
               name
               (Broadcast index lanes)))
(rewrite (Broadcast (Cast type expr) lanes)
         (Cast (MultiplyLanes type lanes)
               (Broadcast expr lanes)))
(rewrite (Add (Ramp base stride r-lanes)
              (Broadcast x b-lanes))
         (Ramp
           (Add base
             (Broadcast x (/ b-lanes r-lanes)))
           stride
           r-lanes)
    :when ((= (% b-lanes r-lanes) 0)))
(rewrite (Ramp x s 1) x)
```

(c) Axiomatic rules

```
(rule ( ;; loading B in the standard layout
  (= orig-B (Load (BFloat16 8192) B-name B-index))
  (= B-index (Broadcast
    (Ramp (Ramp B-base B-stride 32)
          (Broadcast 1 32)
          16)
    16))
) (
  (let type-B (BFloat16 256))
  (let load-B (Load type-B B
    (Ramp (Ramp B-base 1 32)
          (Broadcast B-stride 16)
          16)))
  (let shuffle-B (ExprVar (Call "KWayInterleave"
    (vec-of 2 32 load-B)))
  (let amx-B (Call "tile_load"
    (vec-of shuffle-B 0 64)))
  (amx-B-tile orig-B amx-B)
))
(rule ( ;; loading B in the VNNI layout
  (= orig-B (Load (BFloat16 8192) B B-index))
  (= B-index (Broadcast
    (Ramp (Ramp (Ramp B-base 1 2)
                (Broadcast B-stride 2)
                16)
          (Broadcast 2 32)
          16)
    16))
) (
  (let amx-B (Call "tile_load"
    (vec-of (Var B) B-base B-stride)))
  (amx-B-tile orig-B amx-B)
))
```

(b) Application-specific rules

```
(function MultiplyLanes (Type i64) Type)
(rewrite (MultiplyLanes (Float32 l) x)
         (Float32 (* l x)))
(rewrite (MultiplyLanes (BFloat16 l) x)
         (BFloat16 (* l x)))
(relation has-type (Expr Type))
(rule ((= e (FloatImm v)))
      ((has-type e (Float32 1))))
(rule ((= e (Ramp inner s l))
       (has-type inner ty))
      ((has-type e (MultiplyLanes ty l))))
(rule ((= e (Broadcast inner l))
       (has-type inner ty))
      ((has-type e (MultiplyLanes ty l))))
```

(d) Supporting rules

Fig. 10: Example egglog rules in HARDBOILED by category.

following rule[10]:

```
(rewrite
  (Add (Ramp x s l1) (Broadcast a l2))
  (Add (Broadcast (Broadcast a (/ l2 l1)) l1)
       (Ramp x s l1))
  :when ((> l2 l1)
         (= 0 (% l2 l1))))
```

This rule says whenever a `broadcast` expression has a sibling `ramp` expression with a different argument lanes[11], we can nest the `broadcast` expression so that the `broadcast` expression has the same number of lanes as the `ramp`. In standard term rewriting, this is a pointless rule, because it complicates the original expression. However, with equality

[10]In our implementation, we parametrize binary operators as arguments to a more general `Bop` constructor, so the presented rule would work for any binary operator, instead of just addition.

[11]Note that for the expression to be well-defined, the two expressions must have the same number of lanes, but `l1` may not be equal to `l2` because `a` and `x` may have different lanes.

saturation, this rule can help other rules to make further simplifications. For example, this rule enables the following rule, which pushes the addition inside the base of a `ramp`.

```
(rewrite (Add (Ramp x s l) (Broadcast a l))
         (Ramp (Add x a) s l))
```

Our second heuristic is to capture common cases. For instance, because of the VNNI layout, if the B matrix has size 2xM, the buffer degenerates to a 1D vector of length $2M$, and the access pattern becomes a single `ramp`. In HARDBOILED, we introduce the following two rules to recover the general pattern from the degenerate case[12]:

```
(rewrite x (Broadcast x l) :when ((IsExpr x)))
(rewrite (Ramp e 1 l)
  (Ramp (Ramp e 1 2) (Broadcast 2 2) (/ l 2)))
```

4) Supporting rules: Many rules we have seen so far depend on certain helper relations, such as `MultiplyBy`, which multiplies the lanes of a vector type by a given number. Here we describe two other helper analyses used in HARDBOILED. These analyses are expressed as Datalog-style deductive rules in egglog.

Type checking. We implement type checking as a few deductive rules over e-classes. The type information is used in several rules.

Shape tracking. AMX and WMMA accept tensors of different dimensions, and different tensor dimensions require different intrinsics for loading/storing. We track tensor dimensions using relations `AMXShape` and `WMMAShape`, which are populated when MatMul intrinsics are emitted and queried when synthesizing intrinsics for loading tensors from/to memory.

HARDBOILED relies on the tile extractor pass to pre- and post-process the program and communicate with Halide.

The tile extractor pass annotates the input program with data movement between memory and accelerator registers. It then looks for allocations marked by the user as being stored in accelerator memory and annotates data movements of these allocations.

Next, the tile extractor collects `store` statements of the input program and prints each `store` statement as egglog expressions. The tile extractor then generates an egglog program with rules described in Figure A and spawns an egglog process to run the program. After EqSat finishes, egglog extracts the optimal program with regard to a cost model and outputs it. HARDBOILED's current cost model prioritizes terms with smaller sizes. The tile extractor parses egglog's output back to the internal representation in C++ and replaces the original `store` statements with optimized programs egglog produces.

The tile extractor lowers potential `ExprVar` nodes in the optimized program by emitting allocations with appropriate sizes and their initializations. Such allocation declarations are lifted as far as possible to maximize their reuses across loop iterations. Finally, the tile extractor desugars the shuffling intrinsics emitted during EqSat into shuffle operators in Halide IR (omitted in Section A).

[12]Because `x` on the LHS is unbounded, it needs to be grounded by a supporting relation `IsExpr`.

B. Supporting convolution-like patterns using Toeplitz transformation

Assume the user writes the following program:

```
ImageParam K(Float(16), 1),
           I(Float(16), 1);
RDom rx;
conv(x) = 0.f;
conv(x) += cast<float>(K(rx)) * cast<float>(I(x +
     rx));
output(x) = conv(x);

output.split(x, x, xi, 256)
      .vectorize(xi)
      .gpu_blocks(x);
conv.compute_at(output, x)
    .store_in(WMMAAccumulator)
    .split(x, x, xi, 256)
    .vectorize(xi);
conv.update()
    .split(x, x, xi, 256)
    .split(rx, rx, rxi, 8)
    .reorder(rxi, xi, rx, x)
    .atomic()
    .vectorize(xi)
    .vectorize(rxi);
```

The following IR program shows the convolution expression that Halide lowers to:

```
conv[ramp(0, 1, 256)] =
  (float32x256)vector_reduce_add(
    float32x2048(I[
      ramp(ramp(rx, 1, 8), x8(1), 256)
    ]) *
    x256(float32x8(K[ramp(rx, 1, 8)]))
  ) + conv[ramp(0, 1, 256)]
```

Under the hood, HARDBOILED looks for e of the following form that describes a 1D convolution.

```
(= e (Add C (VectorReduceAdd 256
  (Mul (Cast (Float32 2048)
             (Load (Float16 2048) I index-I))
       (Cast (Float32 2048)
             (Load (Float16 2048) K index-K))))))
(= index-I (Ramp (Ramp base-I 1 8)
                 (Broadcast 1 8)
                 256))
(= index-K (Broadcast (Ramp base-K 1 8) 256))
```

and produces a temporary buffer that stores the Toeplitz matrix obtained by `ConvolutionShuffle`-ing kernel K.

```
(let new-K (ExprVar (Call "ConvolutionShuffle"
  (vec-of (Var K) base-K 16 8))))
```

HARDBOILED then asserts that the convolution between I and K is equivalent to a MatMul between I loaded using pattern in Equation 1 and `new-K`.

`ConvolutionShuffle` is an intrinsic that we add to Halide to support convolution-like computations. It constructs the matrix A_K from kernel K and is implemented using LLVM's `shuffle` instructions (as Halide compiles to LLVM).

After HARDBOILED's pass, the following code is generated.

```
allocate tempbuf[float16 * 128]
tempbuf[ramp(0, 1, 128)] =
  ConvolutionShuffle(K, rx, 16, 8)
conv[ramp(0, 1, 8)] =
  wmma.mma.sync.aligned.row.row.m32n8k16.f32.f32(
```

```
      wmma.load.a.sync.aligned.row.m32n8k16.f16(
        I, rx, 8),
      wmma.load.b.sync.aligned.row.m32n8k16.f16(
        tempbuf, 0, 8),
      conv[ramp(0, 1, 8)])
```