

Towards a high-performance AI compiler with upstream MLIR

Renato Golin, Lorenzo Chelini, Adam Siemieniuk, Kavitha Madhu, Niranjan Hasabnis, Hans Pabst, Evangelos Georganas, and Alexander Heinecke

Intel Corporation

Abstract

This work proposes a compilation flow using open-source compiler passes to build a framework to achieve *ninja* performance from a generic linear algebra high-level abstraction. We demonstrate this flow with a proof-of-concept MLIR project that uses input IR in Linalg-on-Tensor from TensorFlow and PyTorch, performs cache-level optimizations and lowering to micro-kernels for efficient vectorization, achieving over 90% of the performance of *ninja*-written equivalent programs. The contributions of this work include: (1) Packing primitives on the tensor dialect and passes for cache-aware distribution of tensors (single and multi-core) and type-aware instructions (VNNI, BFDOT, BFMMMLA), including propagation of shapes across the entire function; (2) A linear algebra pipeline, including tile, fuse and bufferization strategies to get model-level IR into hardware friendly *tile* calls; (3) A mechanism for micro-kernel lowering to an open source library that supports various CPUs.

1 Introduction

Production high-performance code often needs to use highly-optimized libraries to achieve acceptable performance on modern hardware. Using generic micro-kernel libraries allows users to combine low-level calls into larger operations without worrying about the “*last mile*” optimizations or optimal hardware utilization. However, designing highly-optimized libraries requires intricate knowledge of the problem space and the target architecture, thus leading to lack of generality. Furthermore, mapping those kernels on the existing applications is still a sizeable challenge, inaccessible to most application programmers [1].

Other AI compiler frameworks (ex. IREE ¹) have presented opportunities to accelerate application code by combining language extensions, graph sharding and data reordering, IR compiler memory bandwidth and compute density optimizations, and super-optimized libraries. However, due to the complexity of those frameworks and the vast domains they need to cover, support for efficient execution gets limited by common high-level patterns, leading to an explosion of problem-specific kernel implementations.

In our previous work on implementing a micro-kernel library for the Tensor Processing Primitives (TPP) [2], we demonstrated that one can build a comprehensive set of deep learning and high-performance algorithms and achieve state-of-the-art performance by selecting appropriate micro-kernels for the suitable tensor shapes. We have implemented those primitives in the open-source micro-kernel library named libxsmm ², which can reach over 90% performance of the achievable peak (hand-written assembly).

This work³ builds on the success of TPP’s state-of-the-art performance of various algorithms on a variety of CPUs by bringing a set of high-level linear algebra compiler passes to automatically choose the correct TPP operations, in the right order, with the suitable flags, including packing tensors and adjusting iteration spaces for optimal traversal and full utilization of hardware resources. More importantly, it is possible (and desirable) to achieve this goal by having those passes in LLVM upstream, while adding a small low-level layer (dialect and conversion passes) specific to the library being used.

Our compiler is based on the well-known MLIR [4] technology by extracting programs from existing high-level frameworks (such as TensorFlow and PyTorch) through our tensor manipulation passes, into low level library and hardware dialects for further lowering. This work exposes compiler heuristics via command line flags, allowing users to identify what constraints work best for each case, and investigate the boundaries with which to create a cost model that would drive this automatically. The construction and utilization of this cost model is a subject for future work.

Our main contribution is to enable the ease of using frameworks and automatic compilers on high-level programs with the performance of “*ninja written*” low-level libraries, taking advantage of the last drop of hardware performance on a selection of architectures without resorting to user-specified schedules, pragmas, hints or hand-crafted intrinsics and inline assembly.

¹<https://github.com/openxla/iree>

²<https://github.com/libxsmm/libxsmm>

³<https://github.com/plaidml/tpp-mlir>

2 MLIR and the Linalg Dialect

The MLIR compiler infrastructure is a project under the LLVM umbrella well-suited for multi-level IR rewriting. MLIR provides an intermediate representation (IR) with only a few concepts being built, leaving most IR customizable. Such IR allows compiler developers to match the right abstraction level for their problems by introducing custom types, operations, and attributes. We use the static single-assignment (SSA) form, which is suitable for imperative programming styles that machine learning (ML) and high-performance computing (HPC) applications lower to.

A dialect is a basic structure that enables the MLIR to implement a stack of reusable abstractions, composed of operations, types, attributes, etc. Each abstraction encodes and preserves transformation validity preconditions directly in its IR, reducing the complexity and the cost of analysis passes.

Each dialect models a specific domain. For example, the Linalg dialect⁴ captures linear-algebra operations on either tensor or buffer operands. Listing 1 shows a single layer of a multi-layer perceptron ML model, with a matrix multiplication followed by a “*bias*” addition and rectifier (ReLU) implemented as $\max(x, 0.0)$. The operations’ semantics describe the computation in the “*inner loop*” and define iteration order via “*indexing-maps*” and “*iterator-types*”.

As is common in the MLIR ecosystem, we originally developed our own (TPP) tile dialect, between Linalg on Tensors and our XSMM dialect, which allowed us to manipulate tiling, fusing, and bufferization on our terms. However, through upstream discussions on a tile dialect design, we have concluded that the upstream Linalg dialect should have such abstractions. We then updated the Linalg dialect with our operations and now use only Linalg for both whole-tensor and tile semantics, on tensors and memrefs. This allows us to further our mission to impact and reuse upstream high-level transformations for a common compilation infrastructure.

3 Compilation Strategy

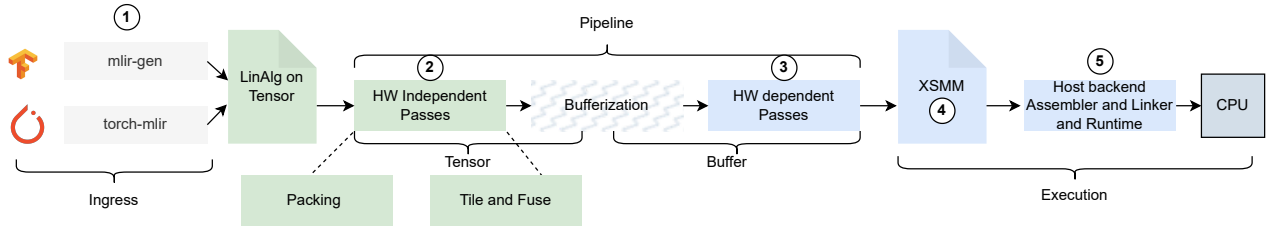


Figure 1: A simplified view of the proposed compiler strategy. In gray are external components, in green are the upstream compiler technology while in blue are the potentially downstream parts. Boundaries depend on which ingress format and which hardware abstractions are used. XSMM is our choice of CPU library, and OpenCL is a potential choice for GPU libraries. The proposal is equally valid with dialects and further compilers (ex. LLVM) down the line.

Our compilation strategy in figure 1 is based on five main components: ① An ingress layer⁵, that extracts MLIR from existing frameworks into Linalg-on-Tensor Intermediate Representation (IR); ② A high-level hardware-agnostic Linalg pipeline; ③ A low-level lowering dialect; ④ A pipeline for libxsmm; ⑤ An execution strategy for the generated code, including runtime libraries and wrappers.

The ingress layer is based on external frameworks: IREE for TensorFlow models and torch-mlir⁶ for PyTorch Dynamo. On top of that, we have created a tool named *mlir-gen* that replicates the IR that IREE generates from TensorFlow but is extensible enough to generate various shapes, number of layers, size of matrices, type packing, etc. This IR generator aims at providing dynamic inputs to our compiler tests and replaces our dependency on IREE, as seen in figure 1.

The second component is a Linalg-based pipeline built by composing upstream building blocks. Following previous work [6], we have implemented a focused pipeline, using existing passes with our new operations (`tensor.pack` and `tensor.unpack`) and their respective transformations.

While our implementation is not yet a fully flexible Linalg optimizing pipeline, we demonstrate the viability of such a pure Linalg-based approach. In essence that means, relying on only using Linalg passes and reasonable heuristics to achieve high performance by combining compiler passes with minimal vendor-optimized routines. The idea is to leverage compilers for what they are good at: fusion, tiling, and data layout while relying on vendor-optimized routines for optimizations like vectorization and instruction selection.

⁴<https://mlir.llvm.org/docs/Dialects/Linalg>

⁵<https://github.com/plaidml/mlir-generator>

⁶<https://github.com/llvm-project/torch-mlir>

```

// Affine maps  $M, K * K, N \rightarrow M, N$ 
#map-mk = affine_map<(d0, d1, d2) -> (d0, d2)>
#map-kn = affine_map<(d0, d1, d2) -> (d2, d1)>
#map-mm = affine_map<(d0, d1, d2) -> (d0, d1)>

// A perfectly nested affine fused multiply and accumulate operation (matmul)
%0 = linalg.generic {
    indexing_maps = [#map-mk, #map-kn, #map-mm],
    // Reduction iterator type is the third, ie. ``d2'', which is the ``K'' dimension
    iterator_types = ["parallel", "parallel", "reduction"]
}
// Inputs are A and B matrices, C is the initialized of the output (generally zero).
ins(%A, %B : tensor<128x256xf32>, tensor<256x512xf32>)
// Output is the C matrix, here representing initialization ( $C += A * B$ ), where C can be zero
outs(%C : tensor<128x512xf32>) {
    ^bb0(%in: f32, %in_1: f32, %out: f32):
        %3 = arith.mulf %in, %in_1 : f32
        %4 = arith.addf %out, %3 : f32
        linalg.yield %4 : f32
} -> tensor<128x512xf32>

// Affine maps element-wise & broadcast
#map-ew = affine_map<(d0, d1) -> (d0, d1)>
#map-bc = affine_map<(d0, d1) -> (d1)>

// A binary operation on the output of the matmul above (ex. Bias Add)
%1 = linalg.generic {
    indexing_maps = [#map-ew, #map-bc],
    iterator_types = ["parallel", "parallel"]
}
// Inputs are C and Bias matrices.
// Note: the bias is a 1D vector being broadcasted to add element-wise.
// Note: the C matrix is the initializer of the output, so it's in `outs`.
ins(%BIAS : tensor<512xf32>)
outs(%0 : tensor<128x512xf32>) {
    ^bb0(%in: f32, %out: f32):
        %4 = arith.addf %in, %out : f32
        linalg.yield %4 : f32
} -> tensor<128x512xf32>

// A unary operation on the output of the binary above (ex. ReLU)
%ZERO = arith.constant 0.000000e+00 : f32
%2 = linalg.generic {
    // Element-wise parallel operation only uses MN maps
    indexing_maps = [#map-ew],
    iterator_types = ["parallel", "parallel"]
}
// Input is just the result above.
// Note: the result is the initializer of the output, so it's in `outs`.
outs(%1 : tensor<128x512xf32>) {
    ^bb0(%out: f32):
        %4 = arith.maximumf %out, %ZERO : f32
        linalg.yield %4 : f32
} -> tensor<128x512xf32>

return %2

```

Listing 1: A simple multi-layer perceptron (MLP) layer represented in Linalg generic operations. The affine maps in “indexing_map” describe the iteration space of each input, the “iterator_types” the type of loop (parallel or reduction), while the inner region specifies the computation. The arguments (“ins” and “outs”) are tensors created beforehand.

Our core mission is to have a Linalg-based optimization pipeline upstream in MLIR. Our first step in this direction was the upstreaming of operations (`tensor.pack`, `tensor.unpack` and several Linalg named operations) and creation and improvement of upstream passes (packing, tiling, fusion).

The fourth component is an in-house dialect (XSMM) that plays the role of interfacing with our “last-mile” library: *libxsmm*. The XSMM dialect does not need to be upstream, as it represents a third-party library and its semantics. As such, we show that it’s feasible to have an upstream Linalg-based compiler for the high-level transformations and a downstream target-specific dialect to do low-level transformations. XSMM dialect is used to perform transformations that are specific to the library, not necessarily the final target architecture. For example, it exposes fused library calls that can implement a whole MLP layer (GEMM + Binary + Unary), saving on loads/stores, register renaming, cache flushes, and buffer allocation. The high-level Linalg IR does not need to know or care about this. Lowering Linalg on memrefs into other hardware specific dialects (such as `vector` or GPU target dialects) is subject for future work.

The fifth and last component is a simple runtime wrapper for *libxsmm* and a just-in-time compiler infrastructure to get executable code. We reuse the upstream `mlir-cpu-runner`, extending it to include our dialects and passes as well as to generate a benchmark loop for the kernels being tested (using a local `perf` dialect that we’re discussing how to upstream).

3.1 Why Linalg on Tensors?

Different frameworks (e.g. TensorFlow, PyTorch) implement their own MLIR dialects (StableHLO, Torch) with semantics equivalent to their internal graph formats. There are other high-level dialects (ex. TOSA⁷) but they also have their individual semantics, which makes it hard to work on a common optimization infrastructure.

To avoid this complexity, compilers aim at converting those dialects into a common “*compiler IR*”: Linalg on Tensor. These two dialects can describe a substantial part of ML applications with a small number of generic operations (as seen in Listing 1) and are very amenable to transformations.

Being at a tensor *value* semantics, where each new operation materializes a new tensor, allows us to perform most of the optimization we need, namely packing, kernel fusion, and tiling, without having to worry about memory constraints. In addition, there are upstream passes that we want to reuse that run on either Linalg or its interfaces, many of those to which we contributed substantially.

This set of dialects, with tiled and fused operations is then bufferized by the *one-shot* bufferization pass, cleaned, canonicalized and further lowered to low level dialects such as XSMM, where library/hardware specific passes can operate on an already memory-friendly shape.

4 Compiler Passes

The compiler pipeline consists of two parts: ② high-level optimizations, generic to all targets and based on cost models, heuristics-based decisions, and graph-rewrites, and ③ low-level optimizations, specific to each chosen target, primarily adjusting the high-level decisions to the target expectations (low-level dialects, function calls, intrinsic, etc).

High-level passes like packing, tiling, and fusing are hardware-agnostic (with hardware-specific costs) and aim to reduce round-trip to memory by providing aggressive fusion strategies around contraction-like operations (i.e., fuse element-wise operations with matmul). Tiling exposes scalar or parallel loops around operations that can map 1:1 with micro-kernel operations in *libxsmm* (exposed by the XSMM dialect), but also fits common CPU and GPU code generation patterns (using `vector`, `gpu` and other close-to-hardware dialects).

Low-level passes can then be done on the specific low-level dialects (such as XSMM and `vector`), where target-specific transformations can be done at a representation closer to the hardware/library.

4.1 Pack, Unpack, and Propagation

Packing (or data-blocking) is a well-known transformation in high-performance libraries that copies a non-contiguous block of data to a contiguous block in memory to reduce the number of TLB entries required to access each page. When copying data, packing rearranges block elements to decrease the stride between consecutive accesses, improving spatial locality and cache behavior.

Bringing it into the compiler has advantages, as we can propagate the layout through the IR instead of paying the price for every invocation. We perform packing by introducing two new operations in the tensor dialect: `tensor.pack` and `tensor.unpack`. The former takes a tensor as input and produces a re-layout tensor as output, while the latter undoes the packing, bringing the tensor back to the original layout. Table 2 shows one of the layouts we use for different matmul operations.

⁷<https://mlir.llvm.org/docs/Dialects/TOSA>

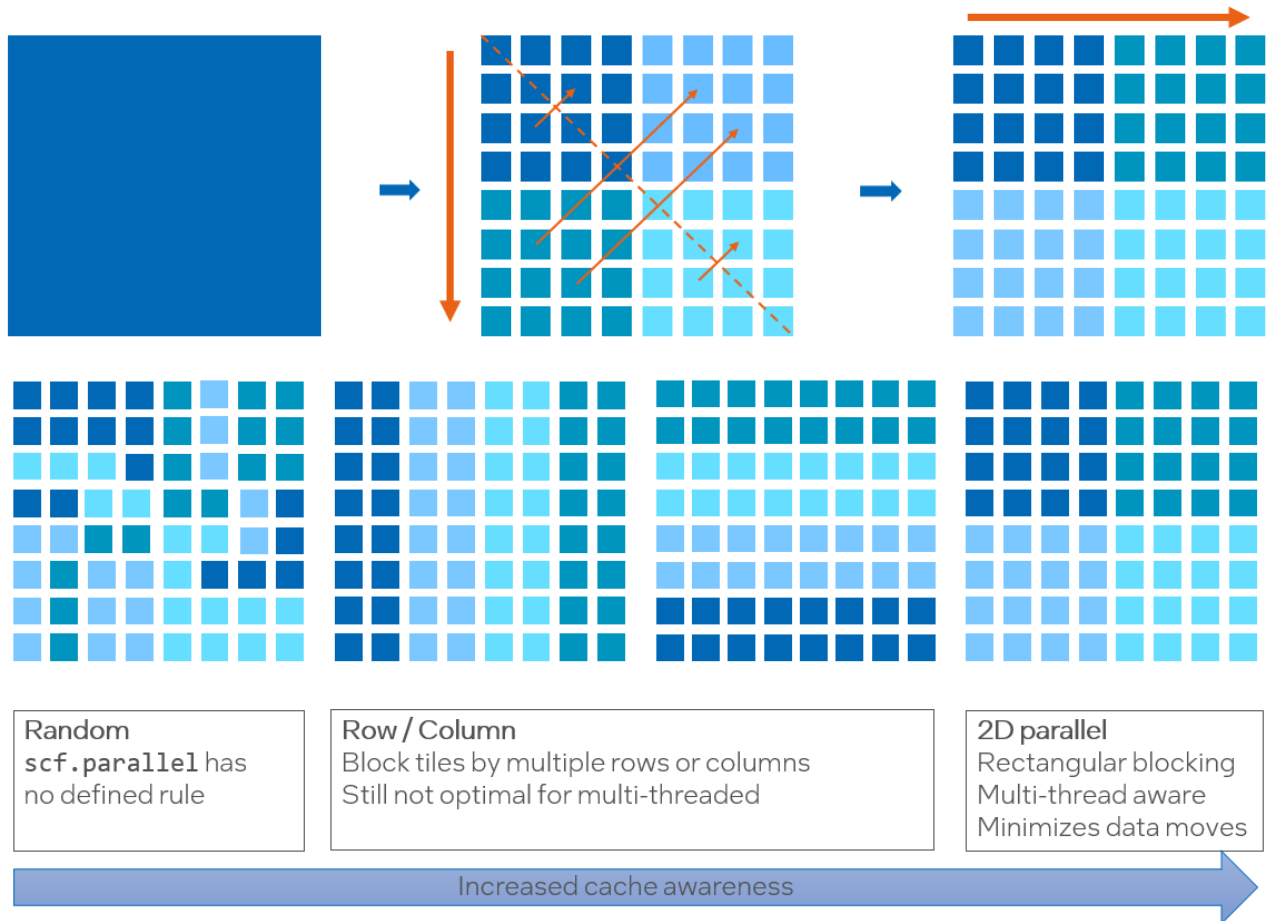


Figure 2: Packed layout for GEMM operation. After tiling (smaller square), the tiles are transposed whole (“*block-transpose*”). For optimal multi-threaded locality we also group different blocks (single-colored areas) for each thread.

Currently, we employ very simple heuristics around packing: select an optimal tile size and divide the outer dimensions around it, reordering the blocks of the second tensor to improve cache locality. The optimal tile size is currently fixed but with the option to allow users to override it. A cost-model-based heuristics to select the best tile size for a given architecture is subject to further work.

Each operation is packed as long as its iteration space has more iteration than the packing factor encoded in the pass. Usually, if the iterations on each dimension are less than a certain architecture-specific amount, the computation fits in the cache, and packing is unnecessary.

To amortize the cost of packing, packing operations are propagated through element-wise operations, and a pair of `unpack(pack(t))` are folded away when possible (figure 3). This allows us to be naive on the initial packing, just looking at large matrix multiply and contractions, then expand the same shapes throughout the following operations reaching the next stages (following layer, next step in the same layer), and remove all intermediate packing operations except the first ones on inputs and last ones on return values.

Packing constants (weights and biases on inference models) can be done at compile time if we have access to the data in the model (for example as a constant global or an `arith.constant`). However, more advanced folding [5] can be done at run time even on models where the constant data is passed as arguments by packing and computing the first time, caching and reusing the subsequent times. Moreover, zero-initialized buffers (for example, the output C matrix tile for matmul) can be just reshaped at compile time, since it’s only a *splat* of zeroes onto an arbitrary shape.

4.2 Tile and Fuse

We use upstream building blocks to assemble our tiling and fuse pass. The main idea is to consider contraction operations and fuse them along the parallel dimensions with consumers’ or producers’ element-wise operations.

As in Listing 2, we fuse along the M and N parallel loop. Using a bottom-up traversal, we consider each contraction and collect possible fusable consumers or producers looking at their iteration domain. This effectively creates a cluster of element-wise operations around a contraction. We require each operation to belong to a

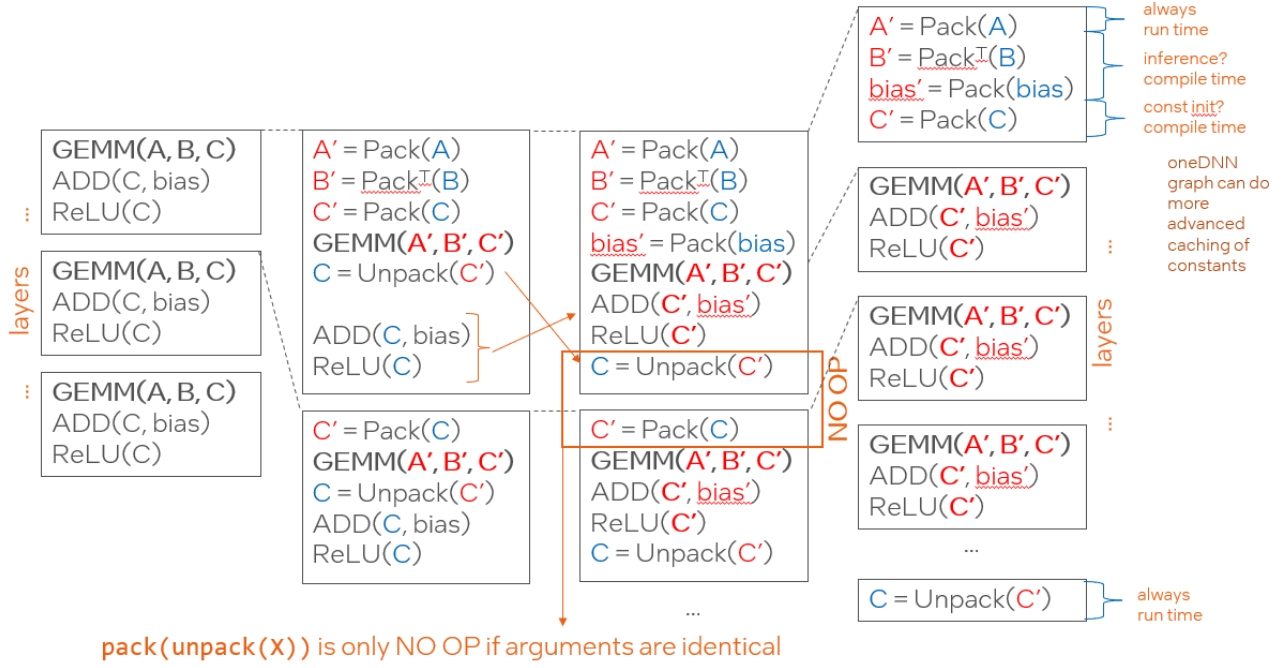


Figure 3: Pack propagation through a multi-layer model. Packed GEMMs propagate their layout to the following element-wise operations, exposing canonicalization in between layers to elide all intermediate packs and unpacks, leaving only the initial packs and final unpack.

single cluster domain, and we prevent recomputation by avoiding duplicating operations that belong to multiple fusion domains.

```
// Convert tile-wise operations
for(MB, NB) {
  for (KB) {
    C[MB] [NB] [mb] [nb] += A[MB] [KB] [mb] [kb] * B[NB] [KB] [kb] [nb]
  }
  C[MB] [NB] [mb] [nb] = add(C[MB] [NB] [mb] [nb], bias[mb] [nb])
  C[MB] [NB] [mb] [nb] = max(C[MB] [NB] [mb] [nb], 0)
}

// Into a parallel BRGEMM "tile" op + element-wise tail ops
parallel(MB, NB) {
  -> extract { A, B, C } x [MB] [NB] [KB] as appropriate
  // Note: this is a batch-reduce GEMM into a "tile"
  C[mb] [nb] += A[KB] [mb] [kb] * B[KB] [kb] [nb]
  C[mb] [nb] = add(C[mb] [nb], bias[mb] [nb])
  C[mb] [nb] = max(C[mb] [nb], 0)
  <- insert into C[MB] [NB]
}
```

Listing 2: The tile and fuse pass will materialize the two parallel loops MB and NB, because both are parallel outer dimensions, i.e., there are no loop carried dependencies between the tile operations.

4.3 Lowering to Hardware Dialects

After the hardware independent passes, we run the upstream bufferization pass to move from a value-based tensors to memory buffer `memrefs`. After bufferization, the Linalg on buffers IR, now operating in tile shapes inside parallel loops, is lowered to our XSMM dialect, which acts as an interface to libxsmm. After low-level dialect-specific optimization passes, XSMM is lowered to function calls that invoke libxsmm through a simple C runtime.

4.3.1 The XSMM dialect

The XSMM dialect maps the behavior of the libxsmm library and enables library-specific optimizations, for example, call fusion.

The libxsmm library is a JIT-ing library and is split into two stages: *dispatch* and *invoke*. The dispatch stage receives the shapes of the buffer, leading dimensions, broadcast, and fusion flags and compiles the microkernel in memory, returning a pointer into its implementation. The second time a *dispatch* function is called, it just returns a cached pointer to the same implementation. The invoke stage calls that function pointer with the actual tensor data (usually a tile into a larger buffer with the appropriate strides), which computes the operation, writing the result to the output buffer.

XSMM exposes only five operations: **unary**, **binary**, **gemm**, **brgemm** and **fused_brgemm**. **unary** are element-wise unary operations like ReLU, but also broadcasts, transposes and reductions. **binary** are element-wise binary operations like add or multiply. **gemm** represents General Matrix Multiplications mirroring the BLAS interface; **brgemm** is a more powerful abstraction that carries an extra reduction dimension on the input operands, allowing reducing tiles of A and B in the same C tile. Finally, **fused_brgemm** allows register fusion between BRGEMM and an element-wise prologue and epilogue.

The last operation above demonstrates the power of library-specific (not just hardware-specific) compiler optimizations. To allow for better low-level optimization, libxsmm implements a fused BRGEMM, which can add the following arbitrary binary and unary operations to the BRGEMM, including broadcast semantics.

```
// Convert multiple XSMM calls
%3 = xsmm.unary.dispatch zero [...] flags = (none)
%4 = xsmm.brgemm.dispatch [none] flags = (none)
%5 = xsmm.binary.dispatch add [...] flags = (bcast_col_in0)
%6 = xsmm.unary.dispatch relu [...] flags = (none)
scf.parallel (MB, NB) {
  %subview_A = memref.subview ... // into A
  %subview_B = memref.subview ... // into B
  %subview_C = memref.subview ... // into C

  // C[MB][NB] = { 0.0 }
  xsmm.zero(..., %3, %subview_C)
  // C[MB][NB] = BRGEMM(A[MB][NB], B[MB][NB], C[MB][NB])
  xsmm.brgemm(data_type = f32, %4, %subview_A, %subview_B, %subview_C, %c0)
  // C[MB][NB] = ADD(broadcast(Bias[NB]), C[MB][NB])
  xsmm.binary add(..., %5, %BIAS, %subview_C, %subview_C)
  // C[MB][NB] = ReLU(C[MB][NB])
  xsmm.unary relu(..., %6, %subview_C, %subview_C)
}

// Into a single fused one with all flags
%3 = xsmm.fused_brgemm.dispatch [...] [add,relu] flags = (beta_0)
  binary_flags = (bcast_col_in0) unary_flags = (none)
scf.parallel (MB, NB) {
  %subview_A = memref.subview ... // into A
  %subview_B = memref.subview ... // into B
  %subview_C = memref.subview ... // into C

  // C[MB][NB] = { 0.0 }
  // C[MB][NB] = BRGEMM(A[MB][NB], B[MB][NB], C[MB][NB])
  // C[MB][NB] = ADD(broadcast(Bias[NB]), C[MB][NB])
  // C[MB][NB] = ReLU(C[MB][NB])
  xsmm.fused_brgemm(..., %3, %subview_A, %subview_B, %subview_C, %BIAS %c4)
}
```

Listing 3: XSMM pass that fuses multiple XSMM calls into a fused call to improve register and memory usage.

4.4 Parallelism

We support parallelism using OpenMP by converting `scf.forall` into `scf.parallel` to expose CPU thread parallelism by running the upstream OpenMP conversion passes and linking with LLVM's OpenMP library.

We force the OpenMP affinity to guarantee the correct distribution of the tiles onto the appropriate threads with the setting `KMP_AFFINITY` to `granularity=fine, verbose, compact,1,0`.

4.4.1 2D parallelism

We perform 2D parallelism transformation on the loops around the XSMM dialect (see figure 2). This transformation distributes the OpenMP threads across the appropriate tile regions to increase reuse and minimize core-to-core traffic in a multi-core chip by best possible leveraging matrix multiplication’s algorithmic properties. Different tile region shapes are optimal for different number of threads, micro-architecture extensions and XSMM-specific behaviour, so this pass is done at the low-level pipeline to extract the most of it.

This pass further tiles the outer parallel dimension loops into smaller iteration spaces (multiple of the number of threads) and creates sequential loops inside each thread, guaranteeing the execution of all tiles in the same *group* to be on the same thread. Getting that balance right (memory bandwidth, caching, compute distribution) is not trivial.

Compiler command line options are introduced to select the *best known* factor for each run on our benchmarks. However, as above, this technique is only being used to collect data on costs to support high-level heuristics decisions and will be subject to future work.

4.4.2 AMX tile configuration hoisting

The Intel AMX matrix unit needs to be configured via a status register before usage and reset after usage. This configuration is expensive as it resets the unit twice inside the inner loop.

Without further information, libxsmm must setup and reset around every loop according the ABI as both the configuration of the status register and the actual registers are defined as volatile. To avoid this extra cost, the compiler splits the GEMM operation into a triplet: **tile config** + **GEMM** + **tile reset**, and because the tile operations do not use any data from inside the inner loop (all dominators are outside), we can hoist the call outside of the loop.

However, as shown in figure 4, we must setup and reset the tiles at least once per thread, so we only hoist the tile calls past the sequential loops, not past the outer parallel loop (that will be executed on different threads).

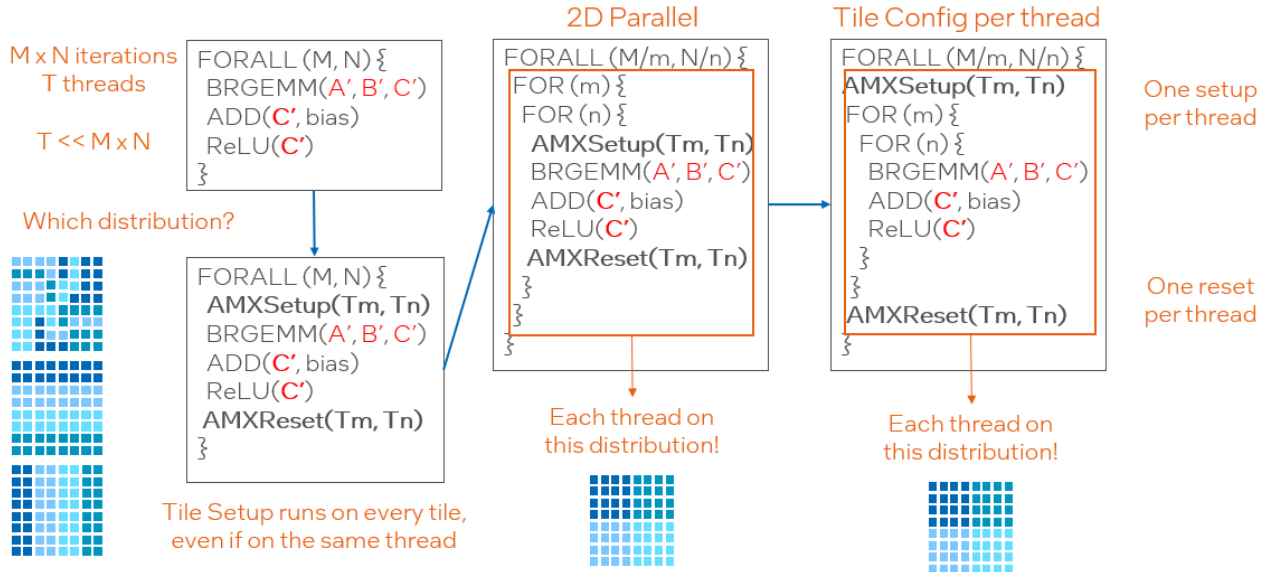


Figure 4: Each layer is executed across all cores (data parallelism). For data locality, we block each thread within a single block (of tiles) within the original matrix (see figure 2). To amortize the cost of using the matrix extension on Sapphire Rapids, we hoist the setup and reset calls within each thread.

5 Results

We compare our compiler’s performance against libxsmm hand-written code (from its `libxsmm-dnn` repository), which, after micro-architecture analysis, is known to reach higher than 90% of the achievable total performance of each CPU we target. Our compiler emits calls to the same micro-kernels, so the delta between the hand-written and compiler versions is due to the automatic parallelization of the input IR and not the low-level hardware optimizations.

The benchmarks use an MLP model to demonstrate the critical optimization in machine learning, not as a measure of output performance in existing ML models (such as ResNet, DLRM, and BERT). Our test kernels

mimic a 3-layer MLP with batch size 256 and hidden sizes 1024. We consider only inference where weights and biases are constants. Using such motifs for workload abstraction is common practice. It has been used by Google when introducing the TPU hardware or Meta when proposing the DLRM benchmark.

The CPU architectures we run on from AWS instances are:

- (c6a): AMD(R) EPYC 7R13 (Zen3), supporting AVX2 (BF16 emulated)
- (c6i): Intel(R) Xeon(R) Platinum 8375C (CLX), supporting AVX512 (BF16 emulated)
- (c7i): Intel(R) Xeon(R) Platinum 8488C (SPR), supporting AMX (BF16 native)
- (c7a): AMD(R) EPYC 9R14 (Zen4), supporting AVX512_bf16 (BF16 native)
- (c7g): AWS Graviton 3 (Gvt3), ARM V1 core supporting SVE (BF16 native)

The c6 architectures do not support native BF16 float types and were benchmarked to demonstrated our strategy works for 32-bit floating point on older hardware. As the BF16 types are emulated, we achieve lower performance than 32-bit, which is expected, since we cannot use 16-bit floats natively.

The c7 architectures all have native BF16 support and show our results in three different vendors. On Intel Sapphire Rapids (c7i), we use the AMX extension for 2D matrix operations; On AMD Genoa Zen4 (c7a) we use the AVX512_bf16 extension; On AWS Graviton 3 (c7g), we use SVE’s scalable vector BFMMMLA extension for fast BF16 matrix operations.

We pin all our single-core benchmarks on core 3 and use OpenMP affinity from core 1 onward to avoid noise in benchmark results as core 0 is being used by the kernel.



Figure 5: Single-thread results for all CPUs. The compiler’s performance is on par against all hand-written results except c7i (SPR), where compute density can be affecting memory bandwidth. Note, the scale on the first three plots are up to 250 GFLOPS, while the last two are up to 2 TFLOPS.

5.1 Single-thread performance

Figure 5 compares two types of runs:

1. **LIBXSMM-DNN**, using hand-written C++ code calling libxsmm micro-kernels in the optimal shapes for maximum known performance. This is our baseline.
2. **TPP-MLIR**, using our compiler to call the same micro-kernels above, but with the optimal configurations selected by the compiler. The model shapes here are in 4D, since the hand-written code does not do *online* packing. This allows a fair performance comparison.

On 32-bit floats (FP32), the compiler achieves the same overall performance as the hand-written code on all machines, with a deviation of less than 5%. The compiler is slightly slower on Intel machines (c6i and c7i), equally fast on Arm (c7g) and slightly faster on AMD machines (c6a and c7a). This is due to the hand-written code being optimized on Intel machines and the compiler being a more generic approach.

This is not too surprising, given that small variations can be seen with different choices for loop order, tile sizes and tensor caching that **parlooper** [3] finds with an exhaustive search. We have seen our compiler “*get lucky*” a few times when comparing with hand-written code. Future work is needed to explore that space systematically and always produce the most efficient configuration for every target.

We have identified a further difference between the hand-written code and the compiler, where the former allocates a single (strongly aligned) *arena* region to use as scratchpad, while the compiler relies on individual (not necessarily aligned) memory allocations. While this is unrelated to micro-kernel execution, it is a high-level decision that needs to be in the compiler’s tool kit. Further analysis and implementation of these fixes is subject to future work.

On 16-bit “*brain*” floats (BF16), the picture requires a bit more explanation. Neither Zen3 nor CLX have native support for BF16, so we emulate it with older instruction sets, and therefore the performance is lower than FP32. However, they’re still the same as the hand-written code on all machines except c7i.

Sapphire Rapids (c7i) uses the AMX extension, which provides a much higher compute density than FP32, and therefore the effects of memory bandwidth are more apparent. Especially, if some of those loads and stores are done on unaligned memory, we can get a pathological case that is 4x slower than peak, which can easily lead to a 30% performance degradation. Similar effects can be happening on Graviton 3’s SVE extensions, but at a lower rate given the smaller compute density.

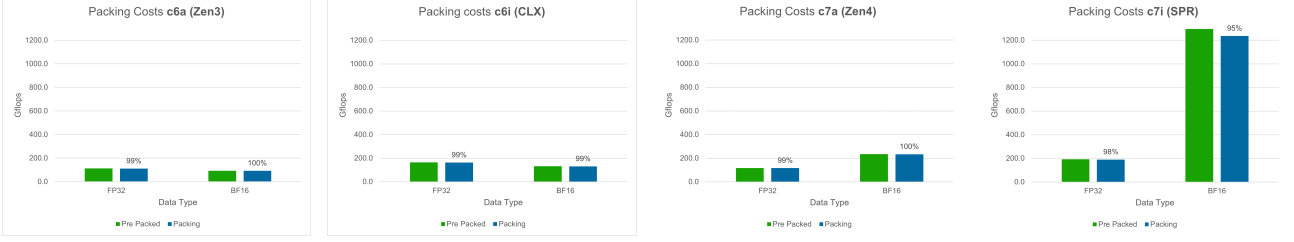


Figure 6: Comparison of `mlir-gen` (TensorFlow-like) IR with and without compiler packing applied. This shows that the cost of packing can be easily hidden with constant packing and fusion with consumers and producers.

5.2 Single-thread packing costs

Figure 6 compares two types of runs:

1. **Pre Packed**, where `mlir-gen` creates the model shapes in their already packed (4D) versions. This is the TPP-MLIR results above and are our baseline.
2. **Packing**, where `mlir-gen` creates the model shapes in their original (2D) versions, using our compiler to find and perform tensor packing (at compile or run time).

These results show the impact of compile and run time packing performed by the compiler, which are on average 1% for all machines and data types. Sapphire Rapids (c7i) on BF16 shows a greater impact (7%) due to the compute density problem identified above.

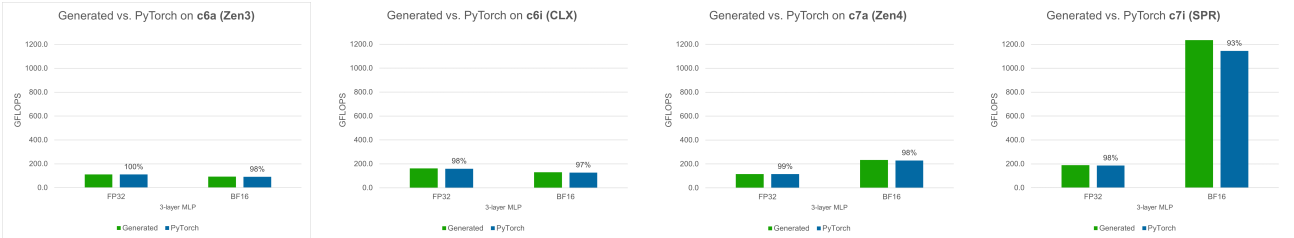


Figure 7: Comparison of `mlir-gen` (TensorFlow-like) IR that we based our work on against PyTorch IR that we extracted through Torch Dynamo and `torch-mlir` at the end of the project.

5.3 Models from new frameworks

Figure 7 compares two types of runs:

1. **Generated**, where `mlir-gen` creates the model shapes in their original (2D) versions, using our compiler to find and perform tensor packing. This is the **Packing** results above and are our baseline.
2. **PyTorch**, where we extract MLIR from PyTorch on a similar kernel as the above, and pass it through our compiler. This shows how our compiler performs when encountering new IR that it hadn’t seen before.

These results show, at least for the PyTorch case, that as long as the frameworks lower the same models in similar ways, the compiler can perform the same transformations and achieve roughly the same performance (2% on average).

The main differences between these PyTorch models and our generator were:

1. TensorFlow lowers ReLU as `maxf(0, x)` while PyTorch lowers it as `max(0, x) + select(0, x)`, which needed to be recognized as an XMM operation to lower and fuse correctly.

2. PyTorch passes constants as arguments, so we had to force PyTorch to disable auto-grad and some other features to generate a similar kernel.
3. PyTorch stores its weights in a transposed way, which incurs in additional memory-bound operations.
4. PyTorch lowers the matmul and the following element wise operations writing to different buffers, so fusion does not work out of the box.

Sapphire Rapids (c7i) on BF16 shows again a greater impact (5%) due to the compute density problem identified above, since most of the changes are bandwidth related (optimal traversal, stray memory allocations, weight transposes).



Figure 8: Scalability of the compiler results on c7a (Zen4), c7g (Gvt2) and c7i (SPR) architectures, compared to libxsmm-dnn. The numbers on top represent speed improvement over their own single-threaded baseline. Very good scalability across the board.

5.4 Parallel scalability

Figure 8 compares two types of runs:

1. **LIBXSM-DNN**, our hand-written C++ code with OpenMP for multi-threaded execution. This is our baseline.
2. **TPP-MLIR**, using our compiler on 4D shapes as above, with OpenMP for multi-threaded execution.

We use OpenMP to test thread scalability on 2, 4, 8 and 16 threads on each hardware architecture. We use `KMP_AFFINITY` as `granularity=fine,verbose,compact,1,0` to distribute the threads across all cores (not hyper-threads) from core 1 (since 0 is being used by the kernel).

Our results show very good scalability across the board, on both FP32 and BF16 data types. The compiler shows similar multi-threaded performance as the hand-written code, which means the same scalability on both Zen4 and Graviton 3, but increased scalability on Sapphire Rapids due to the lower single-threaded performance, while the absolute reach multi-thread performance is identical, hence the compiler delivers the same performance as the handwritten library on all platforms.

6 Conclusion and Future Work

In this work, we demonstrate one can achieve comparable *ninja performance* on ML models using our high-level MLIR compiler that, in turn, uses low-level tile-based building blocks without needing user input. Using a ninja-based heuristic, we have shown how to select the appropriate micro-kernels for known high-level operations.

It’s important to emphasize that by upstreaming our high-level logic, we can reduce the maintenance cost for generic abstractions that are not architecture-specific, and participate in the design the the MLIR compilation infrastructure to define an industry standard, guiding design of future compilers, libraries and hardware architectures.

We have also shown the benefits of the micro-kernel approach in compilers towards separating the concerns between what a compiler can do well (high-level optimizations like tiling, fusion, and data layout) from what it is hard to achieve in practice: efficient vectorization, optimal instruction selections, pipelined register allocation. This also allows downstream experimentation on pre-production hardware extensions without having to re-write the entire stack or inject high maintenance changes into production compilers.

As future work, we plan to develop a cost model to drive the compiler heuristics, extend the programs we can optimize into more DL/HPC workloads (ex. transformers), expand the micro-kernels into other targets (GPU, accelerators), look into the interaction between micro-kernels, IR kernels and compiler vectorization on general code generations, improve the heuristics search for optimal tiling strategies and loop orders, and improve integration of downstream passes into the upstream pipeline to improve collaboration between the two worlds.

References

- [1] Lorenzo Chelini, Oleksandr Zinenko, Tobias Grosser, and Henk Corporaal. Declarative loop tactics for domain-specific optimization. *ACM Trans. Archit. Code Optim.*, 16(4), dec 2019.
- [2] Evangelos Georganas, Dhiraj Kalamkar, Sasikanth Avancha, Menachem Adelman, Cristina Anderson, Alexander Breuer, Jeremy Bruestle, Narendra Chaudhary, Abhisek Kundu, Denise Kutnick, Frank Laub, Vasimuddin Md, Sanchit Misra, Ramanarayan Mohanty, Hans Pabst, Barukh Ziv, and Alexander Heinecke. Tensor processing primitives: A programming abstraction for efficiency and portability in deep learning workloads. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’21, New York, NY, USA, 2021. Association for Computing Machinery.
- [3] Evangelos Georganas, Dhiraj Kalamkar, Kirill Voronin, Antonio Noack, Hans Pabst, Alexander Breuer, and Alexander Heinecke. Harnessing deep learning and hpc kernels via high-level loop and tensor abstractions on cpu architectures, 2023.
- [4] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.
- [5] Jianhui Li, Zhennan Qin, Yijie Mei, Jingze Cui, Yunfei Song, Ciyong Chen, Yifei Zhang, Longsheng Du, Xianhang Cheng, Baihui Jin, Yan Zhang, Jason Ye, Eric Lin, and Dan Lavery. onednn graph compiler: A hybrid approach for high-performance deep learning compilation. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 460–470, 2024.
- [6] Nicolas Vasilache, Oleksandr Zinenko, Aart J. C. Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, and Albert Cohen. Composable and modular code generation in mlir: A structured and retargetable approach to tensor compiler construction, 2022.

Optimization Notice: Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to <http://www.intel.com/performance>.

Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other

A Appendix: Reproducing results

Our benchmarks were executed on AWS public instances: `c6a.metal` (Zen3), `c6i.metal` (Cooper Lake), `c7i.metal-24xl` (Sapphire Rapids), `c7a.metal-48xl` (Zen4), `hpc7g.16xlarge` (Graviton 3) for the results outlined above.

Most instances are *metal* instances to avoid other users interfering with our measurements. The only one that isn't is the Graviton 3, which has scalability problems on the *metal* instances but not on the HPC instances. For this reason we reserve an entire virtualized Graviton 3 instance in order to achieve the same result and not allow other tenants on our machines. We use a standard Amazon Linux OS on those instances.

We have fixed the state of our git repository in Github by creating a branch called `cgo-c4ml-2024`, from which these benchmarks were executed. To reproduce the benchmark, clone our repository on that branch, enter the `scripts/benchmarks` directory and run the `build_and_run.sh` script, following the instructions in the README file ⁸.

We have used these instructions directly to run all of the numbers on this paper.

⁸<https://github.com/plaidml/tpp-mlir/tree/cgo-c4ml-2024/scripts/benchmarks>