Universiteit Leiden

Compiling CUDA Tile IR to multi-core CPUs using MLIR

Thomas Schaap (s3946363)

Supervisors:
Dr. K.F.D. Rietveld & Ir. W.J. Van den Berg

Bachelor Thesis – Informatica

Leiden Institute of Advanced Computer Science (LIACS)
liacs.leidenuniv.nl

July 1, 2026

Abstract

CUDA Tile IR is a tile-based intermediate representation for expressing computations as operations on statically shaped tiles rather than explicit SIMT threads. This higher-level abstraction simplifies the expression of tensor computations and exposes optimization opportunities while leaving the mapping to hardware resources to the compiler. CUDA Tile IR only has a proprietary compiler for NVIDIA GPUs. However, the tile abstraction itself is not inherently tied to GPU hardware. Tiles expose structured computation, data locality, and parallelism in a way that is largely independent of the underlying architecture. If these abstractions can be mapped efficiently to CPUs, CUDA Tile IR could become a more portable representation that enables the same high-level program to target multiple hardware platforms. This thesis investigates that by implementing a compiler that targets x86 CPUs using the MLIR infrastructure. The evaluation demonstrates that the generated code achieves performance comparable to both handwritten C++ implementations and Triton's CPU backend.

Contents

1 Introduction

Modern high-performance computing increasingly relies on specialized hardware accelerators such as Graphics Processing Units (GPUs). These devices provide massive parallelism and high memory bandwidth, making them particularly effective for machine learning, scientific computing, and other data-intensive workloads. However, programming directly for GPUs often requires reasoning about low-level execution details such as threads, thread blocks, synchronization, and memory hierarchies. As a result, several higher-level programming models and intermediate representations have been developed to improve programmer productivity while still enabling efficient execution.

One recent example is CUDA Tile IR, an intermediate representation developed by NVIDIA for expressing GPU computations using tiles rather than explicit SIMT threads. Instead of describing how individual threads operate, CUDA Tile IR represents computations in terms of structured tile operations and leaves the mapping to hardware resources to the compiler. This abstraction simplifies the expression of tensor computations and enables compiler-driven optimization. CUDA Tile IR is available as an open-source MLIR dialect and serves as the compilation target for the cuTile programming framework. However, current compilation support for CUDA Tile IR is limited to NVIDIA GPUs, and no open compiler exists that targets alternative hardware architectures.

The lack of a alternative backend for CUDA Tile IR motivates the work presented in this thesis. Rather than lowering Tile IR directly to low-level CPU instructions, this work investigates whether the existing MLIR ecosystem can be leveraged to compile Tile IR programs efficiently for x86 processors. We implement a compiler that aims to preserve the tile-based structure of computations for as long as possible by translating Tile IR operations into standard MLIR tensor and linalg abstractions. This allows the compiler to reuse existing MLIR optimizations before lowering the program to LLVM IR and machine code.

We evaluate the compiler by comparing the runtime performance of the generated code against equivalent implementations produced by a C++ compiler and the Triton CPU backend. In addition to overall performance, we investigate the effectiveness of several compiler optimizations, including the lowering of gather and scatter operations and the impact of tiling and fusion on execution time and generated code size.

This thesis is organized as follows. Chapter 2 introduces the background on compilers, MLIR, and CUDA Tile IR. Chapter 3 describes the design and implementation of the proposed compiler. Chapter 4 presents the performance evaluation and analyzes the impact of the implemented optimizations. Chapter 5 discusses the current limitations of the compiler and identifies future work. Finally, Chapter 6 concludes the thesis.

2 Background

2.1 Compilers

A compiler is a program that translates source code written in a programming language into executable machine code. Since programming languages are designed for human readability and machine code is designed for direct execution by hardware, compilers must bridge the semantic gap between the two. In addition to translation, compilers are responsible for detecting errors, analyzing program behavior, and improving performance through various optimizations.

To manage this complexity, compilation is typically divided into several stages. First, the source code is parsed and checked for syntactic and semantic correctness. The compiler then performs analyses to understand properties of the program, such as data dependencies, control flow, and memory usage. Based on this information, it applies transformations and optimizations that improve performance, reduce memory consumption, or expose parallelism. Finally, the program is lowered into a form that can be translated into machine instructions for the target architecture. By organizing compilation as a sequence of analysis, transformation, and code generation stages, compilers can translate high-level programs into efficient executable machine code.

2.2 MLIR

2.2.1 IR

Compilers typically compile source code to machine code in multiple stages. During this process, they usually generate one or more intermediate representations, or IRs, of the program. An IR represents the program in a form that is easier for the compiler to analyze and transform than the original source code. Optimizations and analyses are then performed on this representation before it is eventually lowered to machine code. Traditional compiler infrastructures often provide a fixed IR at a particular level of abstraction. This works well when the source language and target domain fit that abstraction, but it becomes limiting when a compiler needs to preserve higher-level semantic information or model domain-specific constructs. As a result, compiler implementations often introduce additional IRs before lowering to a more general backend representation. MLIR was designed to address this problem by providing an extensible and modular compiler infrastructure [4]. Instead of defining a single fixed IR, MLIR provides common infrastructure for defining and combining multiple IRs. These IRs are organized into dialects, which group related operations, types, and attributes. Different dialects can coexist in the same program, allowing a compiler to progressively lower only the parts of the program for which a lower-level representation is appropriate.

2.2.2 Standard dialects

Several standard MLIR dialects are commonly used in lowering pipelines. The `arith` dialect contains arithmetic operations such as addition, multiplication and subtraction. The `scf` dialect represents structured control flow such as loops and conditionals. The tensor dialect operates on immutable tensor values. The `memref` dialect represents memory buffers and memory access operations. Together, these dialects provide reusable building blocks for progressively lowering

higher-level programs toward executable code. The lowest level dialect is often the LLVM dialect, which is translated to LLVM IR [3] for further lowering to machine code.

The linalg dialect provides structured linear algebra operations on tensors and memrefs, such as matrix multiplication, convolutions, reductions, and generic elementwise computations. Instead of representing these computations as explicit loop nests, linalg operations describe their iteration spaces, indexing patterns, and computation semantics as a single operation with a region. The central operation is `linalg.generic`, which all other operations in the linalg dialect are based on. Other named operations in the linalg dialect such as matrix multiplication are just predefined forms of `linalg.generic`. Representing operations in this way allows the compiler to perform optimizations on these structures without having to recover it from explicit loop nests [14].

2.2.3 Operations

The basic unit of MLIR is the operation. Operations consume and produce typed static single assignment (SSA) values, meaning that each value is defined once and can then be used by later operations. An operation may also contain regions and blocks, which makes it possible to represent both simple instructions and structured constructs such as loops, functions or branches in if-statements. Operations may also have attributes that specify constant values. Examples of this are declaring constant variables, or specifying that an integer addition cannot overflow. Operations belong to dialects, and their exact semantics are defined by the dialect that introduces them.

2.2.4 Type system

MLIR also has an extensible type system. Every SSA value has a type. Dialects may define additional types when needed. Common MLIR types include scalar integer and floating-point types, tensor types for abstract multi-dimensional values, memref types for multi-dimensional memory buffers, and vector types for SIMD-style values. Tensors are useful for representing computations in a value-based form. They do not specify a concrete memory layout and are treated as immutable SSA values. This makes them well suited for high-level transformations such as tiling, fusion, canonicalization, and other rewrites before committing to a real memory representation. The actual memory representation is done by the memref type. A memref contains information about sizes, offsets and strides of the data. These values could be static or determined at runtime.

2.2.5 Passes

Compiler transformations in MLIR are implemented by defining rewrite patterns. These passes operate on IR units such as modules, functions, or operations, and may either target a specific dialect or use generic interfaces shared by many operations. A transformation pass rewrites the IR while staying at roughly the same abstraction level, for example by simplifying operations, tiling loops, or canonicalizing expressions. A conversion pass lowers operations from one dialect or abstraction level to another. It involves marking specific dialects, or sometimes individual operations, as ‘illegal’. The illegal operations are then replaced by operations from lower level dialects.

2.3 CUDA Tile

2.3.1 Tile IR

CUDA Tile [7] is a tile based programming model for NVIDIA GPUs. It allows developers to express tile based kernels using a python based DSL, cuTile python [9]. Programs written in cuTile are lowered to CUDA Tile IR [8]. Tile IR is an intermediate representation for programming NVIDIA GPUs as tile-based processors rather than as explicit SIMT thread machines. This is different from CUDA C++ or PTX, where the programmer reasons more directly about threads, warps, thread blocks, and their mapping to data. In Tile IR, the programmer or compiler describes how a problem is partitioned into tiles, while the Tile IR compiler is responsible for mapping those tiles onto hardware resources such as threads, tensor cores, and the memory hierarchy. The Tile IR specification is available as an open source MLIR dialect. However, the final lowering from Tile IR to NVIDIA GPU hardware is done using a closed source compiler. Additionally, no compiler currently exists for Tile IR that targets other hardware than NVIDIA GPUs.

2.3.2 Types

Several element types exist in Tile IR. Element types include signless integers such as i1, i8, i16, i32, and i64, IEEE floating-point types such as f16, f32, and f64, and hardware-oriented floating-point formats such as tf32, bf16, e4m3, and e5m2. Scalar values are not a separate top-level type. Instead, they are represented as rank-0 tiles, for example `tile<f32>` is used for a scalar 32 bit floating point value.

The `tile` type is the core compute type. A tile is a statically shaped tensor whose extents are known at compile time and must be powers of two. For example, `tile<2x8xf32>` denotes a two-dimensional tile of `f32` values. Tile values are immutable SSA values. Their physical register or memory layout is intentionally not visible in the IR semantics, which means that the compiler may choose an efficient representation for the target architecture.

Pointers are written as `ptr<E>`, where E is the non-pointer element type stored at the referenced memory location. A pointer is a 64-bit address into global device memory, and pointer arithmetic is defined in terms of the storage size of the pointee type. Tile IR can also form tiles of pointers, such as `tile<4x8xptr<f32>>`. This represents a two-dimensional tile whose elements are addresses. Loading such a tile of pointers produces a tile of scalar values with the same shape. This supports gather/scatter-like access patterns, because the pointer elements do not need to refer to contiguous memory.

The `tensor_view` describes global memory. It contains an element type, a shape, and strides. Unlike `tile`, a tensor view may include dynamic extents or strides, which are bound at runtime when the view is constructed. However, a tensor view cannot be loaded directly. It must first be subdivided tiles of static size. Subview types provide that subdivision. The only subview that currently exists is the `partition_view`, which partitions a tensor view into a grid of non-overlapping, statically sized tiles. For example, a large matrix view can be partitioned into tiles of size 128x4 with element type `i32`. Loading at a (possibly dynamic) partition index will give a `tile<128x4xi32>`. Partition views can also encode a dimension map and padding behavior. At boundaries, out-of-bounds

elements can be replaced by a padding value on load, while out-of-bounds stores are masked. More subview types are expected in future versions of Tile IR.

2.3.3 Operations

Operations in Tile IR are primarily centered around the manipulation of tiles. The operations are grouped into several categories. Core operations cover basic tile manipulation, such as defining constants, broadcasting, concatenation, reshaping, permutation and element extraction. Conversion operations are used to explicitly cast between compatible integer, floating-point, pointer, and tile representations. Arithmetic operations are split into floating-point and integer families, reflecting differences in semantics such as rounding, NaN behavior, fast-math flags, signedness, and overflow assumptions. While many operations function as simple elementwise computations, Tile IR also includes more involved tile-level operations, such as the matrix multiplication/accumulation, reduction and prefix sum operations, which express larger structured computations over tiles. Memory operations cover loading from or storing to either a tile of pointers or a tensor view. These operations can use memory tokens, which explicitly represent ordering dependencies between memory effects. This makes memory ordering part of the IR instead of relying on textual program order. The compiler is free to reorder these memory operations in different threads if no token is provided. Finally, Tile IR includes structured control-flow operations such as conditionals and loops, along with their region terminators: break, continue, return, and yield, allowing tile computations to be embedded in program control flow. An example of Tile IR is shown in Listing 1. The top level operation of a Tile IR program is the module. The `entry` operation marks the entry for a tile kernel. Its arguments must be 0 dimensional tiles and it cannot have a return value. Non-entry functions are expected in future Tile IR versions.

```
cuda_tile.module @example_module {
  entry @add_constant_fn(%arg0 : tile<ptr<f32>>) {
    %a_val = constant <f32: [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]> : tile<8xf32>
    %b_val = constant <f32: [9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0]> : tile<8xf32>

    %c_val = addf %a_val, %b_val rounding<nearest_even> : tile<8xf32>

    %offset = iota : tile<8xi32>
    %ptr_base = reshape %arg2 : tile<ptr<f32>> -> tile<1xptr<f32>>
    %ptr_broad = broadcast %ptr_base : tile<1xptr<f32>> -> tile<8xptr<f32>>
    %ptr_tile = offset %ptr_broad, %offset :
      tile<8xptr<f32>>, tile<8xi32> -> tile<8xptr<f32>>

    store_ptr_tko weak %ptr_tile, %c_val : tile<8xptr<f32>>, tile<8xf32> -> token
  }
}
```

Listing 1: A Tile IR kernel that performs an addition on two constant tiles of 8 `f32` elements and stores the values at a given address.

2.4 Related Work

Compiling code meant to run on GPUs for multi-core CPUs has been done before by Polygeist [5]. Polygeist is a compiler that raises C and C++ programs into MLIR in a form that is suitable for polyhedral optimization and other high-level transformations. It preserves source-level structure such as structured control flow and multidimensional arrays, raises suitable regions to MLIR's Affine dialect, applies polyhedral scheduling and additional optimizations, and then lowers the result toward executable code. Polygeist was later extended to compile traditional CUDA C/C++ code to multi-core CPUs [6], where CUDA kernels are represented using high-level parallel constructs and then lowered to multi-core CPU code using OpenMP. By performing transformations on a high-level MLIR representation, the compiler can preserve program structure while applying optimizations before lowering to CPU code. This extension achieved a 58% geometric-mean speedup over handwritten OpenMP implementations on the Rodinia CUDA benchmark suite and outperformed PyTorch's native CPU backend by 2.7 times on the CPU-only Fugaku supercomputer using a PyTorch compatibility layer.

Earlier work on GPU-to-CPU compilation includes MCUDA [11], which translates CUDA source code to C by converting CUDA threads into nested loops, and GPU Ocelot [1], which executes PTX programs on CPUs through dynamic translation to LLVM IR. Similar techniques have also been adopted by COX [2], an LLVM transformation pass for CUDA that additionally supports warp-level primitives. These systems generally rely on splitting kernels at synchronization points to eliminate GPU barriers before execution on CPUs. These results demonstrate that the parallel structure of GPU programs can be exploited to generate efficient multi-core CPU code.

One of the first languages for writing tile-based GPU code was Triton [13]. Triton is a language and compiler for writing efficient GPU kernels, motivated by the difficulty of implementing new deep-learning primitives that are not covered by vendor libraries such as cuBLAS and cuDNN. Similarly to cuTile, the Triton programming model is based on tiles. Instead of forcing programmers to write low-level CUDA code or rely only on high-level tensor DSLs, Triton exposes tile-level operations such as loads, stores, transposition, and dot products, while leaving many hardware-specific details to the compiler. This makes it a middle layer between high-level frameworks and hand-written SIMT GPU kernels. Originally, Triton was compiled using a custom IR based on LLVM IR. Nowadays, Triton uses the MLIR infrastructure. It has a custom dialect which encodes information to choose layouts that map memory operations efficiently onto GPU threads. After these Triton-specific analyses and transformations, much of the remaining program is lowered directly into the LLVM dialect inside MLIR, with target-specific operations or intrinsics where needed. There is also a CPU backend for Triton [10], which provides passes to lower the tensor based Triton operations directly to operations based on MLIR's vector type. The programming models of Triton and CUDA Tile are similar. The main difference is that Triton is a domain-specific language with its own compiler IR, while Tile cuTile is a frontend for CUDA Tile IR, making Tile IR a central abstraction which can be targeted from multiple languages.

3 Design and Implementation

This chapter describes the design and implementation of the CUDA Tile IR to x86 CPU compiler. The compiler lowers CUDA Tile IR kernels to executable CPU code through a sequence of MLIR transformations. The main design choice is to translate most CUDA Tile IR operations to MLIR's tensor and `linalg` abstraction level first, instead of immediately scalarizing tile operations into loops or lowering directly to lower level dialects such as `vector`. This keeps tile computations structured for long enough to apply existing MLIR optimizations such as fusion, tiling, vectorization, bufferization, and lowering to LLVM IR.

CUDA Tile IR is primarily organized around tiles. The CPU compiler preserves this structure by mapping non-scalar tiles to ranked tensors. Element-wise tile operations are lowered to tensor-level arithmetic and later represented through `linalg` operations. More structured operations, such as reductions and matrix multiplication, are lowered directly to `linalg` operations. In some cases we do have to break away from the standard MLIR pipeline and create our own dialect, for example to work with pointer values.

The overall pipeline progressively lowers CUDA Tile IR to standard MLIR dialects code through a sequence of compiler passes:

1. **ConvertCudaTileToStandard**: Convert CUDA Tile IR operations and types to standard MLIR dialects and the temporary `cuda_tile_cpu` dialect.
2. **TileAndFuseIntoStore**: Tile large tensor computations and greedily fuse producers into store operations.
3. **VectorizeLinalg**: Vectorize the resulting `linalg` operations using MLIR's existing vectorization infrastructure.
4. **LowerCudaTileCPUMemOps**: Lower Tile IR memory operations to either `vector` based load and store operations or explicit `scf` loops.
5. **InsertParallelLoops**: Introduce `scf.parallel` loops to represent tile-block parallelism.
6. **ConvertCudaTileCPUToLLVM**: Lower the remaining `cuda_tile_cpu` operations to LLVM-compatible operations.

Besides these custom passes, the remaining lowering is performed by many standard MLIR passes, including bufferization, OpenMP conversion, vector lowering, memref-to-LLVM conversion, and final translation to LLVM IR. This allows the compiler implementation to focus primarily on CUDA Tile IR-specific semantics while reusing as much of the existing MLIR infrastructure as possible. An overview of all supported Tile IR operations by the cuda-tile-cpu compiler is provided in Appendix B.

3.1 Dialect and Passes

The implementation adds a small auxiliary dialect named `cuda_tile_cpu`. This dialect acts as a temporary bridge dialect used for operations that cannot be cleanly expressed in existing MLIR

dialects at the point where they are introduced. It contains operations for printing, constructing memrefs from raw pointers, loading and storing tiles through structured memrefs, loading and storing tiles through tensors of pointers, scalar pointer loads and stores, and temporary tile grid-query operations.

The first custom pass is `ConvertCudaTileToStandard`. This pass performs the initial conversion from CUDA Tile IR to standard MLIR dialects. CUDA Tile modules and entry operations are rewritten to ordinary MLIR module and function structure. Constants, broadcasts, reshapes, concatenations, transposes, extraction operations, arithmetic operations, reductions, matrix multiplications, and control-flow operations are converted to dialects such as `arith`, `math`, `tensor`, `linalg`, `scf`, `memref`, and `func`. Operations that still need CPU-specific handling, especially memory operations and printing, are rewritten to `cuda_tile_cpu` operations.

The pass also performs the main type conversion. Non-scalar CUDA Tile values become ranked tensors. Every tile value is mapped to an equivalent tensor value that has the same size and amount of dimensions as the original tile. Zero-dimensional tiles become scalars. Tensor views become memrefs with strided layout information. Pointer values are represented as 64-bit integer addresses until the final LLVM lowering. This type conversion is described in more detail in section 3.2.

The second custom pass is pass `TileAndFuseIntoStore`. It tiles large tile computations. It uses store operations as sinks, tiles their iteration domains, and greedily fuses producers of the generated slices into the tiled loops. This avoids materializing large intermediate tiles when only smaller pieces are needed for the final store. It is especially useful for large tile sizes that are natural in a GPU-oriented IR but inconvenient for CPU vector registers and caches.

The `VectorizeLinalg` pass applies MLIR's existing `linalg` vectorization infrastructure. At this point, many CUDA Tile operations have become `linalg` operations over statically shaped tensors. The pass checks whether a `linalg` operation has a vectorization implementation and, if so, replaces it with that vector operations. Later standard MLIR passes lower these vector operations towards LLVM IR.

The pass `LowerCudaTileCPUMemOps` lowers the temporary CPU memory operations introduced by the first pass. These temporary operations acted on the tensor type rather than the lower level vector and memref types. This was needed for participation in tiling and fusion. Loads and stores through structured tensor views are lowered to `vector.transfer_read` and `vector.transfer_write`. These operations are a good fit for view-based tile accesses because they represent shaped reads and writes from strided memory and can express out-of-bounds behavior through masks and padding. Loads and stores through tensors of pointers are lowered to `vector.gather` and `vector.gather` if possible. If this is not possible they are lowered to explicit `scf.for` loops as a fallback. These loops implements gather and scatter behavior by loading or storing each tile element through its corresponding scalar address.

The `InsertParallelLoops` pass introduces CPU parallelism. It adds three grid-size arguments to each kernel function and inserts a three-dimensional `scf.parallel` loop inside the kernel. The original kernel body is moved into this loop. Operations that query the current tile block id are

replaced by the loop induction variables, and operations that query the number of tile blocks are replaced by the loop upper bounds. The `scf.parallel` loop is later converted to OpenMP by standard MLIR passes.

The final custom conversion pass is `ConvertCudaTileCPUToLLVM`. It lowers the remaining `cuda_tile_cpu` operations to LLVM-compatible operations. Print operations become calls to a small runtime library. Scalar pointer loads and stores convert integer addresses back to LLVM pointers and emit LLVM loads and stores. The `make_memref` operation constructs an LLVM-level memref descriptor from a raw pointer, sizes, and strides. After this pass, the remaining conversion is handled by standard MLIR lowering passes.

The full compiler pipeline also uses many built-in MLIR passes, including canonicalization, common subexpression elimination, element-wise-to-linalg conversion, empty tensor elimination, one-shot bufferization, buffer hoisting, buffer deallocation, scalar replacement, vector lowering, OpenMP lowering, memref-to-LLVM lowering, and final conversion to LLVM IR. These passes are not specific to CUDA Tile IR, but they are essential to the implementation because they allow the custom compiler to remain small and focused on the parts that require CUDA Tile IR semantics. The full compilation pipeline, as shown through use with the command line `cuda-tile-opt` tool is shown in Appendix C.1. This tool is an extension of the `mlir-opt` tool. It allows users to apply specific MLIR transformation and optimization passes on a provided input IR.

3.2 Type conversion

CUDA Tile IR and CPU-oriented MLIR dialects use different abstractions for the same data. CUDA Tile IR has tile types, pointer types, tensor view types, and partition view types. The CPU lowering converts these into scalars, ranked tensors, integer pointer representations, and memrefs.

Tile IR Type	**Converted MLIR Type**
tile<E>	E
tile<?xE>	tensor<?xE>
tile<ptr<E>>	i64
tile<?xptr<E>>	tensor<?xi64>
tensor_view<?xE, strides=[?]>	memref<?xE, strided=[?]>
parition_view	No equivalent. Information of this type will be encoded in cuda_tile_cpu operations.

Table 1: Type conversion rules for converting Tile IR types to standard MLIR types. The ? symbol represents any amount of dimensions.

Non 0d tiles are converted to ranked tensors. A CUDA Tile value of type `tile<shape x element-type>` is converted to `tensor<shape x element-type>`. This preserves the tile as a whole value and allows linalg, tensor, and vector transformations to operate on it. For example, a tile of 32 `f32` values becomes `tensor<32xf32>`, and a two-dimensional tile of shape 16 by 16 becomes `tensor<16x16xf32>`.

Zero-dimensional tiles are handled differently. A zero-dimensional tile contains one element, but semantically it is often used as a scalar value. Converting such a tile to `tensor<element-type>` would create unnecessary tensor and memref structure after bufferization. More importantly, for pointer-like values it would produce the wrong ABI shape. For example, converting a zero-dimensional tile containing a pointer to a zero-dimensional tensor would eventually bufferize to a memref, effectively passing or storing a pointer to a pointer-like container. These are not the intended semantics. Therefore, zero-dimensional tiles are converted directly to their element type. A `tile<i32>` with empty shape becomes `i32`, and a `tile<f32>` with empty shape becomes `f32`. This rule also simplifies control-flow conditions, scalar indices, and scalar pointer operations.

CUDA Tile pointer types are represented as 64-bit integers during most of the CPU pipeline. While MLIR does have a `ptr` dialect and a type representing a pointer in the LLVM dialect, we choose not to use this. Standard MLIR lowering passes do not support these built in pointer representations as elements of tensors, vectors or memrefs. Using `i64` to represent pointers prevents this problem. Later, during LLVM conversion, scalar pointer loads and stores turn these integer addresses back into LLVM pointer values using `inttoptr`. This keeps pointer arithmetic explicit and avoids introducing LLVM pointer types too early.

Tensor views are converted to memrefs. A CUDA Tile tensor view contains shape and stride information, so it naturally maps to an MLIR `memref` type with a strided layout. The compiler uses a memref shape corresponding to the tensor view shape. Dynamic sizes and dynamic strides are passed through the temporary `cuda_tile_cpu.make_memref` operation and later become fields of the LLVM memref descriptor.

Partition views are also converted to memrefs, but the partition-specific information is not represented in the memref type itself. A partition view describes how a tensor view is divided into tile-sized regions. The memref stores the underlying memory view, while partition information such as tile shape, dimension mapping, and padding is used when lowering load and store operations. In other words, the partition view is eliminated as a value, but its metadata is consumed by the conversion patterns that compute tile offsets and padding behavior.

Currently, Tile IR only supports passing scalar arguments to a function. Tensors must be passed as a pointer. Return values are also not supported, the kernel is only executed to have an effect on the global memory passed by the pointers. Therefore we can use the same type conversion rules for function signatures as we do for the types in the function body, meaning that the rank-0 tile elements in the signature are converted to a scalar type. An overview of all the type conversions is shown in Table 1.

3.3 Element-wise operations

Element-wise operations are the simplest class of CUDA Tile computations to lower. A CUDA Tile element-wise operation applies the same scalar operation independently to each element of one or more input tiles. In the tensor-level representation, this maps naturally to tensor element-wise operations. In the `ConvertCudaTileToStandard` pass, the compiler first rewrites many of these

operations to `arith` or `math` operations over the converted operands. Subsequent MLIR passes convert tensor element-wise operations into linalg form.

Integer element-wise operations include addition, subtraction, multiplication, shifts, bitwise operations, comparisons, min/max, division, remainder, negation, and integer extension or truncation. Signedness-sensitive operations are mapped to the corresponding signed or unsigned MLIR operation. For example, signed and unsigned comparisons become different `arith.cmpi` predicates, and signed and unsigned remainder become either `arith.remsi` or `arith.remui`. Integer attributes are partially represented where MLIR supports them, although not every CUDA Tile overflow or rounding attribute has a direct MLIR equivalent. Floating point element-wise operations are work in the same way. They are converted to an equivalent operation in the standard MLIR `arith` or `math` dialect.

Element-wise lowering benefits from keeping the computation at tensor/linalg level for several passes. Linalg element-wise operations can be fused with producers and consumers, tiled into smaller operations, vectorized, and bufferized. If the compiler instead scalarized element-wise operations immediately, these optimizations would have to rediscover tile structure from loops. But keeping operations structured avoids that problem.

3.4 Loads and Stores

Memory operations are more complex than pure computation because Tile IR supports both structured view accesses and pointer-tile accesses. The CPU compiler uses different lowering strategies for these two cases. Structured view accesses are lowered using vector transfer operations. Pointer-tile accesses are lowered using scalar loops that implement gather and scatter.

3.4.1 Views

A tensor view represents a structured memory object. It has a base pointer, shape, and strides. A partition view represents a tile-shaped partitioning of that tensor view. Loading from and storing to partition views is done using the `load_view_tko` and `store_view_tko` instructions. An example is shown in Listing 2. It first constructs a tensor view with static sizes and strides. Then a partition view is created. This indicates the tile size that should be loaded from the tensor view, in this case 64x64. Finally the view is loaded into a tile. It loads at index (0, 0), meaning that a 2 dimensional tile is loaded from the memory. The address calculation from the indices is based on the partition tile size, meaning that if the index is increased to (0, 1), this would skip the first 64 `f32` values of the underlying memory in the corresponding dimension. Furthermore, sizes of the tensor view could be dynamic. If the load of a partition view lies outside the tensor view, a padding value will be inserted. The padding value that should be inserted is part of the tensor view type. By default it inserts an undefined value.

To lower views to a standard MLIR type, the initial conversion pass rewrites the construction of tensor views to a custom `cuda_tile_cpu.make_memref` operation. This operation is intended to construct a memref from a raw pointer as memrefs carry similar information to tensor views. Other than the pointer value itself, it also supports dynamic size and stride operands to construct the full

```
entry @example(%ptr: tile<ptr<f32>>) {
 %tensor_view = make_tensor_view %ptr, shape=[8192, 128], strides=[128, 1]
   : tensor_view<8192x128xf32, strides=[128,1]>

 %view = make_partition_view %tensor_view : partition_view<tile=(64x64),
  ↪  tensor_view<8192x128xf32, strides=[128,1]>>

 %c0 = constant <i32: 0> : tile<i32>

 %tile0, %res_token0 = load_view_tko weak %view[%c0, %c0]
   : partition_view<tile=(64x64), tensor_view<8192x128xf32, strides=[128,1]>>,
   ↪  tile<i32> -> tile<64x64xf32>, token
}
```

Listing 2: A Tile IR function that constructs a partition view and loads a tile from the view.

memref if those are not available at compile time. In the `CudaTileCPUToLLVM` conversion pass, the memref is converted to its runtime structure. This is the time where the pointers, sizes and strides are inserted into the struct if needed.

Initially, `load_view_tko` and `store_view_tko` operations are converted to intermediate operations `load_memref_tile` and `store_memref_tile`, which operate on MLIR tensors rather than tiles. The indices of the load or store are multiplied by the partition view tile size to represent the location of the first element to load in the underlying array. In the `LowerCudaTileCPUMemOps` pass, these tensor based ops are lowered to vector based ops that are part of the standard MLIR vector dialect. This pass converts `load_memref_tile` to a `vector.transfer_read`, which conveniently also accepts a padding value. The result of the transfer read is a vector with the same shape and element type as the tile. The pass then writes that vector into an empty tensor using `vector.transfer_write`, giving the rest of the tensor-based pipeline a tensor value. Conversely, `store_memref_tile` reads the tile tensor into a vector with `vector.transfer_read` and then writes the vector to the destination memref with `vector.transfer_write`. Writing back to tensors here is needed here because the values around it still expect tensors. These will be removed by following canonicalization passes, because a transfer write followed immediately by a transfer read cancels it out. An example of the lowering of Listing 2 can be seen in Listing 3.

3.4.2 Gather and Scatter

Another way to load and store values from global memory in Tile IR is through tiles that consist of pointers. Loading through such a tile can be seen as a gather operation, each output element is loaded from the address stored in the corresponding element of the pointer tile. Storing through such a value is a scatter, each input element is stored to the address stored in the corresponding element of the pointer tile. These addresses may be unrelated, so the compiler cannot model the operation as a single strided vector transfer. These types of gather and scatter loads are performed

```
func.func @example(%arg0: i64) {
  %padding = ub.poison : f32

  %c0_i32 = arith.constant 0 : i32
  %0 = cuda_tile_cpu.make_memref %arg0 sizes[] strides[] : i64 ->
   ↪  memref<8192x128xf32, strided<[128, 1]>>

  %c64 = arith.constant 64 : index
  %1 = arith.index_castui %c0_i32 : i32 to index
  %2 = arith.muli %1, %c64 : index

  %5 = vector.transfer_read %0[%2, %2], %padding : memref<8192x128xf32,
   ↪  strided<[128, 1]>>, vector<64x64xf32>
  return
}
```

Listing 3: Lowering of a partition view load to standard MLIR dialects and cuda_tile_cpu dialect to create the memref.

through the `load_ptr_tko` and `store_ptr_tko` operations. Both these operations also support masks, which are encoded as tile operands that have to be the same shape as the tile of pointers. For loads there can also be tile of padding values. This is different from how view-based loads handle padding, because those only support a single element of padding value that would be used for every out of bounds element.

To generate well vectorizable code, we try to lower this operation to a `vector.gather`. However, `vector.gather` works differently than `load_ptr_tko`. It requires a base pointer and a vector of offsets from the base pointer as two separate operands, whereas `load_ptr_tko` operates on the combination of the base pointers and the offsets as a single tile directly. Another difference is that the padding value of a `vector.gather` is represented as a single scalar that would be substituted for each masked element, rather than a tile of padding values which corresponds to each individual element. For this reason, we need to detect if it is possible to represent the pointer tile in the form of a base pointer and offsets, and if all elements in the padding tile are the same value. Listing 4 shows a sequence of operations that leads up to a `load_ptr_tko`. The compiler checks if the `%ptr_tile` value is defined by an `offset` operation. Then it attempts to follow the chain of `broadcast` and `reshape` operations. These operations only change the shape of the tile, not the value. If this chain leads up to a scalar value, we have found the base pointer that can be used for the `vector.gather`. The padding value is determined to originate from a scalar value in the same way. If we find both the base pointer and a scalar padding value, we lower the `load_ptr_tko` to an intermediate `cuda_tile_cpu.gather_tile` operation first for the purposes of tiling and fusion. In the `LowerCudaTileCPUMemOps` pass it is finally converted to a `vector.gather`.

If we cannot detect that the scalar padding value and a base pointer, our compiler lowers

```
%cst_0_i16 = constant <i16: 0> : tile<i16>
%reshape_0 = reshape %cst_0_i16 : tile<i16> -> tile<1xi16>
%padding = broadcast %reshape_19 : tile<1xi16> -> tile<2048xi16>

%reshape_ptr = reshape %ptr : tile<ptr<i16>> -> tile<1xptr<i16>>
%bcast_ptr = broadcast %reshape_ptr : tile<1xptr<i16>> -> tile<2048xptr<i16>>
%ptr_tile = offset %bcast_ptr, %offsets : tile<2048xptr<i16>>, tile<2048xi64> ->
  ↪ tile<2048xptr<i16>>

%result, %tok = load_ptr_tko weak %ptr_tile, %mask, %padding token=%0 :
  ↪ tile<2048xptr<i16>>, tile<2048xi1>, tile<2048xi16> -> tile<2048xi16>, token
```

Listing 4: Tile IR Operations leading up to a `load_ptr_tko` operation that could be lowered to a `vector.gather`.

`load_ptr_tko` to explicit `scf.for` loops after converting it to an intermediate `cuda_tile_cpu` operation which acts on the standard `tensor` type. First, it reads the tensor of pointers as a vector. Then it shape-casts the pointers to a one-dimensional vector, because the standard MLIR lowering does not support dynamic indices for higher rank vector insert/extract operations. If the operation has a mask, the mask tensor is also read and shape-cast to a one-dimensional vector. If the operation has a padding value, that value is also made available in one-dimensional vector form. In each iteration of the loop the code extracts the pointer for the current element. If there is no mask, it performs a scalar pointer load. If there is a mask, it emits an `scf.if`. The true branch performs the load, and the false branch uses either the padding value or an undefined/poison value. The loaded scalar is inserted into an accumulating vector. After the loop, the one-dimensional result vector is shape-cast back to the tile shape and written into a tensor. Similar to views, this transfer write into the tensor can be removed through canonicalization patterns. The loop that is generated for a gather-like load with a mask is shown in Listing 5.

Scatter stores work in a similar way. First we check if we can lower the `store_ptr_tko` operation to a `vector.scatter` by finding a base pointer. If not, we use a similar scalar-loop structure. The compiler reads the pointer tile and value tile as one-dimensional vectors. It then loops over all tile elements, extracts the pointer and value for each element, checks the mask if present, and emits a scalar store for the active elements. An example of how a scatter with a mask is lowered using loops can be seen in Listing 6.

The loop based lowerings are less compact than vector transfers, but it accurately represents the memory semantics of arbitrary pointer-tile accesses. It also makes masks explicit and gives later scalar and loop optimizations of standard MLIR passes an ordinary SCF representation to work with.

```
%15 = scf.for %arg4 = %c0 to %c16 step %c1 iter_args(%arg5 = %0) ->
  ↪  (vector<16xi32>) {
  %20 = vector.extract %13[%arg4] : i64 from vector<16xi64>
  %21 = vector.extract %14[%arg4] : i1 from vector<16xi1>
  %22 = scf.if %21 -> (i32) {
    %24 = cuda_tile_cpu.load_ptr %20 : i64 -> i32
    scf.yield %24 : i32
  } else {
    scf.yield %2 : i32
  }
  %23 = vector.insert %22, %arg5 [%arg4] : i32 into vector<16xi32>
  scf.yield %23 : vector<16xi32>
}
```

Listing 5: Gather load with a mask that loads i32 values from pointers after the LowerCudaTileCPUMemOps pass.

```
scf.for %arg4 = %c0 to %c16 step %c1 {
  %20 = vector.extract %11[%arg4] : i64 from vector<16xi64>
  %21 = vector.extract %19[%arg4] : i32 from vector<16xi32>
  %22 = vector.extract %14[%arg4] : i1 from vector<16xi1>
  scf.if %22 {
    cuda_tile_cpu.store_ptr %20, %21 : i64, i32
  }
}
```

Listing 6: Scatter store with a mask that stores a tile of i32 values after the LowerCudaTileCPUMemOps pass.

3.5 Tiling and Fusion

CUDA Tile IR programs often use tile sizes that are natural for GPU programming but too large to map directly to CPU vector registers or cache-local operations. For example, a tile may represent a large block of a matrix or a large set of element-wise computations. If such a tile is lowered directly, the intermediate tensor and vector operations may become large. This can increase register pressure or create large unnecessary temporary buffers.

The compiler addresses this using a tile-and-fuse pass centered on stores. The pass identifies tile store operations whose stored value has a static shape. These stores act as sinks for tile computations. The pass then uses MLIR's tiling interface to tile the identified sink operations. Since all store operations are in the `cuda_tile_cpu` dialect, we have to implement this interface ourselves. This is done by defining several properties about the operations such as the bounds

```
func.func @kernel(%arg0: i64) {
  %c0_i32 = arith.constant 0 : i32
  %c0 = arith.constant 0 : index
  %a = arith.constant dense<1> : tensor<128xi32>
  %b = arith.constant dense<2> : tensor<128xi32>
  %0 = arith.addi %a, %b : tensor<128xi32>
  %1 = cuda_tile_cpu.make_memref %arg0 sizes[] strides[] : i64 -> memref<128xi32,
   ↪  strided<[1]>>
  cuda_tile_cpu.store_memref_tile %0, %1[%c0] : tensor<128xi32>, memref<128xi32,
   ↪  strided<[1]>>
  return
}
```

Listing 7: Function at the cuda_tile_cpu level that adds two constants and stores them into a view before the tile and fusion pass.

of the iteration domain, and the iterator type (parallel or reduce). The requested tile sizes are applied to the innermost dimensions of the store iteration domain. Outer dimensions that are not explicitly defined can use a default tile size. After tiling the store, the pass greedily fuses producers of generated tensor slices into the created loops. Listing 7 shows a function before the tile and fuse pass, and Listing 8 shows a function after the tile-and-fuse pass. Here the compiler used a tile size of 16, however this value is configurable. A loop is created where the tensor values of the constants are extracted from a 128 element tensors to a tensor of 16 elements. The following `addi` operation now also acts on tensors of 16 elements. Internally, the store operation was the first operation that was tiled using a loop, and the tensor addition was fused into it.

The effect is that large tile computation which potentially uses multiple operations is transformed into a loop nest that computes and stores smaller slices. This reduces the need to materialize full intermediate tiles in the form of memrefs. It also exposes smaller linalg operations to vectorization, reducing potential register pressure when the original tile sizes are large. On a CPU backend, this is usually preferable because the machine has a different hierarchy from the GPU. Vector registers are smaller than the GPU tile based abstractions where each tile kernel could get mapped onto multiple threads. Cache locality is more important than exposing massive SIMT parallelism within one tile.

Tiling is especially useful for long chains of element-wise operations or operations feeding directly into a store. Producer fusion means that intermediate tensors can often disappear entirely after canonicalization and bufferization. Instead of allocating a buffer for each intermediate tile, the generated loop computes a slice and immediately forwards it to the consumer.

The current implementation uses store operations as the main tiling anchors. This is a practical choice because stores define the externally visible memory result of a tile computation. The store is tiled, and operations producing the tile leading op to that store are fused. However this model is not perfect. Large tile operations could happen inside regions that do not lead up to a

store, for example inside the region of a for-loop. Future work could use a more general cost model to choose tiling anchors, tile sizes, and fusion boundaries based on operation type, target vector width, cache size, or estimated memory traffic.

```
func.func @kernel(%arg1: i64) {
  %c0_i32 = arith.constant 0 : i32
  %a = arith.constant dense<1> : tensor<128xi32>
  %b = arith.constant dense<2> : tensor<128xi32>

  %1 = cuda_tile_cpu.make_memref %arg1 sizes[] strides[] : i64 -> memref<128xi32,
   ↪  strided<[1]>>

  %c0 = arith.constant 0 : index
  %c16 = arith.constant 16 : index
  %c128 = arith.constant 128 : index
  scf.for %iv = %c0 to %c128 step %c16 {
    %a_slice = tensor.extract_slice %a[%iv] [16] [1] : tensor<128xi32> to
     ↪  tensor<16xi32>
    %b_slice = tensor.extract_slice %b[%iv] [16] [1] : tensor<128xi32> to
     ↪  tensor<16xi32>

    %add = arith.addi %a_slice, %b_slice : tensor<16xi32>

    cuda_tile_cpu.store_memref_tile %add, %1[%iv] : tensor<16xi32>, memref<128xi32,
     ↪  strided<[1]>>
  }
  return
}
```

Listing 8: Function that adds two constant values after the tile and fusion pass.

3.6 Thread Level Parallelism with OpenMP

Tile IR exposes parallelism through a grid of tile blocks. On the GPU, tile blocks are part of the kernel launch structure. On the CPU, our compiler represents this grid as an explicit loop nest inside the kernel. To do so, we first add three extra arguments to the current function, which represent the number of tile blocks in each dimension. Then we create a three dimensional `scf.parallel` loop which loops over the x, y and z launch grid dimensions, the upper bounds of this loop are taken from the added function arguments, as can be seen in Listing 9. The full kernel body is moved into this parallel loop. At this stage, the `get_num_tile_blocks` and `get_tile_block_id` operations are also rewritten to simply refer to the loop upper bounds and loop induction variables respectively. This gives the kernel the same logical grid information it would have on the GPU. After the `scf.parallel` loop is inserted, standard MLIR passes can lower it to the OpenMP dialect where it generates an OpenMP loop nest.

```
func.func @example(..., %grid_x, %grid_Y, %grid_z) {
  %c0 = arith.constant 0 : index
  %c1 = arith.constant 1 : index
  scf.parallel (%iv1, %iv2, %iv3) = (%c0, %c0, %c0) to (%grid_x, %grid_y,
   ↪  %grid_z) step (%c1, %c1, %c1) {

    // ... kernel body
  }
}
```

Listing 9: Insertion of three dimensional parallel loop and function arguments into a kernel after the insert-parallel-loops pass.

The important design choice is that the OpenMP parallelism is inserted inside the kernel function rather than outside of it in a separate wrapper. This is unlike the implementation in Triton CPU, which creates a triple for loop in C++ in their kernel launch wrapper that calls the function. Defining the loop inside the kernel function itself keeps the whole kernel body visible to the optimizer. Values that do not depend on the block id can be hoisted out of the parallel loop by a loop invariant code motion pass. Similarly, subset hoisting and canonicalization can simplify operations that are independent of the grid iteration. If the compiler had instead generated an external host loop that repeatedly called the kernel body, the optimizer would not have the opportunity to move invariant work out of the loop.

3.7 Runtime

Most of the compiler output can be lowered to ordinary LLVM IR without a custom runtime. However there is currently one supported operation where this is not possible, which is the `cuda_tile.print_tko` operation. This operation can print both strings and tiles with printf-like format specifiers. Rather than parsing the format specifiers at runtime, the lowering of the print operation is done by splitting the string at each format specifier at compile time. We then introduce an intermediate print operation in the `cuda_tile_cpu` dialect which has a string operand and at most one memref operand, which corresponds to the tile. We generate this instruction for each split of the original string. At the LLVM stage, this intermediate print operation is turned into a runtime function call. The runtime is implemented as a small C++ library which exports symbols related to printing strings and multi-dimensional memref values. Kernels that make use of the print operation, usually for debugging purposes, must be linked against this library.

The generated LLVM IR can be compiled to an object file using the standard LLVM toolchain. To make the compiled kernels callable from other languages, the compiler preserves the original kernel name while lowering the function signature to a C-compatible ABI. In cuTile Python, kernels are written using a high-level interface in which array arguments are represented by a single parameter. For example, a kernel defined in Python as `def func(A)` appears to take only one array argument. During lowering to CUDA Tile IR, however, each array argument is expanded

into multiple parameters describing the underlying memory layout. A kernel taking a single array becomes:

```
cuda_tile_cpu.entry @func(%A: tile<ptr<E>>, %size: tile<i32>, %stride: tile<i32>)
```

The `ConvertCudaTileToStandard` pass lowers these Tile IR types to standard MLIR types, while the `InsertParallelLoops` pass introduces three additional arguments representing the grid dimensions. After conversion to the LLVM dialect, the function signature becomes:

```
llvm.func @func(%A: i64, %size: i32, %stride: i32,
                %gridX: i64, %gridY: i64, %gridZ: i64)
```

After lowering to LLVM IR and compiling to an object file, the function can be linked into a C or C++ program and invoked using the corresponding C function signature:

```
void func(int64_t A, int32_t size, int32_t stride,
          int64_t gridX, int64_t gridY, int64_t gridZ);
```

No additional runtime is needed when the `print_tko` operation is not used.

4 Evaluation

In this chapter we will evaluate the performance of the code generated by our compiler. The generated code is compared against equivalent implementations produced by a C++ compiler and by the Triton CPU backend. In addition, we test the effectiveness of lowering `load_ptr_tko` and `store_ptr_tko` to `vector.gather` and `vector.scatter` instead of lowering them to explicit loop nests. Lastly, we evaluate the impact of the compiler's tiling and fusion pass by measuring performance across different tile sizes. The instructions to run these benchmarks are listed in Appendix A.

4.1 Kernels

To evaluate the performance we select kernels that are used in image processing and kernels that are often used in large language models, as these can be represented well using a tile based programming model such as cuTile and Triton. The benchmarks of the Triton and C++ versions of these benchmark come from Terapines's AI-Benchmark repository [12]. We add the cuTile versions of the kernels ourselves. All the kernels we evaluate are as follows:

- Layer normalization
- SwiGLU
- Softmax
- Image sharpening
- Image resize with bilinear interpolation
- Horizontal image warp

We compile three versions of these kernels. First in C++ with OpenMP directives. For this version, we intentionally do not make use of compiler intrinsics, leaving the optimization up to the compiler. The second version is written in Triton with python. Finally we also create a cuTile python version. The Triton versions of the kernels often contain a constant integer which specifies the size of the tile. The programmer must explicitly loop over parts of the tile using this constant to generate efficent CPU code. For the cuTile version we do not have to this because the tiling and fusion pass will handle this. The IR for the cuTile and Triton versions are extracted into a seperate file. Then they are lowered using the triton-cpu and cuda-tile-cpu compilers to the LLVM dialect, after which they are lowered to LLVM IR where they can be linked with a C++ launcher. This launcher simply contains the function definitions of the kernels and utilities to collect execution time statistics. It also generates the input data that will be used. The full MLIR lowering pipeline that was used for the Triton code can be seen in Appendix C.2. The lowering pipeline for cuda-tile-cpu is shown in Appendix C.1.

4.2 Environment

For the C++ versions of the kernels, we compile with `g++` version 16.1.1. All kernels will be compiled with `-O3` and `-march=native` flags on the target machine. The Triton and CUDA Tile versions of the kernels will be compiled using specifc LLVM commits mentioned in Appendix A. The machine used for benchmarking has a Ryzen 5600x processor, which has 6 cores and 12 threads, and the machine has 16GB of dual channel DDR4 RAM at 2400 MT/s.

4.3 Runtime performance

We run all benchmarks with varying number of threads by setting the `OMP_NUM_THREADS`. Each benchmark is repeated 500 times to get the average execution time in milliseconds and the standard deviation. The result are listed in Figure 1.

Overall, the performance of the generated code is competitive with both Triton and the handwritten C++ implementations. For most kernels, the cuda-tile-cpu compiler either matches the performance of the Triton CPU backend or comes close to it. Across all kernels, the generated code scales well with increasing thread counts. The tiling and fusion pass also seems to be as effective as the manual tiling needed in the Triton for these kernels.

The swiglu and layernorm kernels perform particularly well. Their execution times are nearly identical to those of the Triton CPU backend, and they are consistently slightly faster than the C++ implementations. The softmax kernel also achieves competitive performance. These kernels primarily use `ct.load/store` in cuTile Python, which is lowered through tensor views to `vector.transfer_read` and `vector.transfer_write` operations. MLIR is able to lower these efficiently to masked vector load and store instructions, resulting in code that is comparable to that generated by the Triton CPU backend.

The resize kernel also performs well, despite relying on gather/scatter operations. It performs slightly better on average than the C++ version. Unlike Triton, our compiler is unable to lower the corresponding stores to contiguous vector stores, instead generating `vector.scatter` operations. Nevertheless, the performance remains competitive, suggesting that the kernel is primarily compute-bound rather than memory-bound, making the overhead of scatter stores relatively small. Additionally, some of the gather loads used in the kernel are not contiguous. This causes both the Triton CPU backend and our compiler to use a `vector.gather` here.

The warp and sharpen kernels exhibit the largest performance gap compared to Triton. Both kernels use more gather-like memory accesses than resize, meaning that a larger fraction of their execution time is spent performing gather and scatter operations. While our compiler lowers these operations directly to `vector.gather` and `vector.scatter`, Triton is able to recognize that many of these accesses are in fact contiguous and lowers them to masked vector loads and stores instead. These instructions are more efficient than general gather/scatter operations, which likely explains the remaining performance difference. The C++ versions work at the level of scalars, combined with a `#pragma omp simd` directive at the innermost loop. Evidently, the g++ compiler was also able to vectorize these loads well.

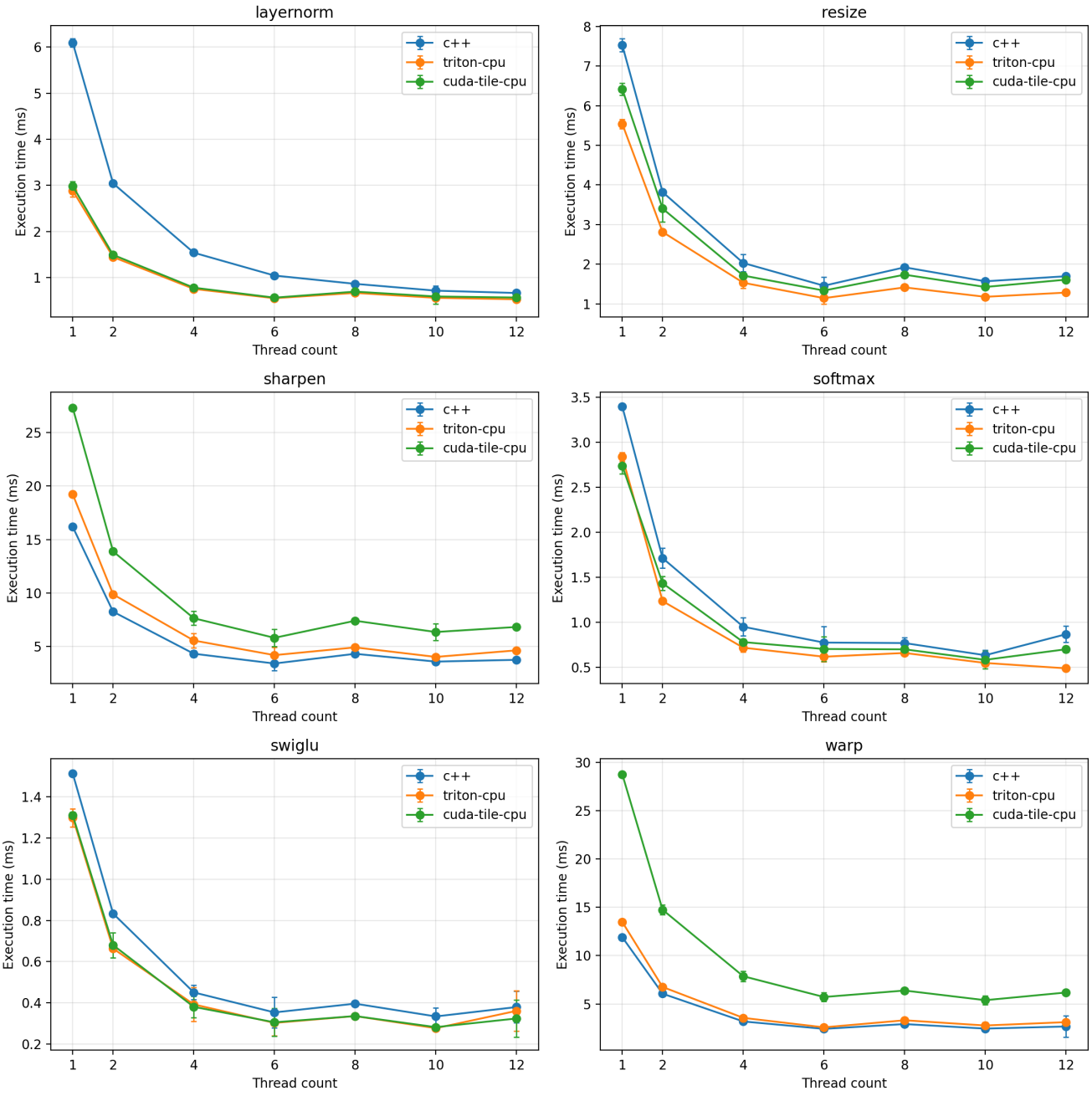

Figure 1: A side by side comparison of the execution time of all kernels in C++, triton-cpu and cuda-tile-cpu.

4.4 Gather and scatter lowering evaluation

To evaluate the effectiveness of the optimization that lowers Tile IR gather/scatter operations to the vector dialect instead of explicit `scf` loop nests, we compile a subset of the kernels with this optimization disabled. We evaluate only the resize, warp, and sharpen kernels because they contain `load_ptr_tko` and `store_ptr_tko` operations, which this optimization applies to. Figure 2 shows the results with `OMP_NUM_THREADS` set to 6.

Enabling this optimization significantly reduces execution time. The IR shows that all gather/scatter operations in these kernels are lowered to `vector.gather` and `vector.scatter` operations instead of explicit scalar loop nests. The resize and warp kernels achieve speedups of approximately 5×, while the sharpen kernel is approximately 1.7× faster. These results indicate that representing gather/scatter explicitly in the IR is significantly more effective than relying on LLVM to recognize the access pattern after lowering to scalar loops. LLVM appears unable to reliably recover the original gather/scatter semantics from the loop representation and therefore fails to generate efficient SIMD gather/scatter instructions.

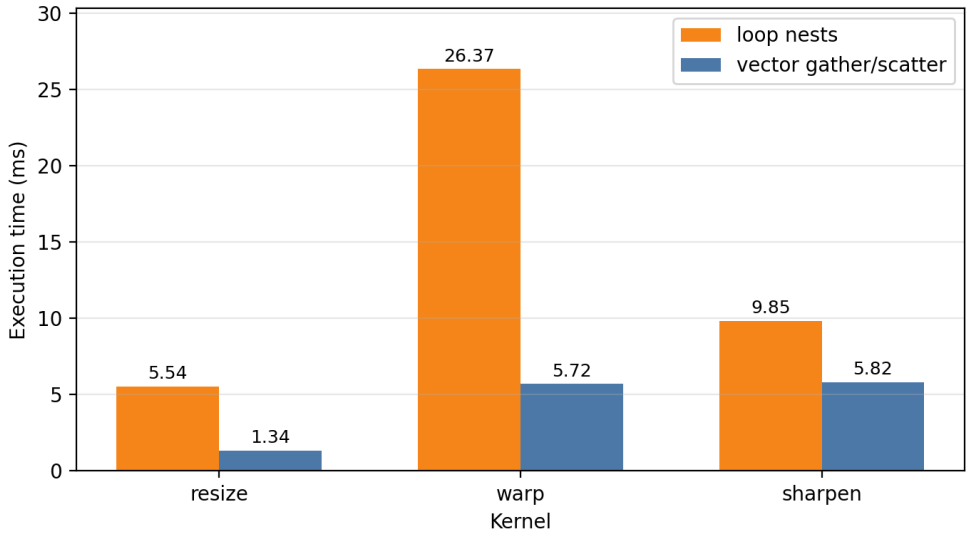

Figure 2: The difference in execution time using 6 threads for resize, warp and sharpen kernels when lowering gather and scatter like loads and stores to `vector.gather/scatter` operations instead of loop nests.

4.5 Tiling and fusion evaluation

To evaluate the effectiveness of the tiling and fusion pass, we compile the resize and warp kernels with tile sizes ranging from 1 to 512. Figure 3 shows the execution time for each tile size, while Table 2 reports the resulting binary sizes. For both kernels, a tile size of 16 yields the best performance. Smaller tile sizes expose less computation to the compiler and therefore provide fewer opportunities for fusion and vectorization. In contrast, larger tile sizes lead to a rapid increase in execution time. Although larger tiles reduce loop overhead, they also produce very large vector

operations that are expensive to lower on CPUs.

Tile operations are lowered to MLIR vector operations whose width is determined by the tile size. For very large tiles, such as `vector<512xi8>`, LLVM expands these operations into long sequences of SIMD instructions because no hardware instruction exists that operates on vectors of this size. Inspection of the generated assembly shows that large vector operations are transformed into thousands of repeated SIMD instructions, increasing both compilation output size and execution time.

The increase in generated code size is also visible in Table 2. For tile sizes up to 128, the binary grows gradually as larger vector operations are generated. Beyond a tile size of 256, however, the binary size increases dramatically, reaching over 450 KB for the resize kernel and over 600 KB for the warp kernel. This code-size explosion is consistent with the observed performance degradation, indicating that excessively large tiles prevent efficient lowering to the CPU's SIMD instruction set.

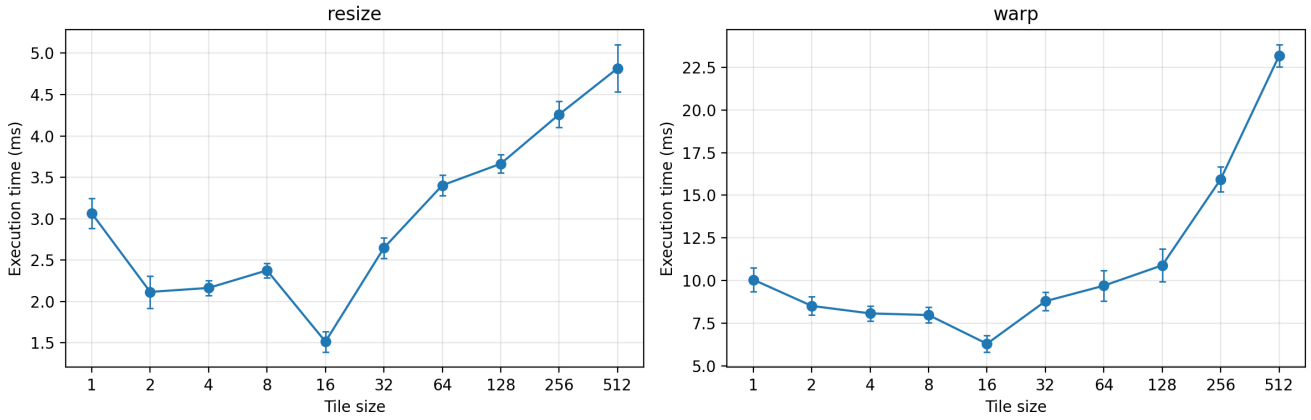

Figure 3: Average execution time in milliseconds for different tile sizes for the resize and warp kernels.

Kernel \ Tile size	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**
resize (KB)	18.48	18.48	18.48	26.48	26.48	34.48	54.48	86.48	450.48	466.48
warp (KB)	18.52	22.52	22.52	22.52	26.52	30.52	42.52	66.52	606.52	638.52

Table 2: Binary size in KB for different tile sizes for the resize and warp kernels.

5 Limitations and Future Work

The compiler is currently able to lower Tile IR gather and scatter operations directly to `vector.gather` and `vector.scatter` operations. However, unlike the Triton CPU backend, it cannot recognize when these gather or scatter operations actually correspond to accesses to contiguous memory. In such cases, a contiguous vector load or store would be more efficient than a gather or scatter. The issue originates from the way cuTile Python lowers array arguments to Tile IR. Instead of passing only a pointer, each array is represented by multiple kernel arguments containing the base pointer, sizes, and strides. Consequently, address calculations for `load_ptr_tko` and `store_ptr_tko` operations involve a multiplication by a runtime stride value. Although this stride is often known to be one, it is represented as a runtime value, preventing canonicalization from simplifying the address computation into the contiguous access pattern that the compiler recognizes. As a result, the compiler cannot distinguish between genuinely irregular memory accesses and accesses that are effectively contiguous.

A possible solution is to generate Tile IR using the newer cuTile C++ API instead of cuTile Python, as the C++ interface does not lower array arguments in this way. Alternatively, the cuTile Python frontend could be modified to replace stride arguments that are known to be one with constant values before generating Tile IR. Subsequent canonicalization would eliminate the redundant multiplications, exposing the contiguous address pattern and allowing the compiler to lower these operations to contiguous vector loads and stores instead of gathers and scatters.

Future work could also focus on the way tiling currently works. Selecting the 'store' operation as the sink for tiling and fusion works for simpler kernels, however by doing so there are situations where the compiler cannot detect operations on large tensors that could benefit from tiling. Examples are operations that contain inner regions, such as in if-branches and for-loops. Additionally, the tiling mechanism could be improved to work with operations that act like reductions. Since evaluation demonstrates that tile size has a significant impact on performance, other work could focus autotuning the tile sizes, as these are configurable in the compiler pipeline.

Other future work consists of adding support for operations that are currently not supported. This includes control flow, where currently the `break` operation is not implemented, and the `continue` operation only works in simple cases because MLIR does not currently have a way to model this cleanly in the `scf` dialect. However it might be possible to represent this control flow using by transforming the Tile IR `loop` and `for` operations to a form of `scf.while`. Another part of adding support for operations would be to update the compiler to work with newer Tile IR versions. The current compiler supports most of the Tile IR 13.2 operations, but at the time of writing Tile IR 13.3 has been released, which adds new view types besides `partition_view`.

Besides support for more operations, the compiler could also be integrated into cuTile Python. To run cuTile kernels, we currently extract the corresponding Tile IR representation manually, and launch it through C++. By integrating the compiler with cuTile Python, it could be possible to run CUDA Tile kernels from Python directly on the CPU.

6 Conclusion

CUDA Tile IR was designed as a tile-based intermediate representation for NVIDIA GPUs, but until now it only had a proprietary compilation path targeting NVIDIA hardware. This thesis demonstrated that CUDA Tile IR can also be compiled to x86 CPUs, improving the portability of the representation and showing that its abstractions are not inherently tied to GPU execution. By leveraging the MLIR infrastructure, a compiler was implemented that translates CUDA Tile IR kernels to executable CPU code while preserving tile-level semantics throughout much of the compilation pipeline.

The evaluation showed that the proposed approach generates efficient CPU code for a range of CUDA Tile IR kernels. For several benchmarks, the generated code matches the performance of the Triton CPU backend and in some cases slightly outperforms equivalent C++ implementations. The generated code also exhibits good scaling with increasing thread counts. While some kernels, such as warp and sharpen, still perform worse than Triton, the remaining performance gap is largely caused by the way cuTile Python lowers array arguments. Runtime stride values obscure contiguous memory accesses, so the compiler cannot recognize opportunities to replace gather and scatter operations with more efficient contiguous vector loads and stores. This limitation is therefore not inherent to CUDA Tile IR itself, but rather to the current frontend used to generate it.

The evaluation also demonstrated that tiling and fusion can be applied effectively when targeting CPUs. Although the current implementation has limitations in the way tiling opportunities are identified, the generated code achieves performance comparable to the Triton kernels that had to be manually tiled by the programmer to achieve good performance. Furthermore, the experiments show that tile size can have a significant impact on performance, with moderate tile sizes providing the best balance between optimization opportunities and generated code size.

Overall, the results demonstrate that Tile IR is a suitable intermediate representation for portable code generation beyond its original NVIDIA GPU target. They also show that MLIR provides an effective foundation for such a compiler. By many reusing existing dialects, analyses, and passes, a relatively compact implementation can generate competitive CPU code while retaining the flexibility to support additional optimizations and targets in the future.

References

[1] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, page 353–364, New York, NY, USA, 2010. Association for Computing Machinery.

[2] Ruobing Han, Jaewon Lee, Jaewoong Sim, and Hyesoon Kim. COX: Exposing CUDA warp-level functions to CPUs. *ACM Trans. Archit. Code Optim.*, 19(4), September 2022.

[3] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, pages 75–86, USA, 2004. IEEE Computer Society.

[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] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. Polygeist: Raising C to Polyhedral MLIR. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 45–59, 2021.

[6] William S. Moses, Ivan R. Ivanov, Jens Domke, Toshio Endo, Johannes Doerfert, and Oleksandr Zinenko. High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, pages 119–134, New York, NY, USA, 2023. Association for Computing Machinery.

[7] NVIDIA Corporation. CUDA Tile. https://developer.nvidia.com/cuda/tile, 2025. NVIDIA Technical Blog. Accessed: 2026-17-06.

[8] NVIDIA Corporation. CUDA Tile IR Specification. https://docs.nvidia.com/cuda/tile-ir/, 2025. Part of CUDA Toolkit 13.1. Accessed: 2026-17-06.

[9] NVIDIA Corporation. cuTile Python. https://docs.nvidia.com/cuda/cutile-python/, 2025. Version 1.0.0, Released with CUDA 13.1. Accessed: 2026-17-06.

[10] OpenAI. Triton CPU. https://github.com/triton-lang/triton-cpu, 2024. Accessed: 2026-17-06.

[11] John A. Stratton, Sam S. Stone, and Wen-mei W. Hwu. Mcuda: An efficient implementation of cuda kernels for multi-core cpus. In José Nelson Amaral, editor, *Languages and Compilers for Parallel Computing*, pages 16–30, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[12] Terapines Technology Co,.Ltd . AI Benchmarks. https://github.com/Terapines/AI-Benchmark/, 2025. Accessed: 2026-17-06.

[13] 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, pages 10–19, New York, NY, USA, 2019. Association for Computing Machinery.

[14] 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. Structured operations: Modular design of code generators for tensor compilers. In Charith Mendis and Lawrence Rauchwerger, editors, *Languages and Compilers for Parallel Computing*, pages 141–156, Cham, 2023. Springer Nature Switzerland.

A Reproducability

To set up the benchmarking environment first clone the cuda-tile-cpu compiler from the following repository: https://github.com/moffel1020/cuda-tile-cpu/tree/cpu-conversion/. The build instructions are provided in the README. It is important that LLVM is built with assertions turned off. It also requires a specific LLVM commit, which can be found inside the file cmake/IncludeLLVM.cmake, which is `13c00cbc2aa2ddc9aae2e72b02bc6cb2a482e0e7` at the time of writing.

Secondly, we need to set up the triton-cpu compiler. Clone the repository from https://github.com/triton-lang/triton-cpu/. For the benchmarks, we used commit `270e696dbff5842b2c8dc6674b6d9382d6f96587`. Follow the instructions in the README to built the compiler (specifically, we need to access the `triton-opt` tool).

Finally, we can clone the benchmarks from https://github.com/moffel1020/tileir-cpu-benchmarks/. Follow the instructions in the README to run these benchmarks. For convenience, the IR for both Triton and CUDA tile has been exported beforehand. If the Python versions of the kernels get modified use the Python scripts `export_cutile_kernels.py` and `export_triton_kernels.py` to re-export the IR from the Python functions.

B Supported Operations

This table contains a full list of Tile IR 13.2 operations that our CPU compiler currently supports. ✓means full support, ∼ means partial support and × means no support.

Operation	Status	Notes
Core		
cuda_tile.broadcast	✓	
cuda_tile.cat	✓	
cuda_tile.constant	✓	
cuda_tile.entry	✓	
cuda_tile.extract	✓	
cuda_tile.get_global	×	
cuda_tile.get_num_tile_blocks	✓	
cuda_tile.get_tile_block_id	✓	
cuda_tile.global	×	
cuda_tile.iota	✓	
cuda_tile.module	✓	
cuda_tile.offset	✓	

Operation	Status	Notes
cuda_tile.permute	✓	
cuda_tile.reduce	✓	
cuda_tile.reshape	✓	
cuda_tile.scan	×	
cuda_tile.select	✓	
Conversion		
cuda_tile.bitcast	✓	
cuda_tile.exti	✓	
cuda_tile.ftof	✓	
cuda_tile.ftoi	✓	
cuda_tile.itof	✓	
cuda_tile.int_to_ptr	✓	
cuda_tile.ptr_to_int	✓	
cuda_tile.ptr_to_ptr	✓	
cuda_tile.trunci	✓	
Control flow		
cuda_tile.assert	×	
cuda_tile.break	×	
cuda_tile.continue	∼	Only if not inside an if-branch
cuda_tile.for	∼	Only with simple continue terminators
cuda_tile.if	✓	
cuda_tile.loop	×	
cuda_tile.return	✓	
cuda_tile.yield	✓	
Memory		
cuda_tile.join_tokens	×	
cuda_tile.load_ptr_tko	✓	
cuda_tile.make_token	✓	
cuda_tile.store_ptr_tko	✓	
Floating point		
(Ignoring FTZ and floating-point rounding modes)		
cuda_tile.absf	✓	
cuda_tile.addf	✓	

Operation	Status	Notes
cuda_tile.atan2	✓	
cuda_tile.ceil	✓	
cuda_tile.cmpf	✓	
cuda_tile.cosh	✓	
cuda_tile.cos	✓	
cuda_tile.divf	✓	
cuda_tile.exp2	✓	
cuda_tile.exp	✓	
cuda_tile.floor	✓	
cuda_tile.fma	✓	
cuda_tile.log2	✓	
cuda_tile.log	✓	
cuda_tile.maxf	✓	
cuda_tile.minf	✓	
cuda_tile.mmaf	✓	
cuda_tile.mulf	✓	
cuda_tile.negf	✓	
cuda_tile.pow	✓	
cuda_tile.remf	✓	
cuda_tile.rsqrt	✓	
cuda_tile.sinh	✓	
cuda_tile.sin	✓	
cuda_tile.sqrt	✓	
cuda_tile.subf	✓	
cuda_tile.tanh	✓	
cuda_tile.tan	✓	
Integer		
cuda_tile.absi	✓	
cuda_tile.addi	✓	
cuda_tile.cmpi	✓	
cuda_tile.divi	✓	
cuda_tile.maxi	✓	
cuda_tile.mini	✓	

Operation	Status	Notes
cuda_tile.mmai	✓	
cuda_tile.muli	✓	
cuda_tile.mulhii	✓	
cuda_tile.negi	✓	
cuda_tile.remi	✓	
cuda_tile.shli	✓	
cuda_tile.shri	✓	
cuda_tile.subi	✓	
Bitwise		
cuda_tile.andi	✓	
cuda_tile.xori	✓	
cuda_tile.ori	✓	
Atomics		
cuda_tile.atomic_cas_tko	×	
cuda_tile.atomic_rmw_tko	×	
Views		
cuda_tile.get_index_space_shape	✓	
cuda_tile.get_tensor_shape	✓	
cuda_tile.load_view_tko	∼	Missing dim_map support
cuda_tile.make_partition_view	✓	
cuda_tile.make_tensor_view	✓	
cuda_tile.store_view_tko	∼	Missing dim_map support
Miscellaneous		
cuda_tile.assume	✓	
cuda_tile.print_tko	✓	

C Compilation pipelines

C.1 Cuda Tile CPU

```
cuda-tile-opt \
    --canonicalize --cse \
    --convert-cuda-tile-to-standard \
    --canonicalize --cse \
    --convert-elementwise-to-linalg \
    --linalg-fold-into-elementwise \
    --canonicalize --cse \
    --cuda-tile-cpu-tile-and-fuse-into-store="tile-sizes=32" \
    --canonicalize --cse \
    --eliminate-empty-tensors \
    --cuda-tile-cpu-vectorize-linalg \
    --canonicalize --cse \
    --lower-cuda-tile-cpu-mem-ops \
    --canonicalize --cse \
    --cuda-tile-cpu-insert-parallel-loops \
    --loop-invariant-code-motion \
    --loop-invariant-subset-hoisting \
    --canonicalize --cse \
    --eliminate-empty-tensors \
    --one-shot-bufferize="bufferize-function-boundaries
    ↪  function-boundary-type-conversion=infer-layout-map" \
    --canonicalize --cse \
    --buffer-hoisting \
    --buffer-loop-hoisting \
    --promote-buffers-to-stack="max-alloc-size-in-bytes=128" \
    --canonicalize --cse \
    --buffer-deallocation-pipeline \
    --mem2reg \
    --canonicalize --cse \
    --lower-vector-multi-reduction="lowering-strategy=inner-reduction" \
    --convert-vector-to-scf \
    --canonicalize --cse \
    --lower-affine \
    --convert-scf-to-openmp \
    --convert-openmp-to-llvm \
    --convert-scf-to-cf \
    --expand-strided-metadata \
    --finalize-memref-to-llvm \
    --convert-vector-to-llvm="reassociate-fp-reductions=true enable-x86=true
    ↪  vector-contract-lowering=outerproduct" \
    --convert-cuda-tile-cpu-to-llvm \
    --convert-to-llvm \
    --reconcile-unrealized-casts \
    --canonicalize --cse \
```

C.2 Triton CPU

```
triton-opt \
        --pass-pipeline='builtin.module(
      triton-cpu-scalarize{skip-gather-scatter=true},
      triton-cpu-convert-memory-ops{use-gather-scatter=true},
      triton-cpu-convert-ptr-ops,
      triton-cpu-convert-elementwise-ops,
      triton-cpu-convert-elem-manip-ops,
      triton-cpu-convert-dot-op,
      triton-cpu-convert-histogram-op,
      triton-cpu-convert-reduction{use-multidim-reduction-op=true use-reduction-op=false},
      triton-cpu-convert-scan,
      triton-cpu-convert-control-flow-op,
      triton-cpu-convert-atomic-ops,
      triton-cpu-convert-debug-ops,
      cse, symbol-dce, canonicalize,

      triton-cpu-canonicalize,
      triton-cpu-optimize-masks,
      canonicalize,
      triton-cpu-convert-dot-generic,
      triton-cpu-add-casts-for-unsupported-ops{promote-bf16-to-fp32=true
      ↪  convert-mixed-precision-matmul=true promote-lib-math-to-fp32=true},
      triton-cpu-decompose-fp-conversions{decompose-bf16-conversions=true
      ↪  decompose-fp8-conversions=true},
      cse, symbol-dce, canonicalize,

      tt.func(triton-cpu-lower-multi-reduction),
      expand-strided-metadata,
      convert-vector-to-scf{full-unroll=true target-rank=1 lower-tensors=false},
      lower-affine,
      convert-scf-to-cf,
      convert-index-to-llvm,
      triton-cpu-func-op-to-llvm,
      triton-cpu-get-program-id-op-to-llvm,
      triton-cpu-memory-op-to-llvm,
      triton-cpu-atomic-ops-to-llvm,
      triton-cpu-debug-ops-to-llvm,
      convert-math-to-llvm,
      convert-math-to-libm,
      convert-vector-to-llvm{reassociate-fp-reductions=true enable-x86=true
      ↪  vector-contract-lowering=outerproduct},
      finalize-memref-to-llvm,
      reconcile-unrealized-casts,
      convert-arith-to-llvm,
      convert-func-to-llvm,
      convert-ub-to-llvm,
      canonicalize,
      cse,
      symbol-dce,
      ensure-debug-info-scope-on-llvm-func
    )'

```