From Pragmas to Tensix: Compiling OpenMP and OpenACC HPC codes for Tenstorrent accelerators

Anonymous Author(s)

Abstract

Tenstorrent accelerators are built from a mesh of Tensix cores and have shown strong performance and energy efficiency for Machine Learning (ML) and High Performance Computing (HPC). However, their software stack is oriented towards ML, and general HPC codes must be rewritten by hand against the bare-metal Metalium interface, with the programmer responsible for key decisions such as tiling, data movement, synchronisation and placement for an unconventional spatial dataflow architecture. This is a major barrier to HPC adoption.

Providing seamless offload of existing Fortran or C loop codes annotated with OpenMP or OpenACC, we present a compilation pipeline that lifts loops to a tensor representation, and lowers through a small number of abstraction levels each determining one class of decision. The tensor level describes what is computed, the shard level which cores hold what data, the data movement level what moves within and between cores, and the hardware level how this is realised. Evaluated on Wormhole and Blackhole, unmodified loops with a one or two line pragma achieve performance within a small factor of vendor-written and hand-written kernels at several times fewer lines of code. Across the Berkeley Dwarfs the same code runs on Blackhole, Zen5 CPU and A100 GPU, with fit to the architecture determining where the Blackhole is beneficial. For the most matrix engine native workload, performance matches cuBLAS on an A100 from unmodified code, whereas workloads dominated by element wise operations or strided data movement are better served by the CPU or GPU.

Keywords: MLIR, Tenstorrent, HPC, compilers, spatial dataflow

1 Introduction

A growing number of AI accelerators have moved away from the cache hierarchy of CPUs and GPUs and are instead adopting a spatial dataflow design. In this approach, the chip comprises a grid of cores each with their own scratchpad memory and pipelined compute engine. A Network on Chip (NoC) connects the cores and it is over this which all communication and synchronisation must be undertaken explicitly. Tenstorrent are one example of this trend, and their latest generation, the Blackhole, provides a theoretical peak performance of 664 BlockFP8 TeraFLOPS with an open-source, ML focussed software stack that ranges from low level bare-metal programming all the way to integration with LLM frameworks such as vLLM. However, it is the programming model that such an architecture exposes, rather than the peak performance, that determines whether a piece of hardware can be used effectively beyond machine learning.

A key opportunity is whether this class of accelerator, of which Tenstorrent is one example, can also be leveraged by High Performance Computing (HPC) developers, many of whom are domain, rather than computer, scientists. Indeed, recent work has demonstrated the potential for Tenstorrent hardware to accelerate key HPC kernels [6, 12, 13, 36, 44], however in each of these cases the programmer must extensively rewrite their code to target the Tenstorrent architecture which is not only time consuming but also requires significant expertise around the architecture itself.

Put simply, the requirement to rewrite algorithms by hand is the major blocker to HPC adoption of spatial dataflow accelerators, of which Tenstorrent is one example. The root cause is that these architectures require the programmer to address decisions that a cache hierarchy makes implicit such as which core holds which data, when data moves, and how it is synchronised. In this paper we propose an alternative solution where these decisions separate cleanly into a small number of classes that a compiler can fix one level at a time provided it starts from a representation that preserves what the loop computes rather than how it was written. OpenMP and OpenACC pragmas, ubiquitous in HPC, supply the parallelism and dependence guarantees needed to lift a loop to that representation. Presenting a multi-level compilation pipeline, the research question this paper addresses is whether lowering through a small number of abstraction levels, each addressing one class of decision, is sufficient to target a spatial dataflow accelerator automatically and effectively from unmodified pragma annotated code. This paper makes the following novel contributions.

- We show that four abstraction levels, each fixing one class of decision suffice to compile loop based HPC codes to a spatial dataflow accelerator. This pipeline is independent of source language, offload model and hardware generation where the same IR is produced from Fortran and C, with OpenMP or OpenACC, and lowered to both Wormhole and Blackhole.
- Our *ttstream* dialect expresses communication as Kahn-process streams over virtual channels, providing deterministic dataflow streaming so the values a program computes are independent of timing and of how a stream is realised.

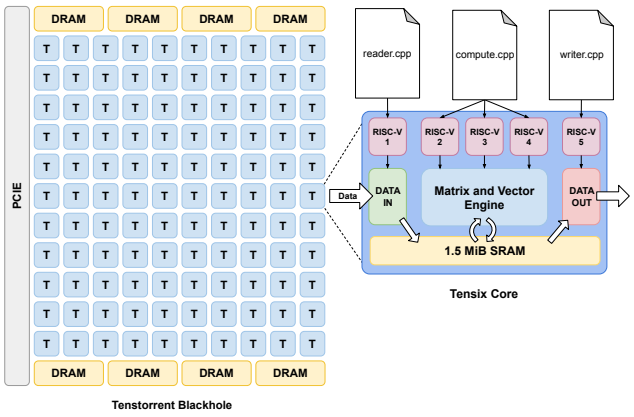

Figure 1. Tenstorrent Blackhole at a glance and the internals of a Tensix core. Application RISC-V CPUs are omitted.

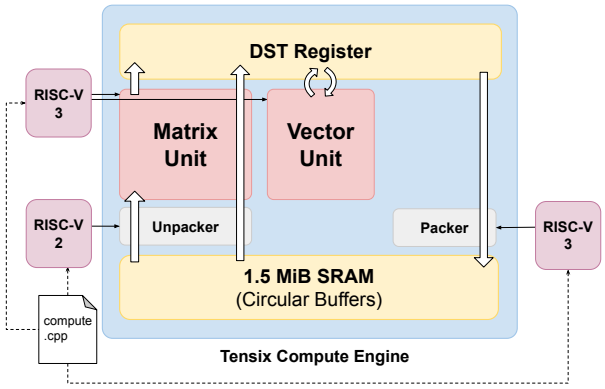

Figure 2. The Metalium programmer's perspective of the Tensix Compute Engine internals

- Low level architecture specific decisions, such as tiling, are made automatically by our approach where unmodified loops with a one or two line pragma achieve performance within a small factor of vendor-written TT-NN and hand-written TT-Lang kernels at several times fewer lines of code.

2 Background

Tenstorrent Blackhole, and previous generation Wormhole, accelerators are built around the Tensix architecture [48]. The chip comprises a grid of Tensix cores, interconnected by two Network-on-Chips (NoCs) with a wraparound producing a full 2D torus where the cost of sending data from one core to another is linear to the number of hops. Figure 1 illustrates the p150 Blackhole, containing 120 Tensix cores and 8 banks of 4 GiB of GDDR6 DRAM for a total of 32 GiB.

The right of Figure 1 zooms into a single Tensix core which is comprised of five "baby" RISC-V cores. These are responsible for control, the first bringing data into the Tensix-local SRAM memory (1.5 MiB per core on the Blackhole), three issue bespoke instructions to the co-processor, and the fifth writes data onto the NoC which can be routed to another core's SRAM, on-device DRAM, or to another p150 card. These baby RISC-Vs run in parallel, creating a hardware pipeline within each Tensix core where data can be brought in, calculations performed, and results sent out concurrently.

Each Tensix core also contains a co-processor compute engine which comprises a matrix unit (FPU), vector unit (SFPU), scalar unit (THCon) and unpacker/packer functionality to move data between SRAM and registers. The internals of this are illustrated in Figure 2, and the FPU can perform 4096 adds and multiplies per cycle on low precision TF32 data [46]. The trade-off is that the FPU can only perform a small number of operations such as matrix multiplication, element wise add, subtract, multiply, and transpose. By contrast the SFPU supports a wider variety of operations in higher precision, up to FP32, but operates on only 32 elements at a time. The unpacker moves data from SRAM into unit-local registers and undertakes any data transformation, such as converting FP32 to TF32 for the FPU. Conversely, the packer copies results back into SRAM from the *dst* register, where both the FPU and SFPU write results into that shared register. Whilst the SFPU can read from this register, the FPU cannot and instead consumes dedicated *srcA* and *srcB* registers which are double buffered for concurrent unpacking and compute.

2.1 The Tenstorrent programming model

Tenstorrent have committed to an open software stack, which is built upon their Metalium, bare-metal, programming model and Low Level Kernel (LLK) library. In this approach, programmers develop three kernels per Tensix core, each running on one of the three groups of baby RISC-V cores. These comprise a reader kernel that reads external data into local SRAM, a compute kernel that drives execution on the co-processor, and a writer kernel. Each group of cores communicates via circular buffers, which are FIFO queues in scratchpad SRAM that provides not only memory but also synchronisation. A core writes into a circular buffer and another consumes from it, with circular buffers being of configurable depth to enable pipelining of data.

To achieve best performance, one uses the RISC-V baby cores to only issue Metalium API calls, where these cores then issue instructions to the NoCs and co-processor, rather than execute any data manipulation or compute on them directly [12]. Compute APIs not only drive the FPU and SFPU, but also copy data between SRAM and co-processor registers, as well as manage the *dst* register. This *dst* register contains slices, each of which stores one tile. The number of slices that can be stored depends on the hardware configuration and element-type, for example four slices of FP32

data can be stored in a ping-pong configuration between the packer and unpacker units. The programmer must explicitly manage locks on this *dst* register, manually enabling consumption between different units in the co-processor, which adds significantly to the burden placed upon them [46].

There are four levels of synchronization which the programmer must explicitly control; kernel dispatch, inter-Tensix semaphores, intra-Tensix circular buffers, and intra-compute engine ownership of the *dst* register. Coupled alongside needing to break data into tiles, manage explicit control over each Tensix's SRAM, and handle a dual Network on Chip system with direct memory addressing and alignment constraints, this provides a bespoke, complex, programming model that is not only hard to design algorithms for, but very hard to exploit in a performant manner.

2.2 MLIR and xDSL

MLIR [29] is an extensible compiler framework for defining Intermediate Representations (IRs) as reusable dialects at varying levels of abstraction and progressively lowered via rewriting transformations. xDSL [19] is a Python compiler toolkit that is one-to-one compatible with MLIR, in which dialects are defined in IRDL [18] and IR passes freely between the two. This enables fast prototyping, an example being the MPI dialect which was first developed in xDSL [8] and then later upstreamed to MLIR. We use these MLIR and xDSL terms interchangeably.

2.3 HPC programming models

Loop based programs are ubiquitous in HPC [26], with the OpenMP and OpenACC programming models being well-established approaches for parallelising and/or offloading these. Furthermore, over 60% of workloads running on the UK national supercomputer are written in Fortran [40]. Both are driven by annotating loops and data with pragmas, OpenMP was first proposed for threaded programming and the addition of *target offload* extended it to support offloading loops to accelerators, typically GPUs. OpenACC was developed with the intention of offloading loops to accelerators, typically GPUs, and follows a similar approach to OpenMP in decorating loops but is more descriptive [28].

Listing 1 sketches the dot product algorithm, annotated with OpenMP target offload, in Fortran where data movement between the host and accelerator is handled implicitly. Both OpenMP and OpenACC provide explicit data movement constructs for optimisation.

3 Lowering through abstractions to the Tensix architecture

Our compilation pipeline lowers programs through four levels of abstraction, and in this section we describe each in turn from the top down, using the dot product of Listing 1 as a running example. Figure 3 sketches the stages of the

Listing 1 Sketch of dot product in Fortran with OpenMP target offload

```
1  !$omp target parallel do reduction(+: result)
2  do i = 1, 1000
3    result=result + v1(i) * v2(i)
4  end do
5  !$omp end target parallel do
```

pipeline, and in our approach the *tensor* level fixes what is computed, the *shard* level which cores hold which data, the *data movement* level both what moves within a core as tiles and between cores as streams, and the *hardware* level how this is actually implemented. Of the main MLIR dialects involved, *tensor*, *tosa* and *shard* are existing MLIR dialects and *device* is from [42], whereas *ttce*, *ttstream*, *cb* and *ttmtlm* are contributed by this work.

3.1 Lifting to the tensor level

As per Figure 3, our compilation pipeline accepts user code written in Fortran or C decorated with either OpenMP or OpenACC. [14] lifted Fortran and OpenMP loops to *tensors* and the *tosa* dialect for AMD's AI Engines, demonstrating that raising to value semantics and rank polymorphism [33, 39] gives the compiler the information it needs to make architectural decisions about how loops map to the hardware. That lifting is significantly generalised here to support a range of languages and offload models, and to go beyond simple kernels to a much wider range of applications. The lifting to tensors pass accepts IR that contains core MLIR dialects, originally decorated with the *openmp* dialect in [14], and generalised to also support OpenACC here. [11] developed a lowering from Flang's *fir* and *hlfir* dialects to core MLIR dialects, and to this end an analogous lowering was developed from Clang's experimental MLIR path, which emits the *cir* dialect, to core MLIR dialects, matching the behavioural mapping of [11] but for C rather than Fortran.

Table 1 sketches the overall mapping from loop construct to tensor representation provided by our generalised lifting transformation. Whilst this covers the majority of loop based codes, we explicitly consider three cases as out of scope and these are rejected at compile time. The first is data dependent control flow within the loop body because a conditional assignment has no counterpart amongst the whole-array operations that we emit. Element wise selection that arises from *max* and *min* operations is supported, as listed in Table 1, but general conditionals are not. These are not so common in compute-intensive HPC kernels, but could be addressed in the future by mapping to the `tosa.select` operation. The second limitation is that we do not handle indirect memory accesses, such as *a(idx(i))*, since the ability to resolve a subscript to a loop dimension and a constant displacement is assumed, with an indirection permitting neither. The third

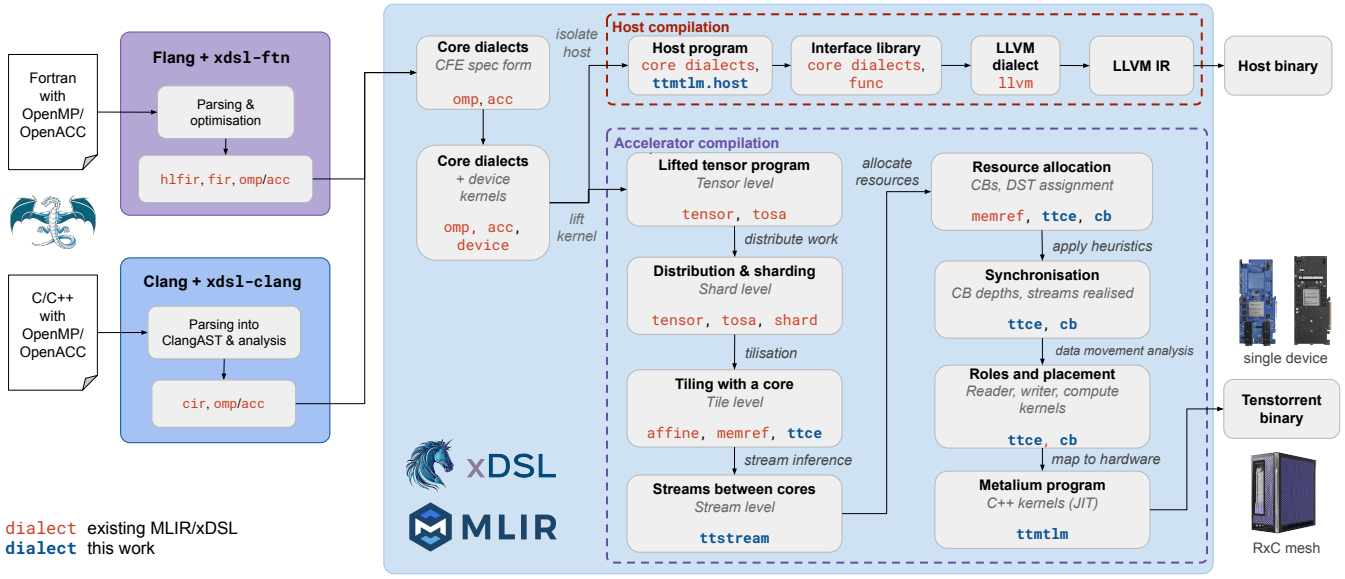

Figure 3. Overview of key stages of our compiler pipeline, from programmer written code in Fortran or C with either OpenMP target offload or OpenACC, to a binary running on Tenstorrent accelerators.

limitation is that our transformation only supports operations for which we have provided rules that map to tensors, for instance we currently support *exp*, *max* and *min* mathematical operations rather than all the operations in the *math* dialect. However, extending the list of rules is trivial and requires no change to the underlying mechanism.

Listing 2 Dotproduct kernel body transformed into device, tensor and tosa dialects operating over 1024000 elements of data.

```
%lb = arith.constant 1 : i32
%s = arith.constant 1 : i32
%ub = arith.constant 1024000 : i32

%res = device.tensor_compute(%a, %b, %lb, %ub, %s) {
^0(%at: tensor<1024000xf32>, %bt: tensor<1024000xf32>):
    %t0 = tosa.mul %at, %bt : tensor<1024000xf32>
    %r = tosa.reduce_sum %t0 : tensor<1xf32>
    device.tensor_yield %r : tensor<1xf32>
}
%val = tensor.extract %res[0] : f32
memref.store %val, %c[] : memref<f32>
```

Listing 2 illustrates the resulting IR after lifting the dot product kernel of Listing 1, *lifted tensor program* in Figure 3, to the *device*, *tensor* and *tosa* dialects. This same IR is produced by the equivalent C and/or OpenACC code, and beyond this point the compiler has no knowledge of which language or programming model a program was written in. There are three device-mapped *memref*s (%a, %b, and %c), whose creation is omitted for brevity and are resident within device DRAM memory. For a problem size of 1024000 elements, the `device.tensor_compute` operation accepts the %a and %b device-mapped memory as arguments and transforms these into tensors within the inner body region. Within this region the computation is described using value semantics, with the yielded out via `device.tensor_yield` and then stored into device-mapped memory %c.

3.2 Distribution and sharding

This is the shard level, which determines how many cores are involved and which data each holds, but not what moves between them. Expressing the parallelism in a Single Program Multiple Data (SPMD) style, the number of Tensix cores involved in the computation is first determined, through inference on the IR or a command line argument, and a device grid using the `shard.grid` operation is defined. Uniform sharding across this grid is adopted via `shard.sharding`, and each `device.tensor_compute` block argument in the IR is annotated with this sharding scheme via `shard.shard`. We then leverage the existing MLIR `sharding-propagation` pass to copy this annotation throughout applicable *tensor* and *tosa* operations so that they all operate on sharded tensors. MLIR's `shard-partition` pass then shards the tensors, and cleans up the IR removing the annotation operations. We have enhanced this last pass to also shrink the `device.tensor_compute` bounds once sharding is performed. The IR resulting from this stage contains individual tensors which are N times smaller, where N is the number of Tensix cores. This stage also adds the number of cores as an annotation onto the `device.kernel_create` operation for later passes, but does not specify which specific cores to run over.

Loop construct	Tensor representation
Arithmetic	
a(i) ± *b(i)*, *a(i)* × *b(i)*	`tosa.add`, `tosa.sub`, `tosa.mul`
a(i) / *b(i)*	`tosa.reciprocal` then `tosa.mul`
max(x,y), *min(x,y)*	`tosa.clamp`
exp(x)	`math.exp` over the tensor
*x**2*	`tosa.mul` of operand with itself
Literal or loop-invariant scalar	`tensor.splat` to iteration shape
Data access	
a(i)	the tensor itself
a(i±1)	`tensor.extract_slice`, offset ±1
a(j,i) under *do i, do j*	`linalg.transpose`
Rank-1 array with rank-3 field	`expand_shape` then broadcast
Thread-private scalar temporary	inlined at each use
Structure	
reduction(+:v), *reduction(max:v)*	`tosa.reduce_sum`, `tosa.reduce_max`
inner contraction *c(i,j)+=a(i,k)*b(k,j)*	`expand_shape`, `tosa.mul`, `reduce_sum` over *k*, `collapse_shape`
Inner loop invariant in its own index	multiply by the trip count
Several assignments in one body	one graph, single `tensor_compute`
Successive loops in a target region	result of one forwarded to the next

Table 1. The rules by which loop constructs are lifted into the *tensor* and *tosa* dialects.

Listing 3 sketches two snapshots of the IR, the first after the MLIR `sharding-propagation` pass which inserts operations from the *shard* dialect, and the second after the `shard-partition` pass which SPMDises the IR over 100 cores. In this second snippet it can be observed that the reduction has become a per-core partial sum, but nothing yet describes how the local reduction over 100 Tensix cores will be combined, or indeed which physical cores are involved.

For a two-dimensional operation, such as matrix multiplication, the grid itself is two-dimensional and each operand is split along one grid axis only, so its slice is identical for every core along the other axis. This replication is implied by the sharding rather than chosen separately, and later levels exploit it as a broadcast when deriving communication (Section 3.3.2) and as an operand that can be held resident in SRAM when allocating resources (Section 3.4.1).

3.3 Determining data movement

The data movement level fixes what moves within a core, between SRAM and the *dst* register, and between cores as streams. It does not determine however how the hardware actually undertakes either of these.

Listing 3 Dotproduct kernel body at the shard level, before and after SPMDisation, each core now holds 10240 elements and computes its own partial sum.

```
// Before SPMDisation: the sharding is on whole tensors
shard.grid @tt_device(shape = 100)
%res = device.tensor_compute(%a, %b, ...) {
^0(%at: tensor<1024000xf32>,
     %bt: tensor<1024000xf32>):
  %s  = shard.sharding @tt_device split_axes = ...
  %as = shard.shard %at to %s : tensor<1024000xf32>
  %bs = shard.shard %bt to %s : tensor<1024000xf32>
  %t0 = tosa.mul %as, %bs : tensor<1024000xf32>
  %ts = shard.shard %t0 to %s : tensor<1024000xf32>
  %r  = tosa.reduce_sum %ts {axis = 0} : tensor<1xf32>
  device.tensor_yield %r : tensor<1xf32>
}

// After SPMDisation: slice per core and partial sums
%res = device.tensor_compute(%a, %b, ...) {
^0(%at: tensor<10240xf32>, %bt: tensor<10240xf32>):
  %t0 = tosa.mul %at, %bt : tensor<10240xf32>
  %r  = tosa.reduce_sum %t0 {axis = 0} : tensor<1xf32>
  device.tensor_yield %r : tensor<1xf32>
}
```

3.3.1 Tiling within a core.

As described in Section 2.1, the Tenstorrent Tensix architecture operates across tiles and so compute must operate on a tile-by-tile basis. Because the Metalium API provides different calls depending upon which unit of the compute engine is used and the container in which its data resides, this adds significantly to the burden placed upon the programmer when writing codes manually. To limit the complexity of handling this in the compiler we provide the Tenstorrent Compute Engine (*ttce*) dialect which is a stepping stone that abstracts these low level details. It does this by focussing on where a tile of data currently resides and what mathematical operation to perform on it, rather than on the mechanics of how this is actually realised. The `ttce.tile` type describes not only the shape and element type of a tile of data, but also where it currently resides for example in an SRAM-backed circular buffer or within a slice of the *dst* register. This location information is later used to determine the specific Metalium calls to be generated as described in Section 3.4.2. The *ttce* dialect contains operations that undertake mathematics over tiles, tile creation and movement, kernel and core-set structuring.

Moving to this level of abstraction is undertaken by splitting each per-core tensor operation into operations over tiles, with an outer *affine* loop that iterates over the tiles one by one. The resulting IR for the dot product example is shown in Listing 4, with types simplified for readability, and follows a distribution of 1000 tiles over 100 Tensix cores with each core operating over 10 tiles.

The `ttce.read_tile` and `ttce.write_tile` operations, shown in Listing 4, load tiles from, and store tiles to, tensors respectively. Loop bounds are calculated based on the

Listing 4 Dotproduct kernel body after tilisation. Each core's loop over its ten tiles is now explicit, but circular buffers and *dst* register slices are not yet assigned at this point.

```
// all within a ttce.program
affine.for 0 -> 10 {chain_head} {
  %t0 = ttce.read_tile %a : tile<1024xf32, CB>
  %t1 = ttce.read_tile %b : tile<1024xf32, CB>
  %t2 = ttce.mul_tiles %t0, %t1 matrix -> DST
  %t2_cb = ttce.duplicate_tile %t2 : tile<..., CB>
  ttce.write_tile %t2_cb -> %im
}
%e = ttce.empty() : tile<..., DST>
%1 = ttce.declare_const_tile 1.0 : tile<..., CB>
%res = affine.for 0 -> 10 (%acc = %e) {chain_tail} {
  %t = ttce.read_tile %im : tile<1024xf32, CB>
  %r = ttce.reduce %t, %1 { sum, acc, matrix } : tile<..., DST>
  affine.yield %r
}
%res_cb = ttce.duplicate_tile %res : tile<..., CB>
ttce.write_tile %res_cb -> %c
```

number of elements in the sharded tensors, divided by the number of elements within a tile, which itself depends on specifics of the architecture and element type being used. The *tosa* operations in the IR have been replaced by their corresponding mathematical operations which act over tiles of data and are provided by our `ttce` dialect. The location of data, for instance whether it is in a circular buffer (*CB* attribute) or a slice of the *dst* register (*DST* attribute) is also annotated on some operations and types. Therefore, whilst there is some notion of data storage present at this stage in the IR, this is the *what* rather than the *how*, as details such as physical backing memory or depth are not yet specified.

In addition to the tile mapping, there are two other key activities undertaken at this point. The first is to ensure correct handling of reductions. A limitation imposed by the architecture is that results of tile reductions are stored in the *dst* register, but if these intermediates are mixed with other compute operations then the ownership of *dst* must be changed between unpacker and packers in the Tensix co-processor. This has the effect of zeroing out the entire register. Our flow handles this by splitting reductions out into their own dedicated loops and storing cross loop intermediates in SRAM as per those two loops in Listing 4. To further complicate things, the constant-tile declaration, `ttce.declare_const_tile`, is required as the Metalium *reduce_tile(a, b)* call operates over two circular buffers. Two arguments are required for averaging, and are a no-op for other reductions such as addition or multiplication, but must still be provided regardless. The reduction result is stored in the *dst* register and the end result packed back into SRAM after all partials have been accumulated. For very large datasets, where a Tensix core's SRAM does not provide enough space to hold all intermediate results from the first loop, then our pipeline automatically splits the reduction into chunks.

The second optimisation is to pattern match against groups of *tosa* operations and map these to primitives where appropriate. Matrix multiplication is an example, where a multiplication followed by a reduction over the contracted axis is recognised and replaced by a triply nested loop over tiles and a dedicated `ttce.matmul` operation. This will ultimately map to the *matmul* API call in Metalium which instructs the FPU to undertake a tiled matrix multiplication.

3.3.2 Streaming between cores.

Tenstorrent describe their Tensix architecture as being built upon a dataflow model [51], however arguably this is not exposed within their software stack, as communication between Tensix cores in Metalium is either via shared memory in DRAM, or by copying into another core's SRAM and using semaphores for synchronisation. It is however possible to build dataflow abstractions atop these foundations, and we take inspiration from AMD's IRON framework [25] which is used for programming their AI engines [21]. Our *ttstream* dialect provides a streaming abstraction between Tensix cores. It is possible to specify which cores, and groups of cores, are involved in both point-to-point and collective communications, the latter supporting reduction, broadcast, scatter and gather, without specifying how this is to be realised.

Streams are a beneficial level of abstraction at this level because their semantics means that values a program computes do not depend upon timing, nor upon how a stream is realised on the hardware. This enables the compiler to treat the choice of realisation, for instance whether via DRAM, via multicast across the NoC from one core to others, or by forwarding between neighbouring cores, as an optimisation decision which can be changed without any other part of the IR being affected. A stream itself is a First-In First-Out (FIFO) queue of typed elements between a set of producer and consumer cores, where writing will block execution until there is space for the data and reading blocks until data is present. Giving a stream a depth greater than one provides the overlapping of communication and compute via pipelining which is crucial for performance [6]. A major difference between our *ttstream* and AMD's IRON is that our data is streamed over virtual channels, that are later lowered to Metalium primitives such as NoC transfers and semaphore synchronisation, rather than over a fixed pool of hardware resources on AMD's AI Engines. Consequently, in our approach there is no limit on the number of streams in a program beyond the availability of circular buffers.

Communications are first inferred from the sharded grid, where the geometry and dependencies control the placement of stream operations. Two kinds of collective are determined at this point, broadcasts and reductions. The first is the fan-out pattern, where if several cores reads data from DRAM, and there is duplication here, then that DRAM bottleneck can be ameliorated via a broadcast. The second is a fan-in, where a loop with a reduction clause results in each core

holding a partial result. A cross core reduction is generated, where every core is a producer and the root the consumer and the channel is decorated with a combination operation.

Listing 5 sketches the definition of a streaming channel for the dot product's reduction over four Tensix cores. The *producer* and *consumer* attributes reference cores on the grid, *depth* is the capacity of the channel, *combine* is an optional reduction operator that combines data from multiple producers, and the optional *staging* operand specifies working memory to carry out the combination. The *realization* attribute is also optional and directs where this channel should be stored, in this case DRAM, and if omitted the compiler infers the most appropriate location based on the data type, size and number of producer and consumer cores.

Listing 5 Stream definition syntax.

```
%ch = ttstream.channel label("stream_a")
    producer [col<4x10: 0>]
    consumers [rank<4x10>: 0]
    depth(1)
    realization(dram)
    combine(sum)
    staging(%partial_storage : memref<32xf32, DRAM>)
    : !ttstream.stream<f32>
```

In this declarative manner it is the channel itself that defines behavioural properties including those ranks that are producers and consumers. Therefore not only is writing to, or reading from, a channel simplified at the data movement operation level, but furthermore it is possible to include those operations within an SPMD program because when this is transformed into MPMD across the grid then reads and writes which do not include that current specific rank are simply not generated. Communication operations in the *shard* dialect map directly onto their *ttstream* counterparts, so MPMD programs written at that level are also supported.

3.4 Lowering to the hardware

The hardware level is responsible for everything that the levels above abstracted. This includes the resources each core uses, where kernels run, and the API calls to perform underlying operations.

3.4.1 Resource allocation. This step determines which Tensix resources such as registers, circular buffers, and SRAM will be used to realise the tilised mathematical operations in the IR. Allocation is achieved by determining what resources are required from, and made available by, the target hardware, as well as how to best allocate them. Our approach supports multiple generations of the Tenstorrent architecture by carrying a description of the board's available resources in the IR via the *dlti* dialect throughout the pipeline. Descriptions exist for each Tenstorrent product, and this information is consumed when making hardware specific decisions such as determining the availability of resources.

Circular buffers play a crucial role within a Tensix core as they allow for data to be passed between the five baby RISC-V cores via SRAM in a safe manner. Presenting a FIFO queue with a consumer and producer, circular buffers combine data storage with synchronisation and enable data to be pipelined between different parts of the Tensix core. Our *cb* dialect contains three groups of operations that represent and manage circular buffers within the IR; the creation of circular buffers based upon a given element type and optional attributes such as size, FIFO queue operations that act on the data stored within the circular buffer, and blocking synchronisation operations that wait for data or a free slot to be available. This dialect in and of itself requires no sizing information, backing information, or synchronization to be inserted within the IR. An initial pass determines where circular buffers need to be inserted throughout the IR, and after that the pipeline progressively adds additional realisation information so that circular buffers are ultimately annotated and sized appropriately for lowering to Metalium.

Listing 6 sketches the IR after resource allocation, where each on-device `memref` is annotated with its source address space in device's DRAM or Tensix-local SRAM. Circular buffers are specified using the *cb* dialect and `ttce.read_tile` and `ttce.write_tile` now work with them directly rather than operating over abstract tiles until this point. This materialises tiles to their concrete realisation via circular buffers, providing both the backing memory and synchronisation. Implicit circular buffers, which typically arise from kernels where a tile must be written back to SRAM for the next operation to consume it via the FPU, are also discovered and inserted. An example of this is `%cb3` in Listing 6.

Once circular buffers have been inserted, each is then annotated with usage information, such as the `intent` (input, output, or intermediate), and its `distribution` (whether it is to be replicated with the same values across cores or instead sharded), as seen in Listing 6. Based on resources provided by the specific architecture, each circular buffer is sized to have an appropriate depth given the available SRAM. Where sharding implies that an operand is replicated along a grid axis, the *matmul* lowering keeps it resident in SRAM, marked `ttce.resident`, and prefetches panels of tiles.

3.4.2 Placement and mapping to Metalium. The next stage separates compute into separate reader, writer and compute kernels that run over the RISC-V baby cores of the Tensix tile, physically places these kernels across the Tensix cores, and lowers to the Metalium API level. As illustrated in Listing 7, the kernel is first split into separate `ttce.reader`, `ttce.writer` and `ttce.compute` kernels with circular buffers added for pipelining data between the RISC-V cores.

In Listing 7 kernels contain the *[*]* modifier, and this directs them to run on all Tensix cores. Instead, a list of specific cores or a range can be specified. In this manner, the IR can

	AMD EPYC 9825 (2024)	Nvidia A100 (2020)	Wormhole n300 (2024)	Blackhole p150 (2025)
Compute units	144 cores	6,912 CUDA cores	2 x 64 Tensix cores	120 Tensix cores ∗
Clock speed	2.2 GHz	0.77 GHz	1.0 GHz	1.35 GHz
Peak perf. (TFLOPS)	20 (FP32)	78 (FP16)	131 (BF16)	166 (BF16)
Peak mem. band. (GiB/s)	614	1555	576	512
Memory capacity	12-channel DDR5	40 GiB HBM2	2 x 12 GiB GDDR6	32 GiB GDDR6
Cache / on-chip SRAM	384 MiB L3	40 MiB L2	180 MiB SRAM	160 MiB SRAM
Launch price	$13,000 [45]	$10,000 [2]	$1400 [50]	$1400 [50]

Table 2. Comparison of hardware used for evaluation. ∗ for p150 only 110 cores are exposed by Metalium.

Listing 6 Dot product after allocating resources. The reduction loop is similarly updated and reads from the discovered intermediate buffer *cb3*

```
%cb0 = cb.create(%a) {distribution = sharded,
    intent = input, id = 0} : cb<10xtile<1024xf32>>
%cb1 = cb.create(%b) {...} : cb<10xtile<1024xf32>>
%cb3 = cb.create(%im) {intent = intermediate,
    id = 3} : cb<10xtile<1024xf32>>  // discovered

affine.for 0 -> 10 {chain_head} {
  ttce.read_tile %a -> %cb0
  ttce.read_tile %b -> %cb1
  %t0 = cb.peek %cb0 : tile<1024xf32, CB>
  %t1 = cb.peek %cb1 : tile<1024xf32, CB>
  // As in Listing 5
}
// Second loop as in Listing 5, reading from %cb3
```

Listing 7 Dotproduct after inferring data movement.

```
ttce.reader [*] ranked (%rrank) {
    affine.for 0 -> 100 {
        ttce.read_tile %dram0 -> %cb0
        ttce.read_tile %dram1 -> %cb1
}   }
ttce.compute [*] ranked (%crank) {
    %t0 = cb.peek %cb0, %t1 = cb.peek %cb1
    %t2 = ttce.mul_tiles %t0, %t1
    cb.push %t2 -> %cb2
    // ...
}
ttce.writer [*] ranked (%wrank) {
    // write out result (single-tile, no loop)
    ttce.write_tile %cb2 -> %dram2
}
```

represent both SPMD and MPMD where there can be multiple reader, compute and writer kernels that operate across different ranges of Tensix cores. Physical placement maps logical ranks to core coordinates by a fixed column first walk and the pipeline does not currently model NoC distance, with communication-aware placement as further work.

The final stage lowers the IR to primitives that call into the Metalium API via the `ttmtlm` dialect. There is a one-to-one mapping from MLIR operations to Metalium C++ API calls, and the purpose is for the IR to then be consumed by our C++ kernel printer which generates code for the device. Each group of operations is intended to be run in a different kernel and defines its own namespace, resulting in four groups of operations across the host, compute, data movement and circular buffer APIs. Code generation is required because the host binary must reference the C++ kernels by filename as these are JIT compiled on first execution.

4 Results and Evaluation

We compare our approach on Tenstorrent Blackhole and Wormhole accelerators against other approaches, and also a CPU and GPU. Table 2 summarises the hardware used for comparison in this section, where on Tenstorrent we use TT-Metalium v0.74.0, TT-Lang v1.1.6 and LLVM v22.1.2. On the CPU we use LLVM v22.1.2 and on the GPU Nvidia HPC compilers v24.5 with CUDA v12.4. Each measured kernel reports the device-only runtime averaged over 50 hot runs, and after the initial program has been JIT compiled on the Tenstorrent. All codes are built with optimisation level three, and BLAS rows of Table 4 use OpenBLAS v0.3.26 and cuBLAS v12.4 (no equivalent vendor library exists on Tenstorrent). Energy is device-only, measured via *tt-smi* on Tenstorrent, *RAPL* on the CPU and *NVML* on the GPU.

4.1 Foundational kernels

Table 3 reports performance and lines of code for foundational HPC and ML kernels, broadly in order of algorithmic complexity, on Tenstorrent Wormhole and Blackhole. We compare our approach to kernels written by Tenstorrent engineers in their TT-NN library and hand-written kernels in Tenstorrent's TT-Lang Domain Specific Language (DSL). Kernels are run over all available Tensix cores, 128 on the Wormhole and 110 on the Blackhole, apart from GEMM which runs over 64 cores throughout, and all data is BF16. It can be seen that our approach requires fewer lines of code than either TT-NN or TT-Lang, which is existing code with a one or two line addition of OpenMP or OpenACC. By contrast, in TT-NN and TT-Lang all lines are bespoke.

From Table 3 it can be seen that the Blackhole is, in the main, faster than the Wormhole and our approach outperforms TT-NN for unfused operations SAXPY, dot product,

		Our approach			TT-NN			TT-Lang	
Kernel	**Problem size**	Wormhole	Blackhole	LoC	Wormhole	Blackhole	LoC	Blackhole	LoC
expm1	1,845,493,760	52.22 ms	34.36 ms	7	18.5 ms	28.04 ms	60	21.44 ms	37
SAXPY	1,845,493,760	53.58 ms	30.36 ms	9	44.36 ms	46.49 ms	310	26.72 ms	43
Dot Product	1,845,493,760	53.55 ms	29.93 ms	10	58.33 ms	50.06 ms	372	20.2 ms	52
L2 Norm	1,845,493,760	27.37 ms	15.00 ms	11	58.37 ms	49.79 ms	122	28.92 ms	99
Softmax	57,671,680	3.43 ms	2.55 ms	24	963.60 ms	861.10 ms	160	3.67 ms	174
GEMM	1024×1024	0.57 ms	0.28 ms	16	0.14 ms	0.15 ms	950	0.42 ms	92

Table 3. Performance comparison of the three programming models, device-only performance in milliseconds.

and L2 norm on the Blackhole. It should be highlighted that Tenstorrent have heavily optimised their GEMM in TT-NN, and this results in higher performance but around 60 times more lines of code. TT-NN's softmax is an outlier because it has been designed for batched workloads and we observed that only one Tensix core is active for this single-array softmax. We conclude from this experiment that, for loops annotated with OpenMP or OpenACC, our compilation pipeline delivers performance within a small factor of vendor kernels written in TT-NN and those hand-written in TT-Lang. Our approach is faster than TT-NN on the unfused streaming kernels, within 1.6x of TT-Lang on the simplest, and within 2x of Tenstorrent's heavily optimised GEMM.

4.2 HPC Berkeley Dwarfs

We expand our set of benchmarks to include a wider range that are common in HPC, based on the most applicable computational patterns identified by the Berkeley Dwarfs [7] taxonomy. Comparing against CPUs and GPUs, these benchmarks are more complex than those in Section 4.1 and help test the generality of our approach. All runs on the p150 Blackhole use FP32 input and output data with BF16 on the FPU for calculation and FP32 for SFPU operations, on the CPU and GPU we use FP32 because this was the most performant configuration. Whilst both the CPU and GPU support BF16 in hardware, the Flang compiler does not generate BF16 instructions for the CPU and the Nvidia compiler produced faster binaries on the A100 GPU. The same code is compiled for Tenstorrent and Nvidia GPUs without any architecture specific optimisations at the code level for either. OpenMP is used for threading over all cores of the CPU. The last three rows in Table 4 instead leverage BLAS libraries, OpenBLAS and cuBLAS, on the CPU and GPU respectively to provide a comparison against optimised baselines.

4.2.1 Dense linear algebra. The *GRAM* matrix formation [24] and *Cholesky* benchmarks in Table 4 are decomposed by our pipeline directly into tiled GEMM-style operations. These matrix engine native applications map naturally onto the Tensix architecture, where the FPU performs the bulk of the computation and reuse of data in local SRAM is high. It is important to stress that Table 4 compares the same unmodified source across all three architectures where the GRAM code is a plain triple loop with a one-line pragma, and neither the CPU nor the GPU build has been tuned, blocked, or calls into a library. Under that condition, the Blackhole reaches 16.9 TFLOP/s against 0.95 TFLOP/s on the CPU and 0.29 TFLOP/s on the GPU, at an order of magnitude lower energy to solution. That difference is not that the Blackhole is intrinsically faster at GEMM than an A100 but that our pipeline recognises the contraction and lowers it to the matrix engine, and the CPU and GPU compilers do not perform equivalent restructuring. For GRAM, cuBLAS reaches 15.1 TFLOP/s on the A100 and the CPU OpenBLAS library 1.95 TFLOP/s, so the Blackhole from unmodified pragma code matches the tuned GPU library in throughput, albeit using BF16 on the FPU where cuBLAS is FP32, and at roughly twice the energy. Cholesky is slower than GRAM as it is latency bound where panel factorisation and triangular solves on the diagonal blocks are inherently sequential, limiting the number of Tensix cores running concurrently although at this size it still outperforms both OpenBLAS and cuBLAS which are also latency bound.

4.2.2 Spectral methods. Spectral methods, which represent solutions in a global basis such as Fourier modes, are underpinned by the discrete Fourier Transform (DFT), one of the most widely used kernels in HPC. Computing the DFT directly costs $O(N^2)$, and the Cooley–Tukey Fourier Transform (FFT) is the standard way of reducing this to O(N log N). However, its butterfly structure has low arithmetic intensity, strided memory accesses, and data patterns that do not tile naturally onto the Tensix architecture. Consequently *FFT CT* performs poorly on the Blackhole (Table 4), with SFPU compute and NoC bandwidth dominating rather than the matrix engine (FPU). We therefore also benchmarked computing the DFT directly as a matrix multiplication, *DFT GEMM* in Table 4, applying the $N \times N$ DFT matrix via the compiler's tiled GEMM. This performs more floating point operations but maps them onto the matrix engine and delivers significantly better performance on the Blackhole and hence is especially beneficial for small FFTs. The Blackhole still outperforms the version using OpenBLAS on the CPU, however cuBLAS on the A100 provides four times the Blackhole's performance.

Benchmark	Programmed in	Problem size	Blackhole		CPU		GPU	
			GFLOP/s	Energy (J)	GFLOP/s	Energy (J)	GFLOP/s	Energy (J)
GRAM	C+OpenMP	2048	16880	0.18	948	2.80	289	4.05
Cholesky	Fortran+OpenMP	512	46	0.1	1.67	8.32	0.44	16.76
DFT GEMM	Fortran+OpenACC	256	1090	0.02	268	0.15	187	0.32
FFT CT	Fortran+OpenACC	256	6	0.03	41	0.007	17	0.03
LJMD	C+OpenACC	2048	15	1.7	1058	0.35	2513	0.02
Nbody	C+OpenACC	81920	286	56.7	855	35.61	4075	0.98
Nekbone	Fortran+OpenMP	100000	660	37.5	686	98.80	928	28.6
GRAM	BLAS	2048	N/A	N/A	1954	0.49	15085	0.10
Cholesky	BLAS	512	N/A	N/A	18	0.45	12	0.50
DFT GEMM	BLAS	256	N/A	N/A	807	0.03	4263	0.01

Table 4. Benchmarks written in mixture of Fortran or C with OpenMP or OpenACC, compiled for Blackhole using our approach and compared to same code running threaded over all cores of AMD Zen5 CPU (in OpenMP) and Nvidia A100 GPU.

4.2.3 N-body. The *LJMD* and *Nbody* benchmarks both undertake pairwise force calculations, placing them in the N-body dwarf. *LJMD* is a molecular dynamics code that evolves a system of particles interacting through the Lennard-Jones potential [30], a widely used model for short-range van der Waals forces [3]. *Nbody* is a direct-summation gravitational simulation [1], where every particle interacts with every other particle via the inverse-square law. In both, the force evaluation is embarrassingly parallel and so distributes trivially across the Tensix cores. However the arithmetic is element wise rather than matrix-oriented, and does not map naturally onto the FPU. Furthermore, the reciprocal and reciprocal square root at the heart of these kernels, needed to form the $1/r^3$ term in *Nbody* and the r^{-6} and r^{-12} terms in *LJMD*, must be executed on the SFPU, whose throughput is considerably lower than that of the FPU.

4.2.4 Unstructured grid. *Nekbone* [5] is a mini-app derived from the Nek5000 computational fluid dynamics code [20], and we focus on its *AX* kernel, the application of the discrete Laplacian, which accounts for around 90% of the runtime. Nek5000 is a spectral element method [35], in which the domain is decomposed into many small hexahedral elements, each carrying a high-order polynomial basis. Under the Berkeley Dwarfs taxonomy this is classified as an unstructured grid, since elements are connected through an arbitrary mesh. Within each element, however, applying the Laplacian is a tensor product contraction that reduces to three small dense matrix multiplications. This mix of a matrix engine friendly inner kernel with irregular data movement between elements makes the suitability of *Nekbone* for the Tenstorrent architecture sit between that of dense linear algebra and N-body benchmarks. As in [6], we offload the per-element calculation only, expressed as loops over element-local arrays where data gather/scatter remains on the host. That work recorded 242 GFLOP/s [6] for the same configuration. Our approach achieves around three times the performance obtained by [6], and this is due to the newer Blackhole generation and the compiler's matmul tiling. It can be seen in Table 4 that this results in poorer performance on the Blackhole compared to the CPU and GPU, with associated higher energy to solution.

Stencil Kernel	Blackhole		CPU		GPU	
	GPt/s	Joules	GPt/s	Joules	GPt/s	Joules
Gauss-Seidel	8.50	0.03	21.06	0.05	26.17	0.05
PW adv	8.05	0.02	113.30	0.03	38.41	0.02
SWM	1.92	0.07	59.61	0.02	16.84	0.08

Table 5. Stencil benchmark performance and energy in Fortran+OpenMP on Blackhole (our compiler), CPU and GPU

4.2.5 Structured grid. Stencil based codes are a major example of the structured grid dwarf, where each point in a regular grid is updated from a fixed pattern of neighbouring values. Stencils are ubiquitous in HPC, forming the basis of PDE solvers used heavily in codes that undertake weather, climate and engineering simulations. Whilst memory access is regular and predictable, arithmetic intensity is low and there is no matrix multiplication to map onto the FPU. Table 5 reports throughput in billions of grid points per second (GPt/s) and energy to solution for a 2D Gauss-Seidel, the 3D PW advection scheme from the Met Office's MONC high resolution atmospheric model [15], and NCAR's 2D Shallow Water Model (SWM) mini-app [34]. For this class of problem performance is dominated by how effectively data can be streamed through the Tensix cores' SRAM and the speed of element wise operations executed on the FPU and SFPU.

5 Related Work

The first generation of HPC work on Tenstorrent hardware ported individual kernels by hand. Examples include stencils to the Grayskull [12] and Wormhole [36], FFTs [13], spectral element methods [6], N-body [4], matrix multiplication [37], and a Conjugate Gradient (CG) solver [44]. These established

that the architecture can deliver competitive performance and energy efficiency for HPC, but requires rewriting of code with explicit management of the four levels of synchronisation described in Section 2.1, and architecture specific techniques such as tile blocking. Our approach folds these techniques into the compiler, so that the decisions those authors discovered by hand can instead be made automatically.

Tenstorrent's own stack raises the level of abstraction in two ways. Firstly the TT-NN library provides hand-optimised operations, and TT-MLIR [47] together with the TT-Forge and TT-XLA frontends, compiles ML graphs from PyTorch, JAX and ONNX onto these operations. This is focused on machine learning and the stack consumes tensor graphs rather than general-purpose codes. Secondly, TT-Lang [49] is a Python DSL developed by Tenstorrent that sits above Metalium. However, the programmer must still restructure their algorithm into separate data movement and compute kernels, manage the synchronisation between them, and understand the architecture to obtain performance. This removes the C++ boilerplate code, but does not address algorithmic recasting. By contrast, our work sits above both these levels where we consume unmodified OpenMP and OpenACC loops, and TT-Lang or TT-NN operations could in principle serve as an alternative target for our compilation pipeline.

TileLoom [32] is an MLIR-based approach that plans tile distribution and inter-core communication across Tensix cores using a cost model, and reports speedups of 1–2× over TT-NN for GEMM and FlashAttention. It is closest to our work, however the input must already be tiled, with block sizes and strides given explicitly, and frontends are Triton and Helium which are tile DSLs. The burden of tiling is therefore moved into the DSL layer and on the programmer. As described in Section 3.4.2, placement in our approach is currently simplistic and instead we could leverage TileLoom's cost model in the future. HetGPU [53], T2T [31] and Booth [23] provide an approach that targets GPU programs written in CUDA, HIP or Triton to Metalium. HetGPU lifts LLVM IR into a shared GPU IR, T2T converts 2D-vectorising CUDA thread blocks into tiles, and Booth provides a bespoke translation tool. All three perform a translation, but undertake no algorithmic restructuring therefore resulting in a kernel that still adopts the GPU's bulk-synchronous, cache-centric approach rather than a spatial dataflow design. By contrast, initially lifting a program to value semantics provides our pipeline with rich information upon which subsequent transformations can make key decisions around distribution, streaming and residency.

Numerous works have explored recovering a programmer's high-level intent from code, from progressive raising within MLIR [16] to synthesis-based lifting such as mlirSynth [10], C2TACO [17], Tenspiler [38] and Tensorize [9]. The last of these lifts legacy C and Python loops to the `linalg` MLIR dialect and from this generates NumPy or StableHLO. These are more general than our lifting, but also more complex because loop independence and operator structure must be identified from unannotated code. We instead leverage the parallelism and dependency guarantees provided by OpenMP and OpenACC pragmas, which enables lifting to be based on the rule-driven mapping in Table 1. As these two pragma models are ubiquitous in HPC we believe this is a reasonable expectation. More importantly, those existing lifting frameworks only target CPUs and GPUs, whose compiler stacks are mature. The exception is [14], which lifted Fortran and OpenMP to `tosa` for AMD's AI Engines and which, as described in Section 3.1, is generalised in this work across languages, offload models, and a range of codes.

Beyond Tenstorrent, MLIR-based lowerings exist for a range of spatial and dataflow targets including IRON [25] for AMD's AI Engines, [43] for the Cerebras WSE, [41] for FPGAs and [52] for CGRAs. The Cerebras and FPGA work shares our approach of leveraging high-level information to drive architecture specific decisions, but unlike our approach is tied to the *stencil* dialect and thus can be used for only one, albeit popular, class of HPC algorithm. SPADA [22] and Spatial [27] are languages for spatial dataflow hardware that provide abstractions comparable to our stream and hardware levels, but require the programmer to leverage their API.

6 Conclusions

In this paper we have presented a compilation pipeline that enables existing loop based HPC codes, written in Fortran or C and annotated with OpenMP or OpenACC, to target Tenstorrent's spatial dataflow accelerators without the programmer rewriting their algorithm. We have demonstrated that lowering through a small number of abstraction levels, each addressing one class of decision, supports a range of codes and programming models where key decisions such as the tiling, resource, synchronisation and placement decisions a Metalium programmer must make by hand are automated.

Evaluated on the Wormhole and Blackhole, loops decorated with a one or two line pragma achieve performance within a small factor of vendor-written TT-NN and hand-written TT-Lang kernels at several times fewer lines of code. Across the Berkeley Dwarfs, compared against an AMD EPYC Zen 5 CPU and Nvidia A100 GPU running the same unmodified source, performance on the Blackhole depends heavily on architectural fit. Matrix engine native workloads, such as GRAM, run from unmodified code on the Blackhole at throughput matching a tuned cuBLAS implementation on the A100, because the compiler maps them onto the FPU.

For future work placement is currently based on a fixed column first approach that ignores NoC distance, and a cost model in the style of TileLoom [32] would enable placement, stream realisation and residency to be better determined. Frontends can also be broadened, for instance supporting C++ and its standard parallelism library, *stdpar* which already supports GPUs.

References

[1] Sverre J. Aarseth. 2003. *Gravitational N-Body Simulations: Tools and Algorithms*. Cambridge University Press.

[2] Advanced Clustering Technologies Inc. 2020. *Designing an HPC Cluster with Expert Help*. Advanced Clustering Technologies Inc. https://www.advancedclustering.com/wp-content/uploads/2021/01/PDF-Building-A-Cluster-Webinar.pdf

[3] Michael P. Allen and Dominic J. Tildesley. 2017. *Computer Simulation of Liquids* (2 ed.). Oxford University Press.

[4] Jenny Lynn Almerol, Elisabetta Boella, Mario Spera, and Daniele Gregori. 2025. Accelerating Gravitational N-Body Simulations Using the RISC-V-Based Tenstorrent Wormhole. In *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25)*. ACM, 1729–1735. doi:10.1145/3731599.3767528

[5] Argonne National Laboratory. 2013. Nekbone. https://github.com/Nek5000/Nekbone.

[6] Daniyal Arshad and Nick Brown. 2026. Exploring spectral element methods on the Tenstorrent RISC-V accelerator. arXiv:2608.22964 [cs.DC] https://arxiv.org/abs/2608.22964

[7] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical Report UCB/EECS-2006-183. EECS Department, University of California, Berkeley.

[8] George Bisbas, Anton Lydike, Emilien Bauer, Nick Brown, Mathieu Fehr, Lawrence Mitchell, Gabriel Rodriguez-Canal, Maurice Jamieson, Paul HJ Kelly, Michel Steuwer, et al. 2024. A shared compilation stack for distributed-memory parallelism in stencil DSLs. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 38–56.

[9] Alexander Brauckmann, Luc Jaulmes, José W de Souza Magalhães, Elizabeth Polgreen, and Michael FP O'Boyle. 2025. Tensorize: Fast Synthesis of Tensor Programs from Legacy Code using Symbolic Tracing, Sketching and Solving. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. 15–30.

[10] Alexander Brauckmann, Elizabeth Polgreen, Tobias Grosser, and Michael F. P. O'Boyle. 2023. mlirSynth: Automatic, Retargetable Program Raising in Multi-Level IR Using Program Synthesis. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 39–50. doi:10.1109/PACT58117.2023.00012

[11] Nick Brown. 2024. Fully integrating the Flang Fortran compiler with standard MLIR. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 939–949.

[12] Nick Brown and Ryan Barton. 2024. Accelerating stencils on the Tenstorrent Grayskull RISC-V accelerator. In *Proceedings of the International workshop on RISC-V for HPC*. IEEE Computer Society Press. International workshop on RISC-V for HPC (RISCV-HPC) at SC24, RISCV-HPC ; Conference date: 18-11-2024.

[13] Nick Brown, Jake Davies, and Felix Le Clair. 2026. Exploring Fast Fourier Transforms on the Tenstorrent Wormhole. In *High Performance Computing*, Sarah Neuwirth, Arnab Kumar Paul, Tobias Weinzierl, and Erin Claire Carson (Eds.). Springer Nature Switzerland, Cham, 598–612.

[14] Nick Brown and Gabriel Rodriguez-Canal. 2026. Lifting to Tensors when Compiling Scientific Computing Workloads for AI Engines. In *2026 IEEE 26th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 647–652. doi:10.1109/CCGrid68966.2026.00075

[15] Nick Brown, Michele Weiland, Adrian Hill, Ben Shipway, Chris Maynard, Thomas Allen, and Mike Rezny. 2015. A highly scalable Met Office NERC Cloud model. In *Proceedings of the 3rd International Conference on Exascale Applications and Software*. 132–137.

[16] Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Uday Bondhugula, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. 2021. Progressive Raising in Multi-level IR. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 15–26. doi:10.1109/CGO51591.2021.9370332

[17] José Wesley de Souza Magalhães, Jackson Woodruff, Elizabeth Polgreen, and Michael F. P. O'Boyle. 2023. C2TACO: Co-designing Program Synthesis and Domain-Specific Code Generators. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE)*. 19–31. doi:10.1145/3624007.3624044

[18] Mathieu Fehr, Jeff Niu, River Riddle, Mehdi Amini, Zhendong Su, and Tobias Grosser. 2022. IRDL: an IR definition language for SSA compilers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 199–212.

[19] Mathieu Fehr, Michel Weber, Christian Ulmann, Alexandre Lopoukhine, Martin Paul Lücke, Théo Degioanni, Christos Vasiladiotis, Michel Steuwer, and Tobias Grosser. 2025. xdsl: Sidekick compilation for ssa-based compilers. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. 179–192.

[20] Paul F. Fischer, James W. Lottes, and Stefan G. Kerkemeier. 2019. Nek5000 Version 19.0. https://nek5000.mcs.anl.gov.

[21] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. 2019. Xilinx Adaptive Compute Acceleration Platform: Versal Architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 84–89. doi:10.1145/3289602.3303991

[22] Lukas Gianinazzi, Tal Ben-Nun, and Torsten Hoefler. 2025. SPADA: A Spatial Dataflow Architecture Programming Language. arXiv:2511.09447 [cs.PL]

[23] Zane Hambly. 2026. Booth. https://github.com/Zaneham/Booth

[24] Roger A. Horn and Charles R. Johnson. 2013. *Matrix Analysis* (2 ed.). Cambridge University Press, Cambridge. Section 7.2 on Gram matrices.

[25] Erika Hunhoff, Joseph Melber, Kristof Denolf, Andra Bisca, Samuel Bayliss, Stephen Neuendorffer, Jeff Fifield, Jack Lo, Pranathi Vasireddy, Phil James-Roxby, et al. 2025. Efficiency, expressivity, and extensibility in a close-to-metal npu programming interface. In *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 85–94.

[26] Ken Kennedy and John R Allen. 2001. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc.

[27] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 296–311. doi:10.1145/3192366.3192379

[28] Jacob Lambert, Seyong Lee, Jeffrey S. Vetter, and Allen D. Malony. 2020. CCAMP: An Integrated Translation and Optimization Framework for OpenACC and OpenMP. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14. doi:10.1109/SC41405.2020.00040

[29] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 2–14. doi:10.1109/CGO51591.2021.9370308

[30] J. E. Lennard-Jones. 1924. On the Determination of Molecular Fields. II. From the Equation of State of a Gas. *Proceedings of the Royal Society*

A 106, 738 (1924), 463–477.

[31] Shuaijiang Li, Jiacheng Zhao, Ying Liu, Shuoming Zhang, Lei Chen, Yijin Li, Yangyu Zhang, Zhicheng Li, Runyu Zhou, Xiyu Shi, Chunwei Xia, Yuan Wen, Xiaobing Feng, and Huimin Cui. 2026. From Threads to Tiles: T2T, a Compiler for CUDA-to-NPU Translation via 2D Vectorization. In *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 362–374. doi:10.1109/CGO68049.2026.11395190

[32] Wei Li, Zhenyu Bai, Heru Wang, Pranav Dangi, Zhiqiang Zhang, Cheng Tan, Huiying Lan, Weng-Fai Wong, and Tulika Mitra. 2026. TileLoom: Automatic Dataflow Planning for Tile-Based Languages on Spatial Dataflow Accelerators. arXiv:2512.22168 [cs.DC] https://arxiv.org/abs/2512.22168

[33] Asher Mancinelli. 2025. *An Ideal Array Language*. Accessed: 2026-09-08. https://www.ashermancinelli.com/csblog/2025-7-20-Ideal-Array-Language.html

[34] NCAR CISL Application Scalability and Performance Group. 2025. SWM: Shallow Water Model Mini-App. https://github.com/NCAR/SWM. Documentation: https://swm-documentation.readthedocs.io/.

[35] Anthony T. Patera. 1984. A spectral element method for fluid dynamics: Laminar flow in a channel expansion. *J. Comput. Phys.* 54, 3 (1984), 468–488.

[36] Lorenzo Piarulli and Daniele De Sensi. 2026. Stencil Computations on Tenstorrent Wormhole. arXiv:2605.07599 [cs.DC] https://arxiv.org/abs/2605.07599

[37] Hiari Pizzini Cavagna, Daniele Cesarini, and Andrea Bartolini. 2025. Assessing Tenstorrent's RISC-V MatMul Acceleration Capabilities. In *High Performance Computing: ISC High Performance 2025 International Workshops*. Springer, 123–134. arXiv:2505.06085 [cs.PF]

[38] Jie Qiu, Colin Cai, Sahil Bhatia, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung. 2024. Tenspiler: A Verified-Lifting-Based Compiler for Tensor Operations. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (LIPIcs, Vol. 313)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 32:1–32:28. doi:10.4230/LIPIcs.ECOOP.2024.32

[39] Dimitri Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams, and Brennan Saeta. 2022. Implementation Strategies for Mutable Value Semantics. *J. Object Technol.* 21, 2 (2022), 2–1.

[40] Gabriel Rodriguez-Canal, Nick Brown, Tim Dykes, Jess Jones, and Utz-Uwe Haus. 2023. Fortran High-Level Synthesis: Reducing the barriers to accelerating HPC codes on FPGAs. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 10–18.

[41] Gabriel Rodriguez-Canal, Nick Brown, Maurice Jamieson, Emilien Bauer, Anton Lydike, and Tobias Grosser. 2023. Stencil-HMLS: A multi-layered approach to the automatic optimisation of stencil codes on FPGA. In *Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. 556–565.

[42] Gabriel Rodriguez-Canal, David Katz, and Nick Brown. 2025. An MLIR pipeline for offloading Fortran to FPGAs via OpenMP. In *Proceedings of the SC'25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1096–1103.

[43] Nicolai Stawinoga, David Katz, Anton Lydike, Justs Zarins, Nick Brown, George Bisbas, and Tobias Grosser. 2026. An MLIR Lowering Pipeline for Stencils at Wafer-Scale. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ACM, 94–109. doi:10.1145/3779212.3790124

[44] Maya Taylor, Carl Pearson, Luc Berger-Vergiat, Giovanni Long, and Jan Ciesko. 2026. Numerical Kernels on a Spatial Accelerator: A Study of Tenstorrent Wormhole. arXiv:2603.23343 [cs.PF] https://arxiv.org/abs/2603.23343

[45] TechPowerUp. [n. d.]. *AMD EPYC 9825*. TechPowerUp. https://www.techpowerup.com/cpu-specs/epyc-9825.c3906

[46] Tenstorrent. 2025. *Tenstorrent ISA Documentation*. https://github.com/tenstorrent/tt-isa-documentation.

[47] Tenstorrent. 2025. tt-mlir: Tenstorrent MLIR Compiler. https://github.com/tenstorrent/tt-mlir. Accessed 2026-09-09.

[48] Tenstorrent. 2025. *Wormhole Tensix Processor*. https://cdn.sanity.io/files/jpb4ed5r/production/6217a901d675d37ccb3cf1e8ba74f91a9f992577.pdf.

[49] Tenstorrent. 2026. tt-lang: A Python-Based Domain-Specific Language for Authoring Custom Operations on Tenstorrent Hardware. https://github.com/tenstorrent/tt-lang. Accessed 2026-09-09.

[50] Tenstorrent Inc. [n. d.]. *Tenstorrent Cards*. Tenstorrent Inc. https://tenstorrent.com/en/hardware/cards

[51] Tenstorrent Inc. 2024. *Compute Engines and Data Flow within Tensix*. Tenstorrent Inc. TT-Metalium Advanced Topics Documentation. https://docs.tenstorrent.com/tt-metal/latest/tt-metalium/tt_metal/advanced_topics/compute_engines_and_dataflow_within_tensix.html

[52] Yuxuan Wang, Cristian Tirelli, Lara Orlandic, Juan Sapriza, Rubén Rodríguez Álvarez, Giovanni Ansaloni, Laura Pozzi, and David Atienza. 2025. An MLIR-based Compilation Framework for CGRA Application Deployment. In *Applied Reconfigurable Computing. Architectures, Tools, and Applications (ARC 2025)*. Springer, 33–50.

[53] Yiwei Yang, Yusheng Zheng, Tong Yu, and Andi Quinn. 2025. HetGPU: The pursuit of making binary compatibility towards GPUs. arXiv:2506.15993 [cs.AR] https://arxiv.org/abs/2506.15993