Graphene: An IR for Optimized Tensor Computations on GPUs

Bastian Hagedorn
bhagedorn@nvidia.com
NVIDIA
Germany

Bin Fan
binf@nvidia.com
NVIDIA
USA

Hanfeng Chen
hanfengc@nvidia.com
NVIDIA
USA

Cris Cecka
ccecka@nvidia.com
NVIDIA
USA

Michael Garland
mgarland@nvidia.com
NVIDIA
USA

Vinod Grover
vgrover@nvidia.com
NVIDIA
USA

ABSTRACT

Modern GPUs accelerate computations and data movements of multi-dimensional tensors in hardware. However, expressing optimized tensor computations in software is extremely challenging even for experts. Languages like CUDA C++ are centered around flat buffers in one-dimensional memory and lack reasonable abstractions for multi-dimensional data and threads. Existing tensor IRs are not expressive enough to represent the complex data-to-thread mappings required by the GPU tensor instructions.

In this paper, we introduce Graphene, an intermediate representation (IR) for optimized tensor computations on GPUs. Graphene is a low-level target language for tensor compilers and performance experts while being closer to the domain of tensor computations than languages offering the same level of control such as CUDA C++ and PTX. In Graphene, multi-dimensional data and threads are represented as first-class tensors. Graphene's tensors are hierarchically decomposable into tiles allowing to represent optimized tensor computations as mappings between data and thread tiles.

We evaluate Graphene using some of the most important tensor computations in deep learning today, including GEMM, Multi-Layer Perceptron (MLP), Layernorm, LSTM, and Fused Multi-Head Attention (FMHA). We show that Graphene is capable of expressing all optimizations required to achieve the same practical peak performance as existing library implementations. Fused kernels beyond library routines expressed in Graphene significantly improve the end-to-end inference performance of Transformer networks and match or outperform the performance of cuBLAS(Lt), cuDNN, and custom handwritten kernels.

CCS CONCEPTS

• **Software and its engineering** → *Parallel programming languages*; **Compilers**; **Software performance**.

KEYWORDS

Intermediate Representation, Optimization, Tensor Computations, GPU, Compiler, Deep Learning, Code Generation

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9918-0/23/03.
https://doi.org/10.1145/3582016.3582018

ACM Reference Format:
Bastian Hagedorn, Bin Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover. 2023. Graphene: An IR for Optimized Tensor Computations on GPUs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3582016.3582018

1 INTRODUCTION

Driven by the need to accelerate deep learning computations, modern GPUs provide instructions to manipulate multi-dimensional tensors. NVIDIA's Volta architecture [14] introduced *Tensor Cores* to compute small matrix multiplications in hardware. Later architectures like Turing [15] and Ampere [16] expose instructions to move multi-dimensional tensors through the GPU memory hierarchy. This trend is continuing and these instructions are mandatory for achieving peak performance on modern GPUs.

Even though the GPU hardware became more tensor-oriented, the software to expose this functionality to programmers and compilers has not changed fundamentally. Most of the fast tensor instructions are only exposed in PTX [19], NVIDIA's virtual ISA, and therefore require low-level CUDA C++ code with inline assembly to be targeted. CUDA C++ has no standard abstraction for tensors to represent multi-dimensional data or threads. This makes expressing optimized tensor computations extremely challenging even for experts.

State-of-the-art tensor IRs like OpenAI Triton [25], or TensorIR [7], and deep learning compilers like XLA [8] or TVM [5] typically follow one of three approaches for representing optimized tensor computations:

(1) They rely on vendor library kernels and leave the challenge of representing optimized tensor programs to the library developers who again use CUDA C++ and PTX, or
(2) They expose the fast PTX instructions as higher-level built-ins operating on tensors (e.g., using TVM's *tensorize* primitive [27], or MLIR's *gpu* dialect [13]) but lack abstractions for some of the most important instructions since they require complex data-to-thread mappings and layouts that cannot be expressed in existing IRs, or
(3) They expose a high-level surface language to programmers while the heavy lifting of generating high-performance GPU code is performed by complex built-in transformation passes (e.g., Triton's instruction selection [26]) whose extension requires intimate knowledge of full-fledged compilers.

In this paper, we introduce Graphene, an IR for representing optimized tensor computations on GPUs. Graphene is expressive enough to explicitly represent highly optimized GPU code targeting tensor instructions and aims to serve as an alternative target language to CUDA C++/PTX that is closer to the domain of tensor computations. We make the following contributions:

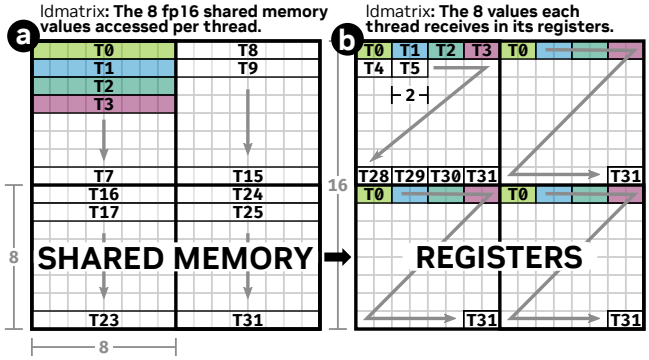
- We introduce *a novel representation for tensor shapes, layouts and tiles*. Graphene's tensors are decomposable into tiles represented as smaller nested tensors. The layout of tensor elements need not be contiguous in memory and we are able to represent the complex shapes and layouts required by GPU tensor instructions.
- We introduce *logical thread groups* for representing the GPU compute hierarchy as a tensor of processing elements. Representing threads as tensors enables reshaping and tiling threads arbitrarily just like we manipulate data tensors and it minimizes the required built-in hierarchies.
- We introduce *decomposable specifications* (specs) as an abstraction for collective computations and data movements. Specs map data to thread tensors and represent computations ranging from device-level kernels to thread-level executable instructions. We expose GPU instructions as *atomic specifications* that describe their semantics as operations on tensors. Optimized GPU kernels are expressed by hierarchically decomposing kernel-level specs into atomic specs for which we know how to generate code.
- In our *experimental evaluation*, we show that Graphene is capable of not only representing the fastest single operator implementations (GEMM) but also some of the most important fused operations in deep learning: GEMM + pointwise epilogues, MLP, LSTM, Layernorm, and FMHA. We evaluate our kernels in isolation and in end-to-end inference benchmarks using various Transformer networks.

2 OPTIMIZED GPU DATA MOVEMENTS

Consider the implementation of an optimized GPU data movement. NVIDIA's Ampere GPUs are capable of moving two-dimensional tensors through the memory hierarchy with a single instruction. Specifically, the `ldmatrix` instruction uses a warp (32 threads) to move up to four 8×8 matrices from shared memory into the registers of the warp's threads. `ldmatrix` is relied upon heavily in fast library kernels and replacing it with equivalent but simpler data movements in GEMM kernels causes performance drops by as much as 17%.

The `ldmatrix` instruction prescribes a strict data-to-thread mapping. Figure 1a shows the values each thread must access in shared memory and Figure 1b shows the values it receives in registers after executing the instruction. Conceptually, a warp is tiled into four groups of eight threads. Each 8-thread group is assigned to a unique 8×8 tile in shared memory. Each thread then accesses one row per 8×8 tile in shared memory (a) and receives two adjacent values per 8×8 tile, eight values in total, in return (b).

`ldmatrix` is only exposed in PTX and requires the code shown in Figure 1c to be targeted. This is the state-of-the-art when it comes to expressing optimized GPU tensor computations and it is the kind of code performance experts have to write and compilers have to reason about today. The multi-dimensional nature of the instruction, which is easily visualized in Figures 1a and b, is severely obfuscated



c CUDA C++/PTX data movement using ldmatrix.

```
1  // Reshape the warp into 2×2 groups of 8 threads
2  int thr_grp_m = ((threadIdx.x / 16) % 2);
3  int thr_grp_n = ((threadIdx.x / 8) % 2);
4  int grp_local_idx = (threadIdx.x % 8);
5  // Tile and map shared mem (see a) - also convert pointer
6  uint32_t src_ptr;
7  void *   dst_ptr = (void *) &smem[
8    (thr_grp_m * 128 + thr_grp_n * 8 + grp_local_idx * 16)];
9  asm volatile ("{ .reg .u64 src_ptr; \
10                 cvta.to.shared.u64 src_ptr, %1; \
11                 cvt.u32.u64 %0, src_ptr; }\n"
12     : "=r"(src_ptr)
13     : "l"(dst_ptr));
14 // Each thread receives 8 vals (2 vals per 8×8 tile) - see b
15 int *dst = (int *)&regs[0]; // dst[x] points to two fp16 vals
16 asm volatile ("ldmatrix.sync.aligned.m8n8.x4.shared.b16 \
17     {%0, %1, %2, %3}, [%4];\n"
18    : "=r"(dst[0]), "=r"(dst[1]), "=r"(dst[2]), "=r"(dst[3])
19    :  "r"(src_ptr));
```

d Graphene data movement using ldmatrix.

```
1  %1:[16, 16].fp16.SH // src } Tensors (Sec.3)
2  %2:[ 2,  4].fp16.RF // dst
3  #3:[ 1].block               } Logical Thread Groups (Sec.4)
4  #4:[32].thread
5  %2 ← Move<<<#3, #4>>>(%1) { // Decomposable Specs (Sec.5)
6       // Reshape the warp into 2×2 groups of 8 threads
7       #5:  [4].[8].thread = #4.tile(8)
8       #6:[2,2].[8].thread = #5.reshape(0, [2,2])
9       (@thr_grp_m, @thr_grp_n), @grp_local_idx = #6.indices()
10
11      // Tile and map shared mem (see a) one 1×8 row per thread.
12      %7:[2, 2].[8, 8].fp16.SH = %1.tile([8, 8])
13      %8:       [8, 8].fp16.SH = %7[@thr_grp_m, @thr_grp_n]
14      %9:[8, 1].[1, 8].fp16.SH = %8.tile([1, 8])
15      %10:      [1, 8].fp16.SH = %9[@grp_local_idx, 0]
16
17      // Each thread receives 8 vals - 2×2 * 1×2 (see b)
18      %11:[2, 2].[1, 2].fp16.RF = %2.tile([1, 2])
19      %11 ← Move<<<#3, #4>>>(%10) }
```

Figure 1: The `ldmatrix` instruction uses a warp (32 threads) to copy 16×16 fp16 shared memory values into registers. The values each thread must access in shared memory (a) differ from the values it receives after executing the instruction (b). This instruction is only exposed in PTX and must be used as shown in (c). Graphene's representation of the tensorized data movement is shown in (d).

in the code: In CUDA C++, the conceptual reshaping of a warp into 2×2 8-thread groups must be expressed as a set of scalar thread index manipulations (lines 1-3). Those are then used in a scalar index for accessing a one-dimensional buffer in shared memory (lines 7,8). Due to the way CUDA and PTX handle address spaces, we must further cast the access of the shared memory buffer into a valid shared memory pointer (lines 9-13), and then call `ldmatrix` (lines 16-19).

Due to the lack of multi-dimensional tensors, this representation of an optimized tensor data movement is incredibly challenging to write and hard to reason about. To the best of our knowledge, existing tensor IRs are unable to represent data movements using `ldmatrix` because they lack expressiveness to specify the prescribed data-to-thread mappings.

Figure 1d shows how the same optimized data movement is represented in Graphene. The required data tensors and the available GPU processing elements are declared in the first four lines. We then describe a data movement (`Move`) executed per thread-block (`#3`) and per warp (`#4`) of a 16×16 shared memory tensor (`%1`) into 2×4 registers (`%2`) (line 5). Note that registers are thread-local. 2×4 values times 32 threads corresponds to 256 values in total, enough to contain the whole source shared memory buffer.

The implementation in curly braces (lines 7-19) explicitly applies the mapping prescribed by `ldmatrix` (see Figures 1a and b). First, we tile the warp into 4 *logical thread groups* with 8 threads each and reshape the groups into a 2×2 shape (lines 7-9). Then, the source tensor is tiled into four 8×8 matrices (line 12), one per thread group (line 13). Each 8×8 tile is further tiled into rows (line 14) which are assigned to the individual threads (line 15). Finally, the destination tensor is tiled (lines 18) and we specify another `Move` operating on the tiles defined earlier. This final `Move` matches the pre-defined semantics of the `ldmatrix` instruction and is considered an *atomic specification* (see Section 5.2) in Graphene that needs no further implementation. Given the IR shown in Figure 1d, Graphene generates the CUDA C++ code shown above.

3 THE SHAPE OF TENSORS TO COME

In this section, we introduce Graphene's tensors, their shapes, and layouts. Multi-dimensional tensors and decomposing tensors into tiles have already been studied in polyhedral compilation. However, Graphene's tensor representation is particularly suited for expressing optimized GPU computations because of two main reasons: 1) We use a concise notation for hierarchically decomposed tensors into tiles which are simply smaller nested tensors themselves. Hierarchical tiles are required for mapping data to the multi-layered compute and memory hierarchy of the GPU. 2) We allow the expression of non-trivial shapes, such as tensors containing non-contiguous elements and tensors with *swizzled* memory layouts beyond the standard "row/column-major" or "NHWC" layouts. Such layouts are performance-critical, for example, when storing intermediate tensors to shared memory. Existing tensor IRs that typically use integer lists for specifying the shapes and strides of tensors cannot express such layouts. Graphene's tensor representation explicitly captures all shapes and layouts that are used in handwritten optimized implementations today. Graphene's shape notation is inspired by and builds upon NVIDIA's CuTe programming model [1, 24], specifically its shape algebra [17].

3.1 Expressing Tensors in Graphene

Figure 2 shows Graphene's syntax for tensors. For now, we focus on data tensors. Expressions formed by this syntax are capable of representing the tensors discussed in the previous sections as first-class citizens. In Graphene, every tensor consists of a name, a shape, an element type, and a label of where it is stored in the GPU

$$
\begin{aligned}
\mathit{Tensor} &= \mathit{Name} : \mathit{Shape} \,.\, \mathit{ElementType} \,.\, \mathit{Memory} \\
\mathit{Name} &= \texttt{\%string} \\
\mathit{Shape} &= \begin{bmatrix} \mathit{Dims} \\ \mathit{Stride} \end{bmatrix} \text{ (\texttt{[dims:stride]} in listings)} \\
\mathit{Dims} &= \mathit{Stride} = \mathit{IntTuple} \\
\mathit{IntTuple} &= (\mathit{Size}, ..., \mathit{Size}) \\
\mathit{Size} &= \mathit{IntExpr} \mid \mathit{IntTuple} \\
\mathit{IntExpr} &= \texttt{int} \mid \texttt{var} \mid (\mathit{IntExpr}\ \mathit{BinOp}\ \mathit{IntExpr}) \\
\mathit{BinOp} &= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{...} \\
\mathit{ElementType} &= \mathit{ScalarType} \mid \mathit{Shape} \,.\, \mathit{ElementType} \\
\mathit{ScalarType} &= \texttt{fp16} \mid \texttt{fp32} \mid \texttt{i32} \mid ... \\
\mathit{Memory} &= \texttt{GL (global)} \mid \texttt{SH (shared)} \mid \texttt{RF (registers)}
\end{aligned}
$$

Figure 2: Syntax for specifying data tensors in Graphene. *IntTuple* and *ElementType* are defined recursively which enables the expression of advanced tensor layouts and tiles.

memory hierarchy. We support the standard CUDA memory regions, global memory (off-chip), shared memory (on-chip, shared by threads within a thread-block) and registers (thread-local). We omit memory labels in the following examples for brevity. A shape is composed of *dimensions* and *strides*, both represented by integer tuples. Note that we used a simplified notation in Figure 1 and omitted strides for all tensors. For example, `A:[16,16].fp16.SH` is actually represented as a row-major tensor `A:[(16,16):(16,1)].fp16.SH` with the colon separating the dimensions and strides. Graphene enables the expression of advanced tensors in two ways:

(1) *Hierarchical Dimensions: IntTuple* is defined recursively. This allows to define the size and stride of a single dimension using multiple integers. This serves a dual purpose: a. It enables the expression of complex memory layouts (Section 3.2), and b. it enables the expression of tensors with non-contiguous elements (Section 3.3).
(2) *Tiles:* The *ElementType* of a tensor is defined recursively and might be another nested *Shape*. We use this to represent hierarchically tiled tensors where the outer (i.e., left) shape represents the arrangement of tiles and the inner shape represents the arrangement of elements within a tile.

3.2 Memory Layouts

Figure 3 shows different examples of laying out a two-dimensional 4×8 tensor in memory. Figures 3a and b show the standard column- and row-major layouts specified with the according strides. The position of a logical coordinate (i, j) in physical one-dimensional memory is obtained by computing the dot-product of the coordinates and the strides of the tensor (as indicated by the gray numbers within the tensor).

Representing memory layouts with strides like this is the standard way in existing tensor IRs. However, this representation limits the expressible memory layouts to only such layouts in which *all* elements of one dimension strictly appear before elements of the

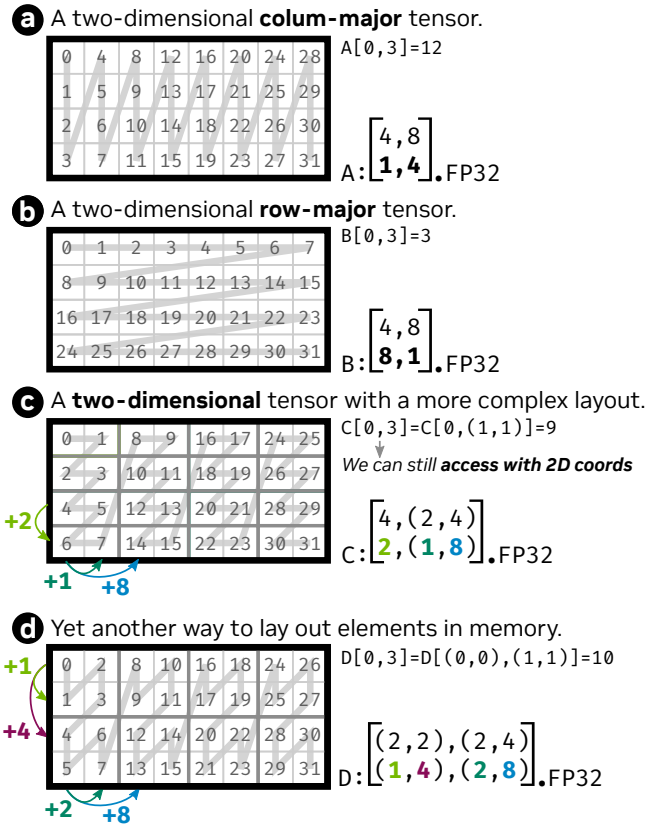


Figure 3: Examples for representing different memory layouts for a two-dimensional 4×8 tensor. Defining a dimension as a tuple of integers (c,d) enables the specification of multiple strides per dimension. This allows Graphene to specify more complex memory layouts beyond row- and column-major layouts.

other dimensions. For example, in the 2D row-major layout, as shown in Figure 3b, we fix a row, then traverse *all* columns before moving to the next row. Merely padded layouts, in which the stride of one dimension exceeds the size of the previous dimension (e.g., `[(4,8):(9,1)]`), are additionally expressible in this format.

This limitation is especially problematic when it comes to storing intermediate results of optimized tensor computations temporarily to shared memory. Depending on how threads read and write those intermediate tensors, more complex layouts are required to use the hardware as efficiently as possible. For example, shared memory on the GPU is organized in banks which can only ever serve one thread at a time. As soon as multiple threads try to access different values stored in the same bank (a so-called bank conflict), the accesses of all conflicting threads must be serialized which significantly hurts performance because it leads to pipeline stalls due to increased memory latency. Optimized kernels achieving peak performance regularly lay out tensors in more complex ways beyond the simpler layouts we described so far.

Figures 3c and d visualize two such complex memory layouts and their representation in Graphene. Figure 3c shows a two-dimensional 4×8 tensor whose second dimension is represented as an integer tuple instead of a single integer. We call this a hierarchical dimension. Hierarchical dimensions enable us to specify more than one stride per dimension. In this case, we lay out two adjacent column values contiguously in memory but then logically move down the rows first before moving the next two adjacent column values. Figure 3d shows a similar yet slightly more complex layout that uses hierarchical dimensions twice.

The key takeaway is that hierarchical dimensions do not increase the rank of the tensor. We can still access the tensors shown in Figures 3c and d with logical two-dimensional coordinates, and then internally compute the corresponding hierarchical coordinate. This enables to optimize and specify the layout of a tensor once, e.g., at the time of its allocation. Afterwards, regardless of how the tensor is laid out in memory, one can keep the same two-dimensional logical coordinates for accessing the tensor. Expressing such layouts in CUDA is possible too but requires complex indexing arithmetic which additionally has to be adjusted at every tensor access throughout the kernel with every layout change.

3.3 Tiling Tensors

Expressing optimized tensor computations for GPUs requires the specification of how multi-dimensional tensors are mapped to the multi-layered compute and memory hierarchy. Graphene's key abstraction for representing such mappings are tiles. Figure 4 shows a basic 4×8 tensor (Figure 4a) and three different examples (b,c,d) for tiling it.

Regular contiguous tiles. Tiles are represented as nested shapes in Graphene. The *ElementType* of a tiled tensor is simply another shape describing the tiles. Figure 4b shows a tiled version of Figure 4a in which B describes a tensor containing four elements (the tiles arranged as 2×2), where each element is a tile of 2×4 floating point elements. Note that as a convention, the strides of *all* shapes specify the distance between the elements of innermost scalar type in memory for simplifying indexing arithmetic during Graphene's code generation. Hence, the $(2, 16)$-stride of the outer (i.e. left) shape in Figure 4b specifies that for moving to the next row tile, one must skip two scalar elements and to move to the next column tile, one must skip 16 elements.

Typically, tile sizes are specified with a single integer per dimension for specifying how many elements each tile contains per dimension. In Graphene, tile sizes are specified with one-dimensional tensors per dimension. The tile sizes `([2:1],[4:1])` used in Figure 4b are interpreted as follows: Two logically adjacent elements in the first dimension (`[2:1]`) and four logically adjacent elements in the second dimension (`[4:1]`) form a tile. As a convention, we omit the strides from unit-strided tensors and can specify tile sizes as shown in Figure 1. Note that this way of specifying tile sizes is agnostic of the memory layout of the tensor to be tiled. Both tile size arguments specify a stride of one, meaning that we want to group logically adjacent elements into the same tile regardless of how they are arranged in physical memory. The strides of the resulting tensor B depend on the strides of tensor A and are computed automatically.

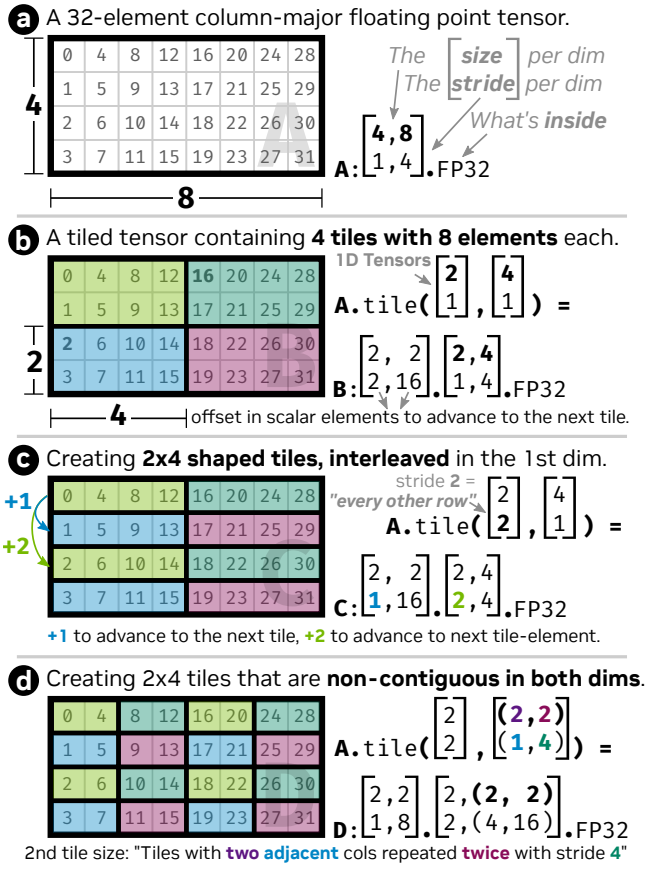


Figure 4: Examples for expressing and tiling tensors in Graphene. Tiles are simply nested smaller tensors. Tile sizes are specified with 1D tensors as well. This allows the specification of traditional contiguous tiles (b) as well as non-contiguous tile in one (c) or more dimensions (d). A single dimension can be represented by an integer-tuple (d) which allows to specify more than one stride per dimension.

Non-contiguous tiles. Using tensors as tile size arguments becomes especially useful once we need to specify tiles that are non-contiguous in one or more dimensions. For example, Figure 4c shows how to tile tensor A into 2×4 shaped tiles that are interleaved in the first dimension. In the visualization, elements of the same color still belong to the same tile. Increasing the stride from `[2:1]`, as used in Figure 4b to group two logically adjacent rows, to `[2:2]` now creates tiles that logically contain every other row. The strides in the resulting tensor C reflect this change as well.

Finally, one can also use a one dimensional tensor with a hierarchical dimension as a tile size argument as shown in Figure 4d. Here, we are representing tiles that again contain every other row but now additionally the column dimension is non-contiguous as well. Specifically, each tile contains *two* logically *adjacent* columns (`[2:1]`) repeated *twice* with a stride of *4* (`[2:4]`), as indicated by the colors. Note that we have already seen similar non-contiguous tiles in two dimensions being used in highly optimized GPU tensor computations like the `ldmatrix` example shown in Figure 1.

3.4 Parametric Shapes and Partial Tiles

Even though all examples in this paper show concrete integer values for dimensions and strides, Graphene is capable of representing tensor with parametric (i.e. symbolic) shapes such as `[M,N].fp32`. This is particularly important for generating kernels used in neural networks with dynamic shapes. Parametric shapes lead to additional kernel parameters during code generation. Index expressions using parametric shapes are simplified using algebraic simplification rules such as `(M % 256)` $\longrightarrow$ `M iff M < 256`.

Tile sizes that do not evenly divide the dimensions of the input tensor lead to one or more so-called partial tiles (e.g., tiling `[1023].fp32` with a tile size of 128). For representing implementations that involve partial tiles, Graphene applies the CuTe approach of overapproximating the involved shapes [18]. Subsequent accesses to tensors with potentially partial tiles must be predicated to prevent out-of-bounds accesses.

4 LOGICAL THREAD GROUPS

Efficient tensor computations require different arrangements of threads at different points in the kernel. For example, the thread arrangement during a data movement using `ldmatrix` differs significantly from the arrangement required for computing a matrix multiplication using the Tensor Core `mma.m8n8k4` instruction [20]. The `ldmatrix` instruction, as discussed in Section 2, is executed with a group of 8 threads within a warp. However, the `mma.m8n8k4` instruction is executed with a different set of eight specific *non-contiguous* threads per warp called quad-pairs. Without Graphene, one must establish all arrangements of threads as a set of scalar index computations and carefully orchestrate how threads are mapped to the data tensors (e.g., see Figure 1c).

In Graphene, we represent the GPU compute hierarchy as a tensor. This representation allows to manipulate threads just like data. Tiling and reshaping thread tensors enables expressing thread arrangements as *logical thread groups*. The advantages of our approach are two-fold:

(1) Explicit shapes: Logical thread groups explicitly represent the arrangement of threads as a multi-dimensional tiled tensor instead of using several scalar thread-index computations. High-performance kernels typically require differing arrangements within a single kernel leading to a plethora of such index transformations. In Graphene, we simply have differently tiled thread tensors and automatically generate the required scalar index expressions at code generation time.
(2) Flexible hierarchies: Specific groups are only required when targeting a certain architecture. For example, quad-pairs have been introduced with Volta and vanished again with more recent architectures. It is desirable to represent high-performance code for *all* architectures (including future ones), however, without adding built-in support for specific compute hierarchies. Logical thread groups enable the expression of arbitrary thread arrangements.

Threads as tensors. The syntax for representing threads as tensors differs slightly from the syntax shown in Figure 2 for representing data tensors. The *ScalarType* of a thread tensor is either `thread` or `block`, echoing the two fundamental hierarchies in CUDA C++,

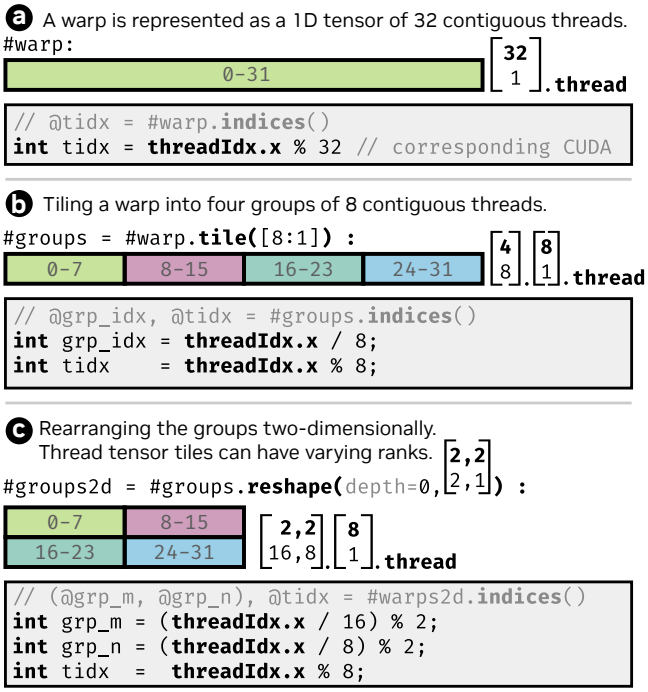


Figure 5: Representing the thread arrangment required for the `ldmatrix` instruction in Graphene: A warp is tiled into four groups arranged, as 2×2, of eight contiguous threads.

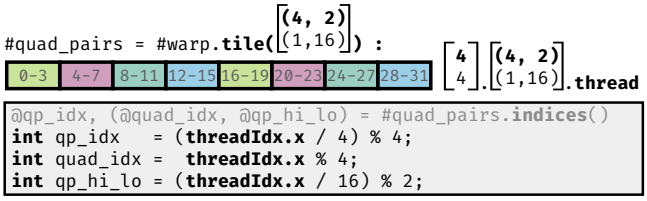


Figure 6: Representing non-contiguous quad-pairs required for Volta's Tensor Core instructions as logical thread groups.

Graphene's code generation target language. The *Memory* label used for data tensors is not needed for representing thread tensors and is thus dropped. As a convention, to visually distinguish thread from data tensors, we use `%` as a prefix for data tensor names and `#` as a prefix for thread tensor names.

Examples. Figure 5 shows how we represent the `ldmatrix` thread arrangement in Graphene as discussed in Section 2. The corresponding scalar thread index expressions for representing the equivalent arrangement of threads in CUDA C++ are shown in the gray boxes.

We begin with a one-dimensional tensor of 32 contiguous threads representing a warp (Figure 5a). This warp is tiled, just like we tile data tensors as discussed in the previous section, into four groups of eight threads as shown in Figure 5b. Afterwards, we rearrange the four tiles into a two-dimensional 2×2 shape by applying the *reshape* function at the outermost level of the tiled tensor (depth 0, Figure 5c). In Graphene, tiled thread tensors are

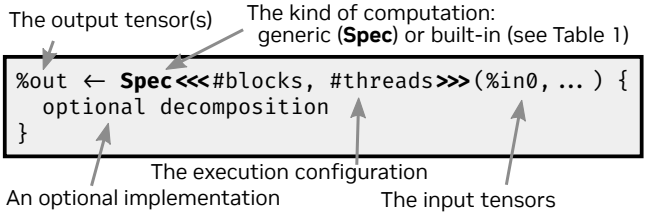


Figure 7: Syntax for using specs (computations on tensors).

allowed to have varying ranks for nested tiles. Here for example, the tiles are arranged two-dimensionally whereas the threads within the tiles are one-dimensional. The corresponding index expressions are computed automatically during CUDA C++ code generation.

Figure 6 shows how to represent the quad-pairs required for executing Volta's `mma.m8n8k4` instruction in Graphene. A quad-pair is a group of eight threads, composed of two specific quads, a group of 4 contiguous threads. For example, the first quad-pair is composed of threads 0-3 and threads 16-19. This arrangement is prescribed by the hardware [20] and must be strictly followed to avoid undefined behavior when using Volta's Tensor Cores. In Graphene, this arrangement is expressed by tiling the warp with a non-contiguous tensor `[(4,2):(1,16)]` which precisely describes the shape and layout of the quad-pairs.

5 SPECIFICATIONS AND DECOMPOSITIONS

In this section, we discuss how Graphene represents optimized tensor computations using data and thread tensors. For the remaining section, when we use the term *computation*, we refer to both operations on tensors *and* data movements.

5.1 Specifications for Expressing Computations

Computations on tensors are represented by so-called *Specifications* (specs). This concept is inspired by Fireiron [9]. The key idea is that specs encapsulate a self-contained block of computation, like a device-level matrix multiplication kernel or a warp-level data movement. Figure 7 shows the syntax for using specs in Graphene. A spec captures its input and output tensors as well as an execution configuration that describes the available threads for executing this computation. The optional decomposition describes how this computation is implemented and might contain simple control flow or other nested specs. For example, we typically begin with a spec describing the kernel-level computation. We then gradually describe its implementation by decomposing it into more fine-grained specs (typically working on tiles of data and thread tensors) until only such specs remain for which we know how to generate code. Those remaining specs are called *atomic specifications* because they need no further decomposition.

For example, a GEMM kernel can be decomposed into "smaller" computational building blocks, e.g., thread-block-level matrix multiplications operating on tiles and data movements of such tiles between levels of the memory hierarchy. Figure 1d showed a concrete example. We declare a warp-level data movement from shared memory to registers (line 5) and gradually decompose it into another nested data movement spec representing the `ldmatrix` instruction (line 19).

Table 1: Graphene's built-in specifications.

Spec	Description
`Move`	Data Movements
`MatMul`	Matrix Multiplications (e.g., Tensor Cores, FMA)
`UnaryPointwise`	Elementwise unary computations (e.g., `exp`)
`BinaryPointwise`	Elementwise binary computations (e.g., `add`)
`Reduction`	Reduce tensor along one or more axes
`Shfl`	Exchange tensor values within thread groups
`Init`	Uniformly assign scalar value to a tensor
`Allocate`	Introduce new temporary data tensor

5.2 Built-in Specifications

Graphene comes with a set of built-in specifications. Built-in specs describe a specific kind of computation while atomic specs are concrete instances of built-in specs and describe computations that are implemented by GPU instructions. For example, Graphene provides a built-in `Move` spec for explicitly representing data movements along with a set of pre-defined atomic `Moves` associated with different data movement instructions like `ldmatrix`.

Table 1 shows the complete set of Graphene's built-in specs. Besides `Moves`, Graphene provides other built-in specs whose atomic versions map to different kinds of instructions exposed in the ISA. `MatMul` represents matrix-multiplication-like computations and atomic `MatMuls` map to scalar and vectorized fused-multiply-add instructions as well as Tensor Core instructions. `Reduction` and `Unary-/BinaryPointwise` represent the expected computations. `Shfl` is used for expressing data movements, not between levels of the memory hierarchy (as represented by `Move`), but instead between specific groups of threads. Atomic `Shfls` map to warp-level `shfl.sync` PTX instructions. Finally, `Allocate` is used for introducing new temporary data tensors in the implementation of another spec, and `Init` is used to uniformly assign scalar values to a tensor.

Atomic Specs. Table 2 shows examples for atomic specifications in Graphene. During code generation, every spec without decomposition is matched against the set of pre-defined atomic specs for the target architecture. For example, whenever we encounter a `Move`, executed per thread, of one scalar floating point value from global memory to registers (row 1, Table 2), we emit the `ld.global.u32` PTX instruction which exactly implements this `Move`. When we see a `Move` of eight contiguous `fp16` values from global memory to registers, we emit the vectorized `ld.global.v4.u32` (second row).

Tensor instructions, like `ldmatrix`, are no longer executed per thread but instead cooperatively by groups of threads. Those instructions also no longer operate on scalars or one-dimensional vectors but instead on multi-dimensional tensors. Graphene's atomic specs explicitly capture the required arrangement of threads and tensor shapes. For example, Table 2 shows two atomic specs representing Tensor Core `mma` instructions. They require different thread arrangements and two-dimensional (tiled) input and output tensors which are all specified explicitly in our IR.

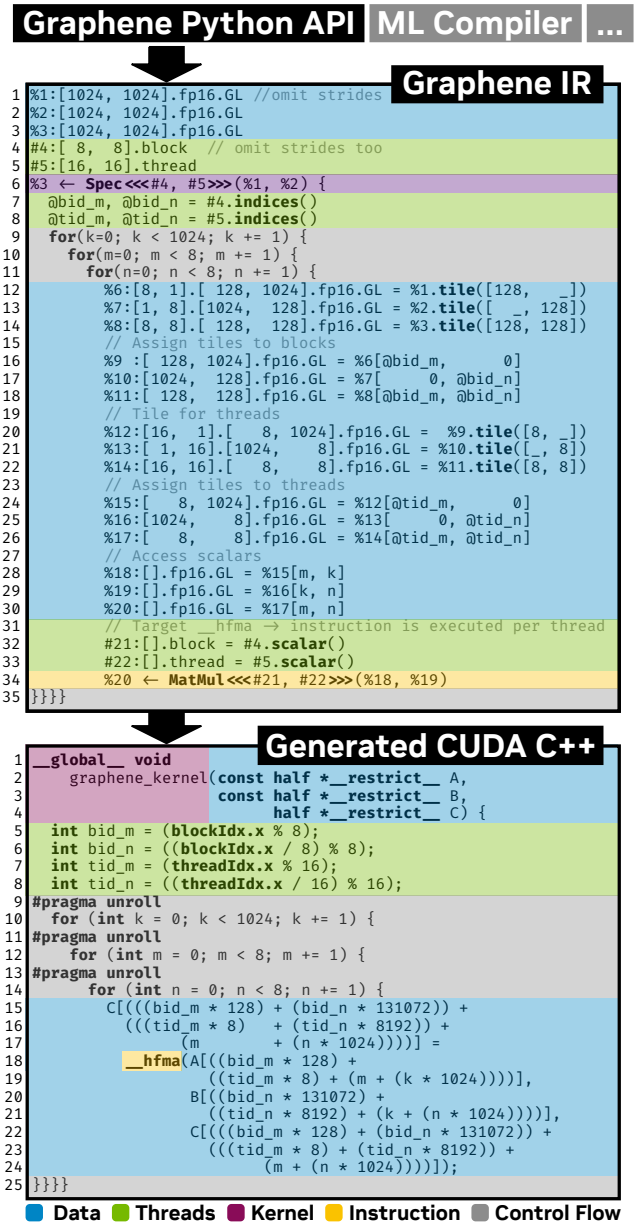


Figure 8: A simple but complete matrix multiplication kernel expressed in Graphene and the resulting CUDA C++ code after code generation. Graphene IR is generated from a simple Python API but could also be integrated into and generated by other machine learning compilers like XLA or TVM.

5.3 Representing Fused Kernels

Graphene is capable of representing all kinds of tensor computations on GPUs including those that cannot be represented by one of the built-in specs like `MatMul`. For example, *fused* kernels implement more than one operation on tensors in a single kernel, e.g., GEMM followed by pointwise operations. Fused kernels are heavily relied upon in modern deep learning networks to achieve the best possible performance. To represent fused computations, we use a generic `Spec`. Generic specs describe the required input and output tensors as well as the participating threads executing this computation. The computation this spec is representing is entirely defined by how it is decomposed.

Table 2: Examples for atomic specifications in Graphene and the associated (PTX) instructions they will be lowered into.

Spec	Threads	Inputs	Outputs	Associated Instruction
Move	[1].thread	[].fp32.GL	[].fp32.RF	ld.global.u32
Move	[1].thread	[8].fp16.GL	[8].fp16.RF	ld.global.v4.u32
Move	[1].thread	[4].fp32.RF	[4].fp32.SH	st.shared.v4.u32
Move	[32].thread	[1,8].fp16.SH	[2,2].[1,2].fp16.RF	ld.matrix.sync.aligned.m8n8.x4.shared.b16
BinaryPW<*>	[1].thread	[].fp16.RF, [].fp16.RF	[].fp16.RF	__hmul
BinaryPW<+>	[1].thread	[2].fp16.RF, [2].fp16.RF	[2].fp16.RF	__hadd2
MatMul	[1].thread	[].fp16.RF, [].fp16.RF	[].fp16.RF	__hfma
MatMul	[1].thread	[2].fp16.RF, [2].fp16.RF	[2].fp16.RF	__hfma2
MatMul	[1].thread	[].fp32.RF, [].fp32.RF	[].fp32.RF	__fmaf
MatMul	$\left[\begin{smallmatrix}(4,2)\\(1,16)\end{smallmatrix}\right]$.thread	[4,1].fp16.RF, [1,4].fp16.RF	[2,4].fp32.RF	mma.sync.aligned.m8n8k4.row.col.f32.f16.f16.f32
MatMul	[32].thread	[2,2].[1,2].fp16.RF, [2,1].[2.1].fp16.RF	[2,1].[1,2].fp32.RF	mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32

5.4 Example: A Simple GEMM Kernel

Figure 8 shows the simplest possible but complete decomposition of a kernel-level spec implementing matrix multiplication. Graphene IR is not meant to be written directly, due to its verbosity and redundant shape annotations. Today, we generate Graphene IR using a Python API. In the future, we envision Graphene to be integrated into existing deep learning compilers like XLA [8] or Triton [25] where it can serve as an alternative target language to CUDA C++ and PTX.

The Graphene code begins with describing the input and output tensors and the available blocks and threads for executing this computation (lines 1-5). The outermost spec (line 6) represents the CUDA C++ kernel. Graphene also provides basic control flow statements, including loops and if-statements, and other expressions not operating on tensors like synchronizations or barriers. In this case, we use a simple triple-nested for-loop (lines 9-11) for iterating over the scalar output elements computed per thread. We tile for thread-blocks (lines 12-18) and immediately tile again for threads (lines 20-26). Finally, we specify the sequential scalar computation per thread (line 34). This spec does not need a decomposition because it will match the pre-defined atomic `hfma` spec (compare the tensor types of the inner `MatMul` spec with the atomic `hfma` spec shown in Table 2). More optimized GEMM implementations describe multiple data movements and target vectorized and Tensor Core instructions instead.

5.5 Code Generation

Since Graphene IR precisely describes the implementation of tensor computations, generating CUDA C++ code boils down to printing the IR as valid CUDA C++. Graphene IR might contain specs, tensor manipulations, or control-flow like loops and conditionals and other expressions not involving tensors. Control flow statements, synchronizations and barriers are emitted using valid CUDA C++ syntax. A spec without implementation is matched against the set of atomic specs and we emit a call to the associated instruction as shown in Figure 8. For decomposed specs, we emit their implementation recursively and for tensor manipulations we build ASTs and compile those into thread index and buffer access expressions. The generated indices are arithmetically simplified.

6 EVALUATION

In this section, we seek answers to the following two questions: Can Graphene

(1) represent kernels that perform competitively to library implementations across different architectures?
(2) compete with handwritten *fused* kernels beyond libraries?

The quality of an IR for optimized tensor computations stands or falls with the performance of the kernels it is capable of representing. Hence, it must be as good as the state-of-the-art across existing architectures, and it must be able to generate kernels for all important tensor computations, including those beyond single operators like GEMMs.

Methodology. The experiments in this section were performed with two GPUs: a V100 (SM70, Volta architecture) and an RTX A6000 (SM86, Ampere architecture). We use CUDA-11.7, cuBLAS(Lt) version 11.10 and driver version 510.68.02. For measuring performance, we use NVIDIA's Nsight-Compute profiler (Version 2021.3.1.0) which automatically locks the clocks to base frequencies. If not otherwise mentioned, all experiments have been performed with FP16 tensors using FP32 Tensor Core accumulation.

All plots in this section show the evaluated tensor computation as a data-flow graph on the left-hand side. Data flows from top to bottom, tensors are denoted with capital (dark green) letters, scalars with lowercase (light green) greek letters.

Hypothesis A: Graphene can represent kernels that compete with high-performance library implementations.
We match the performance of cuBLAS and cuBLASLt which confirms that Graphene is capable of expressing all optimizations necessary for achieving practical peak performance.

GEMM is still the single most important and by far the best optimized tensor computation performed on GPUs today. Hence, the ultimate test for GPU code efficiency remains matching the performance of cuBLAS for general matrix multiplication (GEMM). NVIDIA's cuBLAS library provides many fast GEMM implementations and can be considered to deliver the practically achievable peak performance. Related work that reports outperfoming cuBLAS

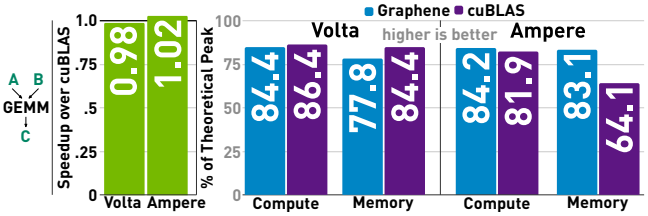


Figure 9: Graphene GEMM performance compared against cuBLAS. We compare speedup and achieved throughput of memory and compute as percentages of the theoretical peak (as reported by the NSight Compute profiler).

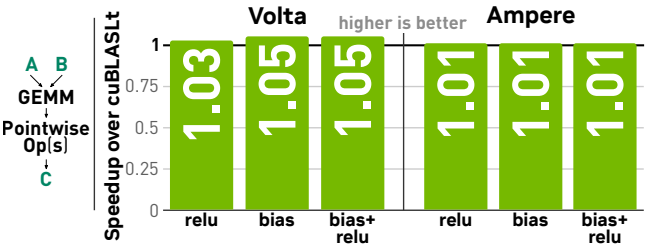


Figure 10: Graphene's performance compared to cuBLASLt for simple fused (GEMM + pointwise) kernels.

for GEMM often simply finds better tile sizes than the ones chosen by cuBLAS runtime heuristics [9].

In our first experiment, we are interested in whether Graphene is capable of representing optimized GEMMs that achieve the same performance as cuBLAS. Specifically, we are interested in how close we come to the theoretical peak performance of the GPU. For most precisely measuring the average utilization of the GPU Streaming Multiprocessor (SM), we choose the problem size such that it is large enough and evenly divides work among the available SMs.[1] We also ensured to use exactly the same tile sizes as those used by cuBLAS (as embedded in the kernel name visible in the profiler).

Figure 9 shows Graphene's GEMM performance compared to cuBLAS on the Volta and Ampere architectures. Our generated kernels exactly match the performance of cuBLAS on both architectures. Therefore, Graphene is capable of representing equivalent optimizations for achieving the highest GEMM performance possible on today's GPUs.

The GEMM kernels used in this experiment are compute-bound as can be seen by the achieved theoretical throughputs reported by the profiler as shown on the right-hand side of Figure 9. This indicates that the Tensor Cores run at maximum capacity. The Ampere cuBLAS kernel is slightly more efficient compared to Graphene's version as it achieves the same performance with significantly less memory throughput. However, since all kernels are already compute-bound, these differences do not affect the overall performance in this case.

Figure 10 shows Graphene's performance compared to cuBLASLt for GEMM kernels with fused pointwise operations, like addition

[1]We use M=N=5120,K=2048 on Volta and M=N=5376,K=2048 on Ampere. Both use a thread-block/SM tile of size M=N=128,K=32

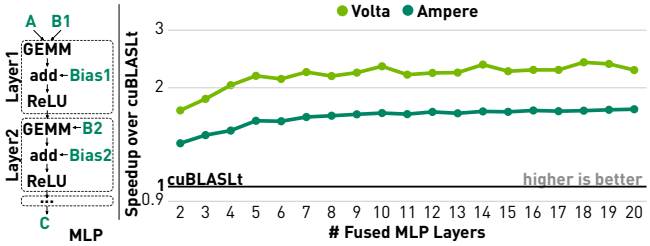


Figure 11: Fusing multiple MLP-Layers (GEMM + bias-add + ReLU activation) into a single kernel compared against equivalent cuBLASLt implementations.

of a bias tensor or applying the Rectified Linear Unit (ReLU) activation function. cuBLASLt provides fused kernels for these tensor computations. Again, Graphene's generated kernels exactly match the performance of the highly tuned library implementations on both architectures which confirms our first hypothesis.

Hypothesis B: Graphene generates competitive *fused* kernels for important deep learning tensor computations. *We find that Graphene matches or outperforms the state-of-the-art by more than 2× for some of the most important deep learning tensor computations including Multi-Layer Perceptron (MLP), Layernorm and Fused Multi-head Attention (FMHA).*

In this experiment, we analyze the performance of Graphene-generated kernels for fused tensor computations beyond single operators or basic pointwise fusions. To the best of our knowledge, we compare Graphene-generated kernels against the fastest known reference implementations targeting GPUs for the individual tensor computations.

Multi-Layer Perceptron (MLP). Figure 11 shows Graphene's performance for MLP, a classic but still relevant computation that regularly occurs in today's deep learning models like Transformers. We compare Graphene to cuBLASLt which provides implementations for a single MLP layer (GEMM + bias addition + pointwise activation) whose single-layer performance was as already shown in Figure 10 "bias+relu". For specific problem sizes[2], it is possible to fuse multiple MLP layers into a single kernel. In these cases, all intermediate tensors fit into the GPU's shared memory allowing to avoid communication via the slower global memory. Graphene's kernel implements this fusion and we compare it against the cumulative performance of cuBLASLt invocations when computing MLP with up to 20 layers.

Figure 11 shows that Graphene outperforms cuBLASLt by up to 2.39x. These results show that a) Graphene is capable of representing tensor computations beyond single-operators like GEMM, and b) that fused kernels should be preferred over cumulative library invocations (which often is the default lowering in deep learning compilers) if problem sizes permit.

Long Short-Term Memory (LSTM). Figure 12 shows Graphene's performance for a simplified LSTM cell. We compute two independent GEMMs followed by an addition and two more pointwise

[2]For example N=K<=128 with arbitrary M

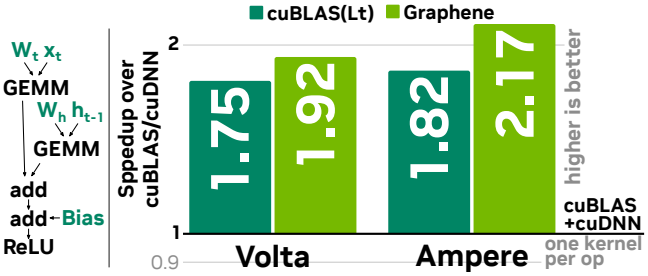


Figure 12: Comparing Graphene's fused kernels to CUDA library implementations of a simplified LSTM-cell tensor computation.

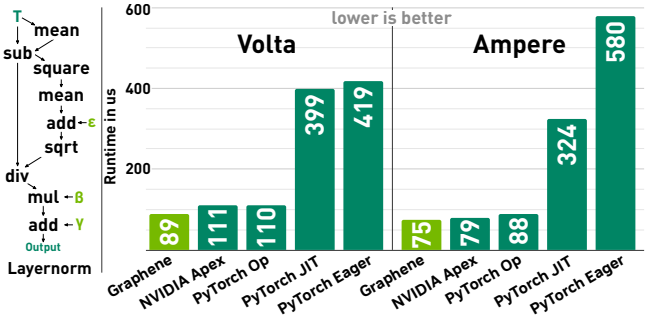


Figure 13: Comparing Graphene's Layernorm performance to multiple PyTorch reference implementations.

operations. This pattern is the foundation of the computation occurring in LSTM-cells. Typically, LSTM-cells use *tanh* as activation functions. However, to be able to compare against CUDA library implementations, we must use a slightly altered version of the LSTM-cell, using ReLU instead of tanh as the activation function, because cuBLASLt does not provide kernels with tanh.

There exist two ways of implementing this computation using CUDA libraries: 1) Use one library kernel per node in the graph (5 kernels in total) using cuBLAS and cuDNN. This lowering strategy is common in many deep learning compilers and we use this version as our baseline. 2) A more optimized version requires only two library kernels: Using cuBLASLt, the second GEMM can accumulate into the output of the first GEMM (implementing the subsequent add-node) and additionally perform the bias addition and the activation.

Graphene's kernel fuses all nodes into a single kernel and therefore again avoids round-trips to global memory for computing intermediate results. Fusing computations into a single kernel like this results in significant speedups (1.75 on Volta and 1.82 on Ampere) compared to the unfused baseline. This additional fusion is beyond the capabilities of today's libraries and explains our speedup.

Layernorm. Figure 13 shows Graphene's Layernorm performance. Layernorm is of particular interest because it is widely used in deep learning models such as Transformers and it does not perform any GEMM computations but instead consists only of a combination of pointwise and reduction computations. In this experiment, we compare Graphene-generated kernels to state-of-the-art fused and unfused PyTorch implementations. Specifically, *PyTorch JIT* and

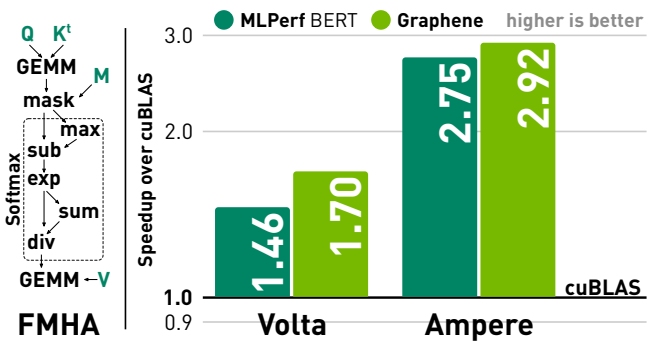


Figure 14: Comparing Graphene's FMHA performance to NVIDIAs handwritten fused kernels used for MLPerf BERT inference and an unfused cuBLAS-based baseline.

Eager show the PyTorch performance using the default eager execution and Torchscript-based JIT compilation. We also measure the performance of two fused kernels: 1) the built-in fused Layernorm operator which lowers to a pre-defined CUDA kernel shipped with PyTorch and 2) NVIDIA Apex which is a PyTorch extension providing alternative high-performance fused kernels for important computations including Layernorm.

Again, we see that Graphene matches the performance of the best known implementation for this particular tensor computation. This shows that our IR is capable of representing high-performance tensor computations beyond GEMMs.

Fused Multi-Head Attention (FMHA). Multi-Head attention is the core computational pattern of Transformer models and consists of two back-to-back GEMMs with a softmax computation in between. The softmax computation itself consists of two reductions and several pointwise operations.

Due to being the most compute intensive part of the Transformer architecture, there exist manually fused kernels for accelerating this computation. Figure 14 shows Graphene's FMHA performance. Our baseline is the cumulative exeuction time for two cuBLAS GEMM invocations and a custom softmax CUDA kernel with a straightforward implementation. Graphene's FMHA kernel is specialized for the problem sizes occurring in MLPerf BERT inference (16 heads, batch-size of 32, hidden size 64, and sequence length 384). We implement a similar fusion strategy compared to NVIDIA's fused FMHA kernels (in TensorRT) that are used in their MLPerf inference submission for BERT [6]. Graphene is capable of generating state-of-the-art fused kernels for optimized tensor computations and we even achieve a small speedup over the MLPerf kernels due to optimized shared memory layouts.

So far, we evaluated our kernels for tensor computations in isolation. To evaluate whether Graphene-generated kernels are useful in practice, we additionally injected our FMHA kernels as custom operators into multiple Huggingface Transformer networks. Figure 15 shows Graphene performance for five different networks of the Transformer family. We report the speedup we achieve when using the custom FMHA kernel compared to the regular PyTorch inference performance and see improvements of up to 59%.

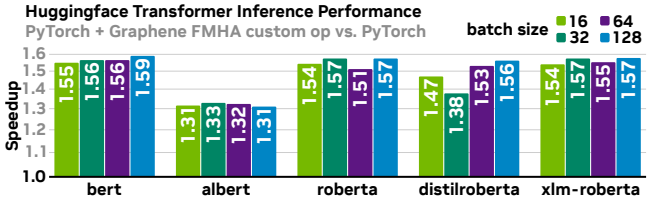


Figure 15: Injecting Graphene-generated Ampere FMHA kernels into Huggingface Transformer networks achieves up to 59% speedup. The speedup correlates with the fraction of FMHA occurences per network.

7 RELATED WORK

Graphene is significantly inspired by Fireiron [9]. Fireiron introduced a GEMM-specific scheduling language for expressing optimized matrix multiplications for GPUs. It also introduced the concept of specifications for representing operations on matrices that we extend in Graphene. Fireiron's specs only cover matrix multiplication and data movements whereas Graphene's set of specs is capable of representing all kinds of GPU tensor computations, far beyond single GEMMs. Furthermore, Fireiron can only handle 2D matrices whereas Graphene works with multi-dimensional tensors.

Threads as tensors. A key novelty of Graphene is the explicit representation of the GPU compute hierarchy as a *decomposable* tensor. This is inspired by frameworks for distributed deep learning which typically enable the specification of a multi-dimensional mesh of devices. This mesh is used for mapping data-dimensions to parallel execution units (devices, not threads in this case), similarly to how we map data tiles to thread tiles. Popular examples for such frameworks are Mesh TensorFlow [21], GShard [12], GSPMD [31], and P^2 [30]. In contrast to Graphene, none of the existing frameworks enables the specification of hierarchical meshes/threads. This hierarchy is crucial when it comes to expressing optimized computations for GPUs whose processing elements are inherently hierarchical (e.g., grids of blocks of warps of threads).

IRs for tensor computations. The most closely related IRs for representing optimized tensor computations are TensorIR [7] used in TVM [5], MLIR [11, 28], and AMOS [32]. Each of these has a concept that is similar to Graphene's atomic specification for representing executable tensor instructions on GPUs. TensorIR uses pre-defined *Tensor Intrinsics*, which, like Graphene, describe *what and how* a tensor instruction computes on which in- and output tensors. MLIR provides a low-level GPU dialect with pre-defined *ops* that map to NVVM's WMMA tensor core instructions. AMOS represents available GPU instruction as *Compute and Memory Abstractions* to which software is automatically mapped. However, none of these approaches explicitly takes the executing threads per instruction into account (e.g., the fact that Volta `mma` instructions are executed by quad-pairs) which is crucial for a precise representation of optimized kernels.

Marvel [4] and MAESTRO [10] enable mapping tiled computations to spatial accelerators, however, they do not expose enough control to express low-level highly optimized GPU kernels.

Code generation for tensor computations. TVM [5] (using the *tensorize* primitive), Diesel [3] and UNIT [29] are compilers capable of automatically generating high-performance tensor computations targeting GPUs. UNIT is built on top of TVM and therefore both ultimately rely on LLVM IR for representing optimized implementations. Diesel is a polyhedral compiler generating CUDA C++ code with inline PTX assembly. Thus, neither of these compilers overcomes the challenges identified in this paper of representing optimized GPU tensor code in CUDA C++ or lower-level IRs.

Lift [23], Rise [2], and Triton [5] are intermediate languages for representing high-performance GPU code including tensor computations. However, Lift can not generate code that uses Tensor Core instructions and is therefore unable to deliver the best possible tensor performance on modern GPUs. Rise recently added limited support for Tensor Cores [22] but still lacks sufficient control for tensorized data movements using instructions like `ldmatrix`.

Triton exposes a high-level Python DSL which is transformed into a custom IR that is eventually lowered into high-performance LLVM code using tensor core instructions. However, Triton's high-level DSL and IR intentionally abstract the complexity of GPU tensor instructions and only introduce them during LLVM code generation. This leads to highly complex compiler transformation passes whose extension requires both initimate knowledge of the targeted GPU architectures as well as the compiler implementation itself. In contrast, Graphene aims to explicity express optimized computations on the IR level and uses a straight-forward code generation.

8 CONCLUSION

In this paper, we introduced Graphene, an IR for optimized GPU tensor computations. Graphene predominantly solves the representation problem: Highly optimized GPU kernels must be written using CUDA/PTX which is not a suitable IR for tensor computations. By introducing an IR closer to the domain of tensor computations we provide an alternative target language for machine learning compilers and performance experts.

Graphene represents both data and threads as first-class decomposable tensors. Graphene's specs are the unifying concept for representing computations and data movements ranging from kernels to executable instructions. High-performance GPU code is represented by decomposing kernel-level computations into thread-level executable instructions. Fast tensor instructions are exposed as atomic specs and we emit CUDA C++ code using inline PTX assembly during code generation.

Graphene achieves competetive performance to manually tuned library implementations and is thus capable of representing the fastest known kernels today. Furthermore, Graphene is capable of representing fused computations for which no library routines exist yet and matches or outperforms manually developed kernels. Our generated kernels achieve significant speedups when they are deployed in real-world deep learning networks. Graphene therefore provides the foundation for novel ML compiler research including systematically deriving optimized tensor computations as well as generating high-performance architecture-specific GPU kernels.

ACKNOWLEDGMENTS

We thank Young-Jun Ko for sharing his insights and explaining how to implement fast Fused Multi-Head Attention (FMHA) kernels. We thank Andrew Liu for his feedback on early prototypes of the Graphene implementation. We thank Girish Bharambe and Duane Merrill for helping us with all kinds of low-level performance-related issues. We also thank Alberto Magni and Vijay Thakkar for their valuable feedback on early drafts of the paper and figures.

REFERENCES

[1] NVIDIA GTC Fall 2022. 2022. CUTLASS: Python API, Enhancements, and CUTLASS 3.0 Preview (Announcing the CuTe programming model). https://static.rainfocus.com/nvidia/gtcfall2022/sess/1655735950588001cX98/supmat/A41131%20-%20CUTLASS_%20Python%20API%2C%20Enhancements%2C%20and%20NVIDIA%20Hopper_1663707717455001eoiD.pdf

[2] Robert Atkey, Michel Steuwer, Sam Lindley, and Christophe Dubach. 2017. Strategy Preserving Compilation for Parallel Functional Code. https://doi.org/10.48550/ARXIV.1710.08332

[3] Somashekaracharya G. Bhaskaracharya, Julien Demouth, and Vinod Grover. 2020. Automatic Kernel Generation for Volta Tensor Cores. *CoRR* abs/2006.12645 (2020). arXiv:2006.12645 https://arxiv.org/abs/2006.12645

[4] Prasanth Chatarasi, Hyoukjun Kwon, Angshuman Parashar, Michael Pellauer, Tushar Krishna, and Vivek Sarkar. 2022. Marvel: A Data-Centric Approach for Mapping Deep Learning Operators on Spatial Accelerators. *ACM Trans. Archit. Code Optim.* 19, 1 (2022), 6:1–6:26. https://doi.org/10.1145/3485137

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/v1/n19-1423

[7] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. 2022. TensorIR: An Abstraction for Automatic Tensorized Program Optimization. https://doi.org/10.48550/ARXIV.2207.04296

[8] Google. 2017. TensorFlow XLA. https://www.tensorflow.org/xla

[9] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodík, and Vinod Grover. 2020. Fireiron: A Data-Movement-Aware Scheduling Language for GPUs. In *PACT '20: International Conference on Parallel Architectures and Compilation Techniques, Virtual Event, GA, USA, October 3-7, 2020*, Vivek Sarkar and Hyesoon Kim (Eds.). ACM, 71–82. https://doi.org/10.1145/3410463.3414632

[10] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. 2020. MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings. *IEEE Micro* 40, 3 (2020), 20–29. https://doi.org/10.1109/MM.2020.2985963

[11] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[12] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2021. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=qrwe7XHTmYb

[13] MLIR. 2022. 'gpu' Dialect. https://mlir.llvm.org/docs/Dialects/GPU/

[14] NVIDIA. 2017. Volta Architecture Whitepaper. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[15] NVIDIA. 2018. Turing Architecture Whitepaper. https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[16] NVIDIA. 2021. Ampere Architecture Whitepaper. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf

[17] NVIDIA. 2022. CUTLASS - CuTe documentation. https://github.com/NVIDIA/cutlass/tree/master/media/docs/cute

[18] NVIDIA. 2022. CUTLASS - CuTe Predication. https://github.com/NVIDIA/cutlass/blob/master/media/docs/cute/0y_predication.md

[19] NVIDIA. 2022. PTX ISA. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[20] NVIDIA. 2022. PTX ISA - SM70 mma-884-f16. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-matrix-fragment-mma-884-f16

[21] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake A. Hechtman. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 10435–10444. https://proceedings.neurips.cc/paper/2018/hash/3a37abdeefe1dab1b30f7c5c7e581b93-Abstract.html

[22] Lukas Siefke, Bastian Köpcke, Sergei Gorlatch, and Michel Steuwer. 2022. Systematically extending a high-level code generator with support for tensor cores. In *GPGPU@PPoPP 2022: Proceedings of the 14th Workshop on General Purpose Processing Using GPU, Virtual Event, Seoul, Republic of Korea, 3 April 2022*, Yifan Sun, Daniel Wong, and Hoda Naghibijouybari (Eds.). ACM, 3:1–3:6. https://doi.org/10.1145/3530390.3532733

[23] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*. 74–85.

[24] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. 2023. *CUTLASS*. https://github.com/NVIDIA/cutlass

[25] Philippe Tillet, Hsiang-Tsung Kung, and David D. Cox. 2019. 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@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, Tim Mattson, Abdullah Muzahid, and Armando Solar-Lezama (Eds.). ACM, 10–19. https://doi.org/10.1145/3315508.3329973

[26] OpenAI Triton. 2022. Triton Instruction Selection Transformation Pass. https://github.com/openai/triton/blob/master/lib/codegen/selection/generator.cc

[27] Apache TVM. 2022. Use Tensorize to Leverage Hardware Intrinsics. https://tvm.apache.org/docs/how_to/work_with_schedules/tensorize.html

[28] 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. 2022. Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction. *CoRR* abs/2202.03293 (2022). arXiv:2202.03293 https://arxiv.org/abs/2202.03293

[29] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. 2021. UNIT: Unifying Tensorized Instruction Compilation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 77–89. https://doi.org/10.1109/CGO51591.2021.9370330

[30] Ningning Xie, Tamara Norman, Dominik Grewe, and Dimitrios Vytiniotis. 2022. Synthesizing Optimal Parallelism Placement and Reduction Strategies on Hierarchical Systems for Deep Learning. In *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*, Diana Marculescu, Yuejie Chi, and Carole-Jean Wu (Eds.). mlsys.org. https://proceedings.mlsys.org/paper/2022/hash/b73ce398c39f506af761d2277d853a92-Abstract.html

[31] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake A. Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. 2021. GSPMD: General and Scalable Parallelization for ML Computation Graphs. *CoRR* abs/2105.04663 (2021). arXiv:2105.04663 https://arxiv.org/abs/2105.04663

[32] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. 2022. AMOS: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang (Eds.). ACM, 874–887. https://doi.org/10.1145/3470496.3527440

Received 2022-10-20; accepted 2023-01-19