Flexible Performant Tensor Contractions on GPUs

Thomas Faingnaert, Ward Vermeulen, Tim Besard, Bjorn De Sutter, *Member, IEEE*

***Abstract*—Tensor contractions extend the concept of the General Matrix Multiplication (GEMM) to high-dimensional spaces. They enable sophisticated computations in various scientific disciplines. Graphics Processing Units (GPUs) are commonly used to accelerate tensor contraction algorithms due to their inherent parallelisability. NVIDIA's cuTENSOR stands as a state-of-the-art library for GPU-based tensor contractions. However, its lack of flexibility limits researchers in tailoring contraction kernels to their specific research needs. This paper presents a novel and flexible implementation of the GEMM-like Tensor Tensor (GETT) multiplication algorithm for tensor contractions in Julia. By repurposing and adapting components of GemmKernels.jl, a versatile library offering customisable and high-performance GEMM kernels for CUDA-enabled GPUs, we construct GEMM-like kernels that cater to the unique requirements of tensor contractions. Despite being entirely written in high-level Julia code and not yet exploiting a range of modern CUDA hardware features, the average performance of our library on standard tensor contractions compares favourably to cuTENSOR's hand-optimised implementations, with outliers in both directions (faster and slower). When flexibility is needed, e.g. to fuse arbitrary elementwise operations into kernels, our library performs up to an order of magnitude faster than cuTENSOR, even on recent, data centre-grade devices such as the RTX 6000 Ada.**

***Index Terms*—tensor contraction, graphics processors, high-level programming languages.**

I. Introduction

Tensor contractions (TCs) have become a fundamental computational primitive in scientific applications ranging from deep neural network training to quantum circuit simulation and computational chemistry. TCs generalise matrix multiplication to higher-dimensional arrays.

Offering massive parallelism and memory bandwidth, Graphics Processing Units (GPUs) are the dominant accelerator architecture for tensor-intensive workloads. However, exploiting their full potential remains a complex task because of several reasons, including the curse of dimensionality inherent in high-order tensors and the tight coupling between algorithmic choices and hardware constraints such as memory hierarchy, thread scheduling, and data movement.

GPU tensor libraries such as cuBLAS [1], cuDNN [2], and cuTENSOR [3] achieve high performance through aggressive specialisation and hardware-specific optimisation. However, this performance comes at the cost of flexibility: these hand-tuned libraries typically support only a fixed set of operations with specific constraints on tensor properties. For instance, cuBLAS optimises for dense matrix operations with standard data types (FP16, FP32, FP64), while cuDNN focuses on a fixed set of deep learning primitives with particular layout requirements. Beyond data type and layout constraints, the libraries offer limited support to fuse elementwise operations into the TC kernels. While some libraries include TCs with some standard, commonly used elementwise operations fused into them, most are unable to fuse arbitrary elementwise operations. When users need non-fused elementwise operations, this leads to excessive memory traffic because intermediate results must be materialised between kernel calls.

For performing operations outside the libraries' boundaries, researchers face a choice: accept orders of magnitude slower performance using generic implementations, or invest substantial effort in custom kernel development. The latter requires those researchers to develop the necessary expertise in GPU architectures and programming models, on top of their domain knowledge, thus putting a drag on their scientific progress.

With our research, we aim to make the full power of GPU acceleration available to programmers without sacrificing flexibility, and without needing them to become GPU experts. To do so, we leverage the capabilities of the scientific programming language Julia. As has been done for other computational primitives, such as general matrix multiplication (GEMMs) [4], we build on Julia's unique features to compile high-level, abstract API usage into performant code. Most importantly, our API design and the underlying implementation allow arbitrary elementwise operations to be fused into the TC kernels, thus avoiding the need to launch additional kernels and incur the cost of their corresponding memory traffic.

The main result of our research is an extension to the Julia package GemmKernels.jl that, despite being entirely written in high-level Julia code and not yet exploiting a range of modern CUDA hardware features, achieves TC performance that is on average better than cuTENSOR's hand-optimised implementations on several generations of CUDA devices, with outlier TCs in both directions (faster and slower). When flexibility is needed, e.g. to perform elementwise operations for which cuTENSOR does not include fused kernels, our library performs up to an order of magnitude faster than cuTENSOR, even on recent, data centre-grade devices such as the RTX 6000 Ada. Our contributions with this paper are:

- The design of our TC extension to GemmKernels.jl, including non-TC-specific adaptations that improve the base GEMM performance of GemmKernels.jl.
- A performance evaluation of our kernels across different GPU architectures, ranging from consumer-grade to data centre-grade hardware, including a performance benchmark against NVIDIA's cuTENSOR library and the impact of operator fusion on performance.
- An in-depth analysis of performance patterns across different GPU architectures and TCs, identifying architec-

T. Faingnaert and B. De Sutter are with the Department of Electronics and Information Systems, Ghent University, Belgium.
W. Vermeulen works at TechWolf.
T. Besard works for JuliaHub.
Manuscript received MMM DD, YYYY; revised MMM DD, YYYY.

ture-specific optimisation opportunities and performance bottlenecks to inform future development strategies.
- An analysis of the impact of the compiler on programmer and end-user productivity when developing or using flexible libraries in high-level programming languages, and suggestions to alleviate potential issues.

The remainder of this paper is structured as follows. Section II provides background on tensors, TCs, GPGPU programming, Tensor Cores, Julia, and the GemmKernels package on which our implementation builds. Section III presents our requirements for a flexible TC API and details the components necessary to implement it in Julia, including extensions to GemmKernels, methods for parameter optimisation, and illustrative usage examples. Section IV evaluates and analyses the performance of our approach. Section V examines the influence of compiler technology on programmer and library user productivity in the context of flexible computational libraries. Section VI surveys related work, while Section VII describes our artefacts' availability. Finally, Section VIII concludes with a summary of our contributions and directions for future work.

II. Background

A. Tensors

Tensors are multidimensional arrays, i.e. the generalisation of one-dimensional vectors and two-dimensional matrices. The *dimensionality* of a tensor $\mathcal{A}$ is its number of dimensions, and will be written as $d_{\mathcal{A}}$. Each element in a $d_{\mathcal{A}}$-dimensional tensor $\mathcal{A}$ is uniquely addressed by its *index*, which is represented as a $d_{\mathcal{A}}$-tuple $(i_1, i_2, \ldots, i_{d_{\mathcal{A}}})$. The set of all indices of a tensor $\mathcal{A}$ is called its *index set* $I_{\mathcal{A}} = \{i_1, i_2, \ldots, i_{d_{\mathcal{A}}}\}$. Each i_n is a natural number from 1 to $E_n^{\mathcal{A}}$, inclusive, where $E_n^{\mathcal{A}}$ is called the *extent* of the nth dimension of $\mathcal{A}$.

Tensors can be stored in many different formats. In this paper, we use Julia's convention of a column-major format, where the storage order in memory corresponds to the order of indices. The *stride* of the nth dimension in tensor $\mathcal{A}$, denoted $S_n^{\mathcal{A}}$, is the number of memory locations in between two elements that are contiguous in that dimension. Hence, for a column major layout, $S_1^{\mathcal{A}} = 1$, and $S_n^{\mathcal{A}} = \prod_{m=1}^{n-1} E_m^{\mathcal{A}}$.

Tensor transpositions permute the order of indices in a tensor, similar to matrix transpositions. They do not change a tensor's dimensionality, they only reorder its elements.

Reshaping operations like flattening and folding do change the dimensionality. *Flattening* reduces it by re-interpreting two or more dimensions of extents $E_i^{\mathcal{A}}, E_{i+1}^{\mathcal{A}}, \ldots, E_{i+k}^{\mathcal{A}}$ as a single dimension of extent $\prod_{m=i}^{i+k} E_m^{\mathcal{A}}$. *Folding* does the inverse, and increases the dimensionality by splitting a dimension into two or more consecutive dimensions. Both flattening and folding are purely logical operations, and do not change the in-memory representation of the tensor.

Finally, *tensor contraction* (TC) is the multidimensional analogon of matrix multiplication. The general form of a binary TC of the tensors $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ is shown in Formula (1), in which we introduce three index sets I_m, I_n, and I_k. The set I_m is the set of indices that is common to both $\mathcal{A}$ and $\mathcal{C}$ (but not $\mathcal{B}$), and is called the *set of free indices of* $\mathcal{A}$. Analogously, I_n is the set of indices common to $\mathcal{B}$ and $\mathcal{C}$ (but not $\mathcal{A}$), and is referred to as the *set of free indices of* $\mathcal{B}$. Finally, the set I_k is the set of indices that occur in both $\mathcal{A}$ and $\mathcal{B}$, but not $\mathcal{C}$. It is the set of indices that is summed over, and is hence also called the *set of contracted indices*. $\Pi^{\mathcal{A}}$, $\Pi^{\mathcal{B}}$, and $\Pi^{\mathcal{C}}$ are permutations that determine the order of indices for $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, respectively.

$$\mathcal{C}_{\Pi^{\mathcal{C}}(I_m \cup I_n)} \leftarrow \sum_{k_1=1}^{E_{k_1}} \cdots \sum_{k_K=1}^{E_{k_K}} \alpha \cdot \mathcal{A}_{\Pi^{\mathcal{A}}(I_m \cup I_k)} \cdot \mathcal{B}_{\Pi^{\mathcal{B}}(I_k \cup I_n)} + \beta \cdot \mathcal{C}_{\Pi^{\mathcal{C}}(I_m \cup I_n)} \tag{1}$$

In the remainder, we will make two assumptions regarding the index sets I_m, I_n, and I_k, in line with the literature [5], [6]. First, we assume that these sets form a partition of the union of the index sets of the input tensors, $I_{\mathcal{A}} \cup I_{\mathcal{B}} \cup I_{\mathcal{C}}$. Hence, we do not consider the case where one index occurs in all of $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, as is the case for batched TCs [7].

Secondly, we assume that all three index sets are non-empty. This means that the TC can be mapped to a matrix multiplication (GEMM), as opposed to a lower-level BLAS primitive like a matrix-vector (GEMV) or vector-vector product (DOT or GER). The latter type of TCs exhibit a lower ratio of floating point operations to memory accesses, and thus offer fewer opportunities for amortisation of memory accesses [8]. For these TCs, re-using the lower-level BLAS kernels typically suffices, as they do not require GEMM-like optimisations like tiling or vectorisation to achieve peak performance [5], [8].

A concrete example of a TC is displayed in Formula (2). In this example, $\alpha = \beta = 1$, $I_m = \{m_1, m_2\}$, $I_n = \{n_1, n_2\}$, $I_k = \{k_1, k_2\}$, $\Pi^{\mathcal{A}} = (m_1, k_1, m_2, k_2)$, $\Pi^{\mathcal{B}} = (n_2, k_2, n_1, k_1)$, and $\Pi^{\mathcal{C}} = (m_1, n_1, n_2, m_2)$.

$$\mathcal{C}_{m_1 n_1 n_2 m_2} \leftarrow \sum_{k_1=1}^{E_{k_1}} \sum_{k_2=1}^{E_{k_2}} \mathcal{A}_{m_1 k_1 m_2 k_2} \cdot \mathcal{B}_{n_2 k_2 n_1 k_1} + \mathcal{C}_{m_1 n_1 n_2 m_2} \tag{2}$$

To express TCs succinctly, we use the Einstein summation convention in the remainder of the paper. In this notation, the sums across the set of contracted indices I_k are dropped, and are assumed to be implicit. Formula (2) thus becomes Formula (3) in the Einstein summation convention.

$$\mathcal{C}_{m_1 n_1 n_2 m_2} \leftarrow \mathcal{A}_{m_1 k_1 m_2 k_2} \cdot \mathcal{B}_{n_2 k_2 n_1 k_1} + \mathcal{C}_{m_1 n_1 n_2 m_2} \tag{3}$$

Matrix multiplication is a special-case TC. Indeed, by substituting $I_m = \{m\}$, $I_n = \{n\}$, $I_k = \{k\}$, $\Pi^{\mathcal{A}} = (m, k)$, $\Pi^{\mathcal{B}} = (k, n)$, and $\Pi^{\mathcal{C}} = (m, n)$ in Formula (1), one obtains the matrix multiplication $C_{mn} \leftarrow \alpha \cdot A_{mk} \cdot B_{kn} + \beta \cdot C_{mn}$ in Einstein notation. Other versions of general matrix multiplication that occur in BLAS-libraries, where either one or both of the input matrices A and B are transposed, are obtained by changing the permutations $\Pi^{\mathcal{A}}$ and $\Pi^{\mathcal{B}}$.

B. Tensor Contraction Algorithms

Approaches for performant TCs have been classified in three categories: loop nesting, loop-over-GEMM, and TTGT [5].

Loop nesting converts TCs into loop nests [9], [10], [11] on which optimisations are applied like loop reordering, fusion,

and vectorisation. While effective for small, cache-fitting TCs, it suffers from poor memory access patterns for larger ones.

Loop-over-GEMM slices tensors into 2D matrices and performs an efficient GEMM kernel on every identified 2D slice [8], [12]. Performance degrades when slices are small or memory accesses are highly strided.

TTGT (Transpose-Transpose-GEMM-Transpose), used by frameworks like Cyclops [13], Tensor Toolbox [14], Tensorlab [15], and libtensor [16], calls GEMM only once in between three transpositions that reorder tensor dimensions. Tensors are first transposed so contracted indices align, flattened into matrices, multiplied via GEMM, of which the result is then folded and transposed back into the resultant tensor. While effective for compute-bound TCs, TTGT requires extra kernel launches and additional memory [5], and may produce highly rectangular matrices that underperform in GEMM libraries [6].

In 2018, GETT (GEMM-like Tensor-Tensor Contraction) emerged as a fourth method that avoids explicit transposes by reorganising dimensions during loading—essentially fusing transposes into the GEMM kernel itself [5]. It has since been adopted by others [6], [17]. GETT requires flexible GEMM implementations but eliminates TTGT's overhead.

C. GPGPU Programming

This paper focuses on NVIDIA's CUDA platform for GPUs [18]. Other GPGPU frameworks have similar concepts, albeit with sometimes different terms.

CUDA employs massive parallelism where a kernel function executes across thousands of parallel threads, executed on the following execution hierarchy:

- *Threads*: Smallest execution unit.
- *Warps*: Hardware groups of 32 threads executing the same instruction simultaneously (SIMT model).
- *Thread blocks*: Programmer-defined groups executed on the same Streaming Multiprocessor (SM) with efficient communication/synchronisation.
- *Grid*: All blocks running the same kernel.

Data is stored in a memory hierarchy with matching levels:

- *Registers*: Per-thread, fastest but smallest.
- *Shared memory*: Per-block, divided into banks for parallel access; bank conflicts cause serialisation.
- *Global memory*: Device-wide access, highest capacity but highest latency.
- *Caches*: L1 (per-SM) and L2 (device-wide) buffer global memory access.

Achieving maximum performance requires 16-byte memory transactions per thread. Smaller data types achieve this through vectorisation—loading multiple contiguous elements per instruction (e.g., 8 16-bit floats at once).

D. Tensor Cores

Modern NVIDIA GPUs include Tensor Cores——hardware accelerators for matrix multiplication that operate in mixed precision (lower-precision inputs, higher-precision accumulation/output). They can be programmed with the WMMA API that provides portable, warp-wide operations where 32 threads cooperatively compute matrix products. Alternatively, cuBLAS, cuBLASLt, cuDNN, cuTENSOR, CUTLASS and other vendor-provided libraries provide pre-optimised kernels.

Since 2017 each GPU generation adds new data types, matrix shapes, and optimisation opportunities, that are then used in the vendor-provided libraries.

As WMMA did not allow reaching optimal performance, NVIDIA added `mma.sync`, a lower-level instruction executed per 8 threads with smaller matrices, enabling finer control and optimisations like 16-byte memory accesses and bank conflict elimination through custom swizzled memory layouts [19]. Later GPUs accumulated more `mma.sync` variants with different data types, shapes, and thread-to-element mappings.

In 2018, Turing's second-generation Tensor Cores introduced a new instruction, `ldmatrix`, which loads matrix elements from shared memory cooperatively by all threads in a warp [20]. To improve performance, each thread loads 16 bytes, which are then broadcast to four other threads to match the data distribution of the `mma.sync` instruction.

The Ampere architecture (2020) introduced asynchronous global-to-shared memory copies without intermediate registers, reducing data movement and register pressure [20]. This enables deep software pipelines to hide memory latency, complementing Ampere's increased shared memory capacity.

Major changes of the Hopper architecture (2022) included Tensor Cores operating across 4-warp groups and reading directly from shared memory [21]; a new execution hierarchy level called thread block clusters that enables concurrent SM scheduling and distributed shared memory for SM-to-SM communication; Tensor Memory Accelerator (TMA) hardware units for offloading address arithmetic and data movement from SIMT cores, supporting multicast, swizzled layouts, and automatic zero-padding; and asynchronous Tensor Core inputs that enable warp specialisation, where producer warps load data via TMA, and consumer warps perform computation.

Ada Lovelace, the consumer variant of the data centre-focused Hopper architecture, was launched a month after the latter. It does not support Hopper architectural features such as asynchronicity, thread block clusters, or TMA, but does allow for optimisations specific to Ampere or earlier [22].

In 2024, Blackwell introduced a new word-addressed Tensor Memory requiring explicit allocation [23]. Tensor Core operations also expanded to two-thread-block-wide with asynchronous inputs/outputs. In addition, decoupled epilogue operations enable three-way warp specialisation (loading, computing, post-processing), and enhanced scheduling options including preferred clusters and persistent scheduling.

E. Julia Programming Language

Julia (2012) is a high-level programming language with modern features (interactivity, package manager, dynamic parametric types) that achieves C-like performance through compilation rather than interpretation [24].

Julia features *multiple dispatch*: Upon a run-time function invocation, one function out of a set of overloaded functions is selected based on the dynamic types of all of the run-time arguments. This contrasts with, e.g., C++'s single-dispatch

virtual functions that are selected solely on the type of the implicit `this` pointer argument.

Julia combines ahead-of-time performance with just-in-time flexibility by performing type inference for all expressions to reduce the run-time uncertainty commonly associated with dynamic typing systems, and by compiling specialising functions per argument types (devirtualising calls and removing type checks) for which efficient machine code is then generated.

Its compilation pipeline first parses source code to an Abstract Syntax Tree (AST); lowers that to untyped Julia IR; performs type inference to convert that to typed Julia IR from which LLVM IR is then generated, after which LLVM [25] is leveraged for code optimisation and machine code generation.

Julia can also be used to program accelerators. Through the CUDA.jl package, one can write CUDA kernels completely in Julia [26]. It reuses Julia's compilation pipeline up until the code generation to LLVM IR. That IR is intercepted and passed to LLVM's NVPTX backend, which generates PTX instructions, a portable virtual instruction set architecture used by NVIDIA GPUs. Before execution, this PTX is then converted by the black-box ptxas compiler to SASS instructions, the actual underlying instruction set of the GPU.

This leverages existing LLVM infrastructure while maintaining Julia's high-level expressiveness.

F. GemmKernels

GemmKernels is a native Julia library for flexible and performant GEMM kernels by Faingnaert et al. [4]. It is focused on NVIDIA GPUs, and supports Tensor Cores to accelerate computations. It exploits the data reuse inherent in GEMM by using recursive blocking techniques, which copy tiles of the input matrices to progressively faster levels of the memory hierarchy. First, a tile of the input matrices A and B is copied cooperatively from global to shared memory by all threads in a thread block. Then, this tile is subdivided in smaller tiles, which are each loaded cooperatively into registers by all threads in the same warp. Each warp then computes a matrix product, e.g. using WMMA, and stores the result back to shared memory. Finally, the result is copied back to global memory by all threads in a thread block.

As with TCs, so too is there a need for flexibility in GEMM kernels, e.g. to support different data types, memory layouts, elementwise operations, etc. GemmKernels achieves this flexibility by providing a set of template kernels. These template kernels consist of orthogonal components that together form a full GEMM kernel. Each of the 5 components has its own responsibility, and allows for a different kind of flexibility. *Params* determine the sizes of the tiles for each step in the GEMM, and the GPU kernel's launch configuration such as the amount of thread blocks and warps. *Layouts* can be used to customise the memory layout and data type of the input or output matrices in both global and shared memory. *Transforms* are functors that are used to apply elementwise operations to the input and output matrices, such as scaling or activation functions. The *Operator* component is used to customise the computation performed by each warp, and can be utilised for custom data types or arithmetic (such as complex or tropical numbers). Finally, the *Epilogue* component allows for customisation of post-processing, such as the addition of a bias-vector addition after the GEMM.

With Julia's multiple dispatch paradigm, it is possible to define fine-grained methods that modify the functionality of each component, allowing users to tailor the library to their specific needs. For example, to customise the memory layout of the input and output matrices, one can define a custom layout data type `CustomLayout`, and then define custom implementations of the `load` and `store!` functions. This new layout component can then be combined seamlessly with existing components in GemmKernels to form a new, specialised GEMM kernel. Julia's just-ahead-of-time compilation and specialisation allows to perform this composition completely at compile time, without any run time overhead.

III. Flexible Tensor Contractions

A. Requirements

NVIDIA's popular cuTENSOR library [3] features highly optimised kernels based on both TTGT and GETT. While cuTENSOR has support for different data types and elementwise operations, its closed-source nature limits its flexibility. This is problematic when a use case does not map exactly to one of its kernels. For example, elementwise operations such as activation functions are ideally fused in the TC kernel, but cuTENSOR only supports a limited set of elementwise operations on tensor inputs and outputs [27]. To use commonly-used, but unsupported activation functions such as the Exponential Linear Unit (ELU) [28] and Leaky Rectified Linear Unit (Leaky ReLU) [29], or to experiment with novel activation functions, one needs extra kernels. This increases memory traffic (to load input tensors multiple times), and requires extra memory to store intermediate tensors. Custom binary operations in the TC (i.e. providing custom definitions of addition and multiplication), which can be useful for e.g. tropical arithmetic, are not supported by cuTENSOR either [27]. Researchers needing such kernels have no choice but to implement them from scratch. Finally, cuTENSOR only supports a limited number of data types.

To alleviate these issues, we propose a flexible TC library using the GETT technique, based on the flexible and performant GEMMs in the GemmKernels library. It supports arbitrary binary and elementwise operations for the input and output matrices. The TC's general formulation of Formula (1) is thus adapted to Formula (4) in Einstein summation. Here, $\Phi_{\mathcal{ABC}}$ and $\Phi_{\mathcal{AB}}$ represent custom binary operations in terms of an arbitrary addition and multiplication, respectively. $\Psi_{\mathcal{T}}$ represents the elementwise operation applied to tensor $\mathcal{T}$.

We put forward the following requirements:

- *Flexibility:* Our library has to support custom elementwise operations, data types, and binary operators, for which we can rely on the flexibility of GemmKernels' Operators and Transforms.
- *Data-layout agnostic:* We can make no assumptions on the data layout of the input or output tensors, in order to improve interoperability with other tensor libraries.

$$\mathcal{C}_{\Pi^{\mathcal{C}}(I_m \cup I_n)} \leftarrow \Phi_{\mathcal{ABC}}\Big(\Phi_{\mathcal{AB}}\big(\Psi_{\mathcal{A}}(\mathcal{A}_{\Pi^{\mathcal{A}}(I_m \cup I_k)}), \Psi_{\mathcal{B}}(\mathcal{B}_{\Pi^{\mathcal{B}}(I_k \cup I_n)})\big), \Psi_{\mathcal{C}}(\mathcal{C}_{\Pi^{\mathcal{C}}(I_m \cup I_n)})\Big) \tag{4}$$

- *Maximise GemmKernels reuse:* We should reuse as much pre-existing infrastructure from GemmKernels as possible. This ensures that optimisations of the base matrix multiplication kernels in GemmKernels can be ported to our TC with minimal changes.
- *Similarity to cuTENSOR:* In the design of our user-facing interface, we should strive for similarity to cuTENSOR where applicable. This increases familiarity and reduces adaptation effort for users that have used cuTENSOR.

B. Components for Flexible Tensor Contractions in Julia

In order to generate the necessary components to specialise the GemmKernels kernels, our library needs the so-called *modes* for each input and output tensor. These modes are an ordered set of tensor indices, in the order that they appear in the TC. Based on these modes, our library first determines the optimal parameters for GETT, and the optimal configuration of the GEMM kernel. The process by which the optimal set of parameters is determined will be discussed in Section III-D. Next, our TC library dynamically generates custom layout components, and a GETT plan containing all information needed to specialise the GEMM kernel and compute the TC.

These layout components will be used as global memory layout of the input and output matrices of the GEMM kernel. They specialise the copying of tiles from global to shared memory by implementing a custom mapping between the GEMM kernel's 2D matrix coordinates and the corresponding tensor coordinates, in line with the GETT technique. Since GemmKernels has kernels for transposed and non-transposed inputs, we have to decide for each tensor whether the corresponding matrix is non-transposed or transposed, i.e. if our new custom layout component should be derived from the existing column-major or row-major layouts in GemmKernels. Furthermore, we need to choose the optimal access order for the indices in each of the sets I_m, I_n, and I_k. Given these parameters, our library dynamically creates a new custom layout type that extends either of the two GemmKernels layouts. Most layout-specific methods (e.g. those defining the iteration order of loops) can be reused without changes, so our GETT-kernel behaves similarly to a normal GEMM. Our implementation only needs to override the `load` and `store!` methods that transfer 2D subtiles of the input and output tensors to/from memory. The position to load from or store to is given as a parameter, and is specified in the (M, N, K)-coordinates of the GEMM. The `load` and `store!` functions convert these logical coordinates to physical memory offsets.

Consider the example TC $\mathcal{C}_{abcde} \leftarrow \mathcal{A}_{ecbfa} \cdot \mathcal{B}_{fd}$, with GETT-plan $\mathcal{C}'_{(abce)(d)} = \mathcal{A}'_{(ecba)(f)} \cdot \mathcal{B}'_{(f)(d)}$, and all tensors stored in a column-major layout. The memory offset for $\mathcal{A}$ in the GETT-kernel can be obtained based on the offsets $\text{off}_i^{\mathcal{A}}$ and strides $S_i^{\mathcal{A}}$ for each dimension i, using Formula (5).

$$\text{off}_e^{\mathcal{A}} S_e^{\mathcal{A}} + \text{off}_c^{\mathcal{A}} S_c^{\mathcal{A}} + \text{off}_b^{\mathcal{A}} S_b^{\mathcal{A}} + \text{off}_a^{\mathcal{A}} S_a^{\mathcal{A}} + \text{off}_f^{\mathcal{A}} S_f^{\mathcal{A}} \tag{5}$$

The offsets for each dimension for the matrix element (M, N, K) are given by Formula (6).

$$\begin{aligned}
\text{off}_e^{\mathcal{A}} &= M && \bmod E_e^{\mathcal{A}} \\
\text{off}_c^{\mathcal{A}} &= \left\lfloor \frac{M}{E_e^{\mathcal{A}}} \right\rfloor && \bmod E_c^{\mathcal{A}} \\
\text{off}_b^{\mathcal{A}} &= \left\lfloor \frac{M}{E_e^{\mathcal{A}} E_c^{\mathcal{A}}} \right\rfloor && \bmod E_b^{\mathcal{A}} \\
\text{off}_a^{\mathcal{A}} &= \left\lfloor \frac{M}{E_e^{\mathcal{A}} E_c^{\mathcal{A}} E_b^{\mathcal{A}}} \right\rfloor && \bmod E_a^{\mathcal{A}} \\
\text{off}_f^{\mathcal{A}} &= K && \bmod E_f^{\mathcal{A}}
\end{aligned} \tag{6}$$

In general, the memory offset for a tensor is a sum of terms of the form $\left(\left\lfloor \frac{\text{GEMM index}}{\text{divisor}} \right\rfloor \bmod \text{modulus}\right) \cdot \text{stride}$. The set of divisors, moduli, and strides for each dimension are collected in three separate lists, and passed as type parameters to a generic GETT-layout. Listing 1 shows the implementation of this layout and the custom implementation of its `load` function. The implementation of the `store!` function is analogous to the `load` function. Lines 1–2 and lines 3–4 each define a parameterised GETT-layout type based respectively on the GemmKernels column-major `UnsafeAlignedColMajor` type and its row-major `UnsafeAlignedRowMajor` type. The type parameters `divT1`, `modT1`, `strides1`, `divT2`, `modT2`, and `strides2` represent the set of divisors, moduli, and strides for the first and second matrix dimension of the layout (e.g. respectively M and K in our previous example). Lines 6–33 provide the implementation of the custom `load` function, which takes as arguments the `workspace` to load from, and a `tile` representing the matrix coordinates to load from. Note that the `load` (and `store!`) implementation is identical for row- and column-major layouts (lines 7–10). Lines 14–15 compute the matrix-index in the first and second dimensions. The loops in lines 17–25 then compute the memory offset to load from, in a similar way to Formulae (5) and (6). GemmKernels' `load` and `store!` functions operate on tiles, which may contain more than one element. The calculated `offset` represents the memory offset of the first element to load only. The remaining lines 27–32 deal with loading the remaining elements of the tile. It relies on two pre-computed type parameters of the GETT-layout: `strideOverExtent`, which determines the stride between contiguous elements in the tile, and `isLoadOrStoreStrided`, a boolean that represents whether elements are stored contiguously in memory. Line 27 computes the total number of elements that need to be loaded. Line 28 checks whether the memory accesses have stride-1. If so, our implementation uses vectorised stores, for which it reuses GemmKernels' built-in `vloada` function on line 29. In the other case, our layout component dispatches automatically to a strided load implementation `sloada` on line 31, passing the stride between adjacent elements `strideOverExtent`. Note that due to Julia's compilation flow with type inference

```
1  abstract type GETTLayoutColMajor{T, divT1, modT1, strides1, divT2, modT2, strides2,
2                   isLoadOrStoreStrided, strideOverExtent} <: Layout.UnsafeAlignedColMajor{T} end
3  abstract type GETTLayoutRowMajor{T, divT1, modT1, strides1, divT2, modT2, strides2,
4                   isLoadOrStoreStrided, strideOverExtent} <: Layout.UnsafeAlignedRowMajor{T} end
5
6  Base.@propagate_inbounds @inline function Layout.load(
7          ::Union{ Type{GETTLayoutColMajor{T, divT1, modT1, strides1, divT2, modT2, strides2,
8                                           isLoadOrStoreStrided, strideOverExtent}},
9                   Type{GETTLayoutRowMajor{T, divT1, modT1, strides1, divT2, modT2, strides2,
10                                          isLoadOrStoreStrided, strideOverExtent}}
11                }, workspace, tile::Tile{size}
12     ) where {T,divT1,modT1,strides1,divT2,modT2,strides2,isLoadOrStoreStrided,strideOverExtent,size}
13
14     G1 = tile.base[1] + tile.offset[1]
15     G2 = tile.base[2] + tile.offset[2]
16
17     offset = 1
18     @loopinfo unroll for i in eachindex(divT1)
19         stride_offset = (G1 ÷ divT1[i]) % modT1[i]
20         offset += stride_offset * strides1[i]
21     end
22     @loopinfo unroll for i in eachindex(divT2)
23         stride_offset = (G2 ÷ divT2[i]) % modT2[i]
24         offset += stride_offset * strides2[i]
25     end
26
27     NUMEL = size[1] * size[2]
28     if (isLoadOrStoreStrided == false)
29         return Layout.vloada(Layout.Vec{NUMEL, T}, pointer(workspace), offset)
30     else
31         return GETTLayout.sloada(Layout.Vec{NUMEL, T}, workspace, offset, strideOverExtent)
32     end
33 end
```

Listing 1: Custom GETT-layout type definition based on GemmKernels' built-in layouts, and its overridden `load` function.

and specialisation, the type parameters are compile-time constants. This eliminates much of the run time overhead of the offset computations in lines 17–25, and the check on line 28.

This process is repeated for the $\mathcal{A}$-, $\mathcal{B}$-, $\mathcal{C}$-, and $\mathcal{D}$-tensors. A GemmKernels kernel is then launched using those layout components in global memory. The size of the GEMM is set to $M = \prod_{i \in I_m} E_i$, $N = \prod_{i \in I_n} E_i$, and $K = \prod_{i \in I_k} E_i$. Optionally, the user can also provide custom operators and elementwise operations to apply to the GEMM, which are passed along to GemmKernels before launching the kernel.

One important limitation in our implementation is that we assume that the entire tensor can be covered by an integer number of thread block tiles. For the M-dimension, this comes down to the requirement that $M = \prod_{i \in I_m} E_i$ is divisible by the thread block tile size in the M-dimension. If this requirement is not met, the user needs to pad the tensors with zeros before the GEMM kernel is launched.

C. Optimisations to GemmKernels

To improve performance, we have also made three changes to the template kernels in GemmKernels. These changes are not specific to TCs; they benefit GEMMs as well.

The first change relates to the iteration order of memory copy operations in GemmKernels. These depend on the row- or column-major nature of the used layouts. In the original kernels, only the copies for the A- and B-matrices supported both row- and column-major orders: changing the memory layout had no influence for the C- and D-matrices. Because BLAS only supports transposing A and B, there was no need for alternative layouts for C or D. In order to fix this, we adapted the prologue and epilogue of the kernels to take into account the layout in the iteration order, and introduced a new kernel configuration parameter, `is_cd_colmajor`.

The second optimisation is borrowed from CUTLASS [30], and is referred to as CTA or thread block swizzling. It applies a swizzling function to the mapping between thread block IDs and tiles of the resultant D matrix. The goal is to maximise the probability that thread blocks that access the same tile of A or B are scheduled on neighbouring SMs at the same time, increasing L2 hit rate. We have noticed that this can have a massive impact on performance. On one TC, we noticed an increased L2 hit rate from 38% to 95%, reducing the total number of bytes requested from DRAM by 91%. This ultimately resulted in a performance improvement of $4.74\times$.

This optimisation is implemented through a new component, `CTASwizzle`, with two associated methods that determine the needed number of thread blocks and the mapping between thread block ID and the rows and columns of the output matrix that should be computed by the thread block with that ID. We support three swizzling functions. The first, `identity`, is equivalent to GemmKernels' old behaviour. The second is `HorizontallyTiled`, and has a configurable tile size N. It assigns the first N tiles in the first row to thread blocks with ID $1, 2, \ldots, N$, the first N tiles in the second row to $N + 1, \ldots, 2N$, and so on. The pattern is then continued for every block of N columns. The third swizzling function, `VerticallyTiled`, works symmetrically to the horizontally tiled swizzling function.

The final optimisation is an additional kernel template utilising software pipelining, as an alternative to the already existing pipelined kernel in GemmKernels. Our alternative reorders memory instructions and computations compared to GemmKernels' pipelined kernel. As a result, we can reduce the number of synchronisation instructions by a factor 3, reducing the number of stalls. This necessitates allocating two tiles of A and B in shared memory, however, so the tile size in the K-dimension must be halved, reducing data reuse.

D. Determining the Optimal GETT-Parameters

As mentioned in Section III-B, our GETT implementation determines the optimal parameters for launching the GEMM kernel during plan generation. This includes parameters for which the possible values are independent of the TC, and those that are TC-specific. The former category includes tile sizes of the underlying GEMM, launch configuration (e.g. the total number of warps per thread block and how to assign subtiles to warps), the operator size to use (e.g. the WMMA shape), the kernel to use (non-pipelined vs. GemmKernels' original pipelined vs. our new pipelined version), and which thread block swizzling function to use (horizontal vs. vertical, and which tile size). It also includes some parameters for each of the custom layouts we generate for the $\mathcal{A}$-, $\mathcal{B}$-, and $\mathcal{D}$-tensors, i.e. whether to base them on existing row-major or column-major layouts. For this category, we consider all possible values that in our experience perform well on one or multiple TCs, totalling half a million possible configurations.

The latter category contains the remaining parameters for the custom layouts, namely the permutations of the index sets I_m, I_n, and I_k that determine in which order the indices in them are accessed. The number of possible values for this category depends on the number of permutations of the index sets I_m, I_n, and I_k, and thus varies for different TCs. For a GEMM, $|I_m| = |I_n| = |I_k| = 1$, so no choices need to be made. For TCs of higher dimensionality, as found, e.g., in quantum chemistry calculations [9], typical values are $|I_m| = |I_n| = 3, |I_k| = 1$, yielding $3! \cdot 3! \cdot 1! = 36$ possibilities.

The total design space size can thus vary from slightly over half a million candidate kernels for GEMM to almost 20 million candidates for quantum-chemistry-related TCs. Given the large design space, the optimal GETT parameters should ideally be determined automatically for the user. We experimented with choosing the optimal parameters based on heuristics that try to maximise stride-1 memory accesses and vectorisation opportunities. Note that each index in the sets I_m, I_n, and I_k occurs in two matrices, which may have conflicting optimal orders. Finding a good compromise turned out to be hard with heuristics, leaving a lot of performance on the table.

We explored hyperparameter optimisation using the sampling strategies supported by the Julia package Hyperopt.jl [31]: uniformly random search, Latin hypercube sampling [32], bandit-based Hyperband [33], and BOHB [34]. None of them yielded significantly improved results over a simple brute-force random search. In correspondence with the package maintainer, we learnt that most optimisers in Hyperopt.jl do not take into account previous samples to guide the selection of future points. The sole exception is BOHB, but this technique is less suited for integer-valued hyperparameters such as the tile sizes.

The approach we settled for in the end, is tuning the parameters using random sampling of the design space. For each TC, we select a random subset of candidate kernels that we compile and profile. The parameters of the configuration that performs the fastest are then selected and stored in the GETT plan for later reuse. In this tuning process, the accuracy of the performance measurements of the candidate kernels is of paramount importance. We improve it by averaging the execution time over 5 runs, and locking the GPU and memory clock speeds. We also monitor the GPU state and sleep momentarily when throttling is detected.

```
1  tensorModes = [[1, 2, 3, 4, 5], # C_abcde
2                 [5, 3, 2, 6, 1], # A_ecbfa
3                 [6, 4]]          # B_fd
4  extents = (16, 16, 16, 64, 16, 64)
5  dataType = Float16
6  computeType = Float16
7  accumulateType = Float32
8
9  elOpB = x -> if x>0 x else exp(x)-1 end # ELU
10 α, β = 1, 0
11
12 A = CUDA.randn(dataType, extents[tensorModes[2]])
13 B = CUDA.randn(dataType, extents[tensorModes[3]])
14 C = CUDA.zeros(dataType, extents[tensorModes[1]])
```

Listing 2: Set-up code common to GemmKernels and cuTENSOR, for the TC $\mathcal{C}_{abcde} \leftarrow \mathcal{A}_{ecbfa} \cdot \mathcal{B}_{fd}$.

E. Example usage

To illustrate how end users can use our flexible TCs, consider again the TC $\mathcal{C}_{abcde} \leftarrow \mathcal{A}_{ecbfa} \cdot \mathcal{B}_{fd}$. Listing 2 shows the necessary Julia set-up code, which can also be used to compute the TC using cuTENSOR. The TC to perform is defined in terms of the modes for each of the tensors $\mathcal{C}$, $\mathcal{A}$, and $\mathcal{B}$ (lines 1–3) and the extents for each dimension (line 4). Lines 5–7 set the data types to store the tensors in global memory, the type that the inputs $\mathcal{A}$ and $\mathcal{B}$ are cast to during computations, and the type of the accumulator. A custom ELU elementwise operation is defined in line 9, and line 10 sets α and β in order to perform an in-place TC without scaling. Finally, lines 12–14 create the input and output tensors as GPU-backed arrays of the correct extent in global memory.

After this set-up, Listing 3 can be used to launch a GemmKernels.jl kernel to perform the TC. First, lines 1–17 create a TC plan, which contains the parameters of the GETT TC. This plan need only be created once for each TC, and can be reused to launch multiple kernels for the same TC. In order to provide the necessary flexibility outlined in Section III-A, the user can pass custom operators (line 11) and elementwise operations (line 4) to the underlying GEMM kernel. Optionally, launch parameters of the GEMM that influence performance such as the chosen kernel and tile sizes can also be customised (see lines 13–16). Secondly, line 19 then performs the TC using this created plan by launching a correctly configured GEMM kernel from GemmKernels.

Listing 4 shows the cuTENSOR version of Listing 3. Note that our interface mimics the design of cuTENSOR, which improves familiarity for users coming from cuTENSOR. cuTENSOR's interface also consists of two steps: creating the plan in lines 4–11, and executing the TC in lines 13–22. There are three important differences between our interface and cuTENSOR's, however. For one, the elementwise operation that we have chosen, ELU, is not (yet) supported by cuTENSOR, and can hence not be fused. Line 2 in Listing 4 therefore launches a separate kernel to perform this elementwise operation. The second difference is that cuTENSOR's customisation is much more limited compared to our interface. While we can choose

```
plan = Tensors.ContractionPlan(
  α,
  A, TensorDescriptor(A), tensorModes[2],
  B, TensorDescriptor(B; unaryOp=elOpB),
  tensorModes[3],
  β,
  C, TensorDescriptor(C), tensorModes[1],
  C, TensorDescriptor(C), tensorModes[1];
  computeType = computeType,
  accumulateType = accumulateType,
  operator = Operator.WMMAOp{8, 32, 16,
                 computeType, accumulateType},
  blockShape = (M = 256, N = 64, K = 64),
  warpsPerBlock = 4,
  computeWarp = (M = 64, N = 64, K = 16),
  kernel = Kernel.matmul_pipelined
)

Tensors.contraction!(plan, α, A, B, β, C, C)
```

Listing 3: The TC $\mathcal{C}_{abcde} \leftarrow \mathcal{A}_{ecbfa} \cdot \mathcal{B}_{fd}$ with our library.

```
# Apply elop not supported by cuTENSOR separately
B = elOpB.(B)

plan = cuTENSOR.plan_contraction(
  A, tensorModes[2], cuTENSOR.CUTENSOR_OP_IDENTITY,
  B, tensorModes[3], cuTENSOR.CUTENSOR_OP_IDENTITY,
  C, tensorModes[1], cuTENSOR.CUTENSOR_OP_IDENTITY,
  cuTENSOR.CUTENSOR_OP_IDENTITY;
  compute_type = computeType,
  algo = cuTENSOR.CUTENSOR_ALGO_GETT
)

cuTENSOR.contract!(
  α,
  A, tensorModes[2], cuTENSOR.CUTENSOR_OP_IDENTITY,
  B, tensorModes[3], cuTENSOR.CUTENSOR_OP_IDENTITY,
  β,
  C, tensorModes[1], cuTENSOR.CUTENSOR_OP_IDENTITY,
  cuTENSOR.CUTENSOR_OP_IDENTITY,
  compute_type = computeType,
  plan = plan
)
```

Listing 4: The TC $\mathcal{C}_{abcde} \leftarrow \mathcal{A}_{ecbfa} \cdot \mathcal{B}_{fd}$ with cuTENSOR.

the compute type and algorithm that are used for the TC, we cannot customise kernel parameters such as tile sizes. The final difference is that our plan generation additionally requires specifying α and β. This allows our library to specialise the kernel depending on their values, e.g. eliminating the loading of the $\mathcal{C}$-tensor if β is 0.

IV. Evaluation

A. Measurements

For our evaluation, we reuse the TCCG benchmark suite initially proposed by Springer and Bientinesi for their performant TC generator [5], which has since been used in follow-up work as well [6], [35], [36], [37]. The benchmark suite consists of 48 TCs, collected from real use cases in the scientific literature. Most of the TCs are from the field of computational chemistry: 18 highly-dimensional TCs from the CCSD(T) method [9], 19 from coupled-cluster methods [38], and 3 arising in change-of-basis formulae for integrals occurring in quantum chemistry calculations [39]. The remaining 8 TCs are tensor-matrix multiplications, also extracted from the literature [12]. Table I lists all 48 TCs, their extents, and equivalent GEMM size.

We use consumer and data centre-grade GPUs of multiple generations: two Volta GPUs (V100 and V100S), one Turing GPU (RTX 2080 Ti), and three GPUs of the Ada Lovelace-generation (RTX 4070, RTX 4080, and RTX 6000 Ada). For each TC, we choose data types supported by all GPU generations: $\mathcal{A}$- and $\mathcal{B}$- tensors are 16-bit floating point, $\mathcal{C}$- and $\mathcal{D}$-tensors are 32-bit floating point. While Tensor Cores of generations after Volta support additional data types, we limit our performance evaluation to 16-bit floating point inputs with 32-bit floating point accumulation[1]. This is in line with recent literature, which even on the recent A100 GPU of the

Table I
TCCG Benchmark TCs

#	Tensor Contraction	Extents	M	N	K
1	abc-bda-dc	312 312 24 312	97344	24	312
2	abc-dca-bd	312 24 296 312	92352	24	312
3	abcd-dbea-ec	72 72 24 72 72	373248	24	72
4	abcd-deca-be	72 24 72 72 72	373248	24	72
5	abcd-ebad-ce	72 72 24 72 72	373248	24	72
6	abcde-efbad-cf	48 32 24 32 48 32	2359296	24	32
7	abcde-ecbfa-fd	48 32 32 24 48 48	2359296	24	48
8	abcde-efcad-bf	48 24 32 32 48 32	2359296	24	32
9	abcd-ea-ebcd	72 72 72 72 72	72	373248	72
10	abcd-eb-aecd	72 72 72 72 72	72	373248	72
11	abcd-ec-abed	72 72 72 72 72	72	373248	72
12	ab-ac-cb	5136 5120 5136	5136	5120	5136
13	ab-acd-dbc	312 296 296 312	312	296	92352
14	ab-cad-dcb	312 296 312 312	312	296	97344
15	abc-acd-db	312 296 296 312	92352	296	312
16	abc-ad-bdc	312 312 296 296	312	92352	296
17	abc-adc-bd	312 312 296 296	92352	312	296
18	abc-adc-db	312 296 296 312	92352	296	312
19	abc-adec-ebd	72 72 72 72 72	5184	72	5184
20	abcd-aebf-dfce	72 72 72 72 72 72	5184	5184	5184
21	abcd-aebf-fdec	72 72 72 72 72 72	5184	5184	5184
22	abcd-aecf-bfde	72 72 72 72 72 72	5184	5184	5184
23	abcd-aecf-fbed	72 72 72 72 72 72	5184	5184	5184
24	abcd-aedf-bfce	72 72 72 72 72 72	5184	5184	5184
25	abcd-aedf-fbec	72 72 72 72 72 72	5184	5184	5184
26	abcd-aefb-fdce	72 72 72 72 72 72	5184	5184	5184
27	abcd-aefc-fbed	72 72 72 72 72 72	5184	5184	5184
28	abcd-eafb-fdec	72 72 72 72 72 72	5184	5184	5184
29	abcd-eafc-bfde	72 72 72 72 72 72	5184	5184	5184
30	abcd-eafd-fbec	72 72 72 72 72 72	5184	5184	5184
31	abcdef-dega-gfbc	24 16 16 24 16 16 24	9216	4096	24
32	abcdef-degb-gfac	24 16 16 24 16 16 24	6144	6144	24
33	abcdef-degc-gfab	24 16 16 24 16 16 24	6144	6144	24
34	abcdef-dfga-gebc	24 16 16 24 16 16 24	9216	4096	24
35	abcdef-dfgb-geac	24 16 16 24 16 16 24	6144	6144	24
36	abcdef-dfgc-geab	24 16 16 24 16 16 24	6144	6144	24
37	abcdef-efga-gdbc	24 16 16 16 24 16 24	9216	4096	24
38	abcdef-efgb-gdac	24 16 16 16 24 16 24	6144	6144	24
39	abcdef-efgc-gdab	24 16 16 16 24 16 24	6144	6144	24
40	abcdef-gdab-efgc	24 16 16 16 24 16 24	6144	6144	24
41	abcdef-gdac-efgb	24 16 16 16 24 16 24	6144	6144	24
42	abcdef-gdbc-efga	24 16 16 16 24 16 24	4096	9216	24
43	abcdef-geab-dfgc	24 16 16 24 16 16 24	6144	6144	24
44	abcdef-geac-dfgb	24 16 16 24 16 16 24	6144	6144	24
45	abcdef-gebc-dfga	24 16 16 24 16 16 24	4096	9216	24
46	abcdef-gfab-degc	24 16 16 24 16 16 24	6144	6144	24
47	abcdef-gfac-degb	24 16 16 24 16 16 24	6144	6144	24
48	abcdef-gfbc-dega	24 16 16 24 16 16 24	4096	9216	24

[1]We additionally verified our kernels' correctness using FPU operations (not Tensor Cores) across multiple data types: 32- and 64-bit floating point, and 16-, 32-, and 64-bit integers. Nevertheless, we restrict the scope of our detailed performance analysis to FP16 using Tensor Cores. The CUDA.jl package on which we depend currently lacks WMMA support for newer data types (TF32, bfloat16, FP64), preventing evaluation on those types. However, since both GemmKernels.jl and our extension are parameterised on WMMA shape and type, adding future support for these data types should require minimal effort.

Ampere generation, still uses 16-bit inputs [40], [41], even for algorithms in scientific disciplines such as computational chemistry where precision is traditionally important [42], [43]. Performance for these other data types may vary, as changing the data types changes the memory access pattern, which can have a significant impact on performance. A detailed performance analysis for these other data types is future work.

We generated kernels with Julia 1.11.4, LLVM 16.0.6, and CUDA.jl 5.9.2, and used CUDA Toolkit 13.0.0, and the open NVIDIA kernel module v.580.95.05. Table II lists specifications of the machines over which our GPUs are spread.

We zero-pad the input tensors for our kernels such that the matrix dimensions are multiples of the thread block tile size. This padding step needs to be performed only once: subsequent tensors that are the output of TCs are automatically zero-padded. We hence do not include the time necessary for this padding in the time measurements in this section. Section IV-F will discuss the memory and run time overhead of padding in detail, as well as potential options to eliminate the need for padding entirely.

We compare our performance to that of cuTENSOR 2.0.1. We use the original, non-padded inputs for cuTENSOR, as it does not require padding. We verified the correctness of our reported kernels by comparing their output to cuTENSOR's.

We take the minimum of 5 runs of each TC, for our kernels and cuTENSOR's. Figure 1 shows the relative performance for each GPU and for each TC i, i.e. $\text{rel\_perf}_i = \frac{t_{\min,i,\text{cuTENSOR}}}{t_{\min,i,\text{ours}}}$. The TCs shown in a colour other than blue indicate that we discuss them in more detail further in this section. The figure also shows the geomean speedup of the relative performances $\text{GM}_1 = \text{geomean}(\{\text{rel\_perf}_i\})$ across all 48 TCs in black. Additionally, we show the geomean speedup $\text{GM}_2 = \text{geomean}(\{\max(1, \text{rel\_perf}_i)\})$ in red. This is the geomean speedup observed by an end user that can choose the best performing kernels between cuTENSOR's and ours over an end user whose only choice was the existing cuTENSOR.

B. Overall Results per Architecture

The discrepancy in relative performance of our Julia TC package between different architectures can generally be explained by the amount of architecture-specific optimisations that are relevant for that architecture. Unlike cuTENSOR, our package does not implement any such optimisations for its kernels, so we expect them to exhibit lower relative performance on newer architectures that can benefit from more available optimisations. For both Volta and Turing, the only significant optimisation is the use of `mma.sync` combined with shared memory swizzling [19]. Since this optimisation seems to be much more important for Volta than it is for Turing (see later), we observe our kernels' best relative performance on the Turing-generation RTX 2080 Ti, with geomean speedups of $\text{GM}_1 = 195\%$ and $\text{GM}_2 = 199\%$. The V100 and V100S, both of the Volta generation, follow with identical speedups of $\text{GM}_1 = 127\%$ and $\text{GM}_2 = 162\%$. Interestingly, this performance was obtained with kernels using the same parameters for the V100 and V100S, i.e. no separate tuning for the V100S was necessary. This is likely because both GPUs are architecturally very similar, even though the V100S has a higher memory capacity, memory bandwidth, and FP16 Tensor Core throughput compared to the V100 [44]. Finally, on the Ada-generation GPUs our kernels exhibit the lowest relative performance, since these GPUs additionally allow for the asynchronous copy optimisation [20]. For these GPUs, we achieve speedups of $\text{GM}_1 = 120\%$, $\text{GM}_2 = 130\%$ (RTX 4070), $\text{GM}_1 = 117\%$, $\text{GM}_2 = 126\%$ (RTX 4080); and $\text{GM}_1 = 121\%$, $\text{GM}_2 = 129\%$ (RTX 6000 Ada). In conclusion, our kernels achieve a net speedup across different GPU architectures, even though we did not yet incorporate any architecture-specific optimisations.

C. Detailed Performance Analysis

To gain more insights in the obtained results, we profile TCs with outlying performance, either for a specific architecture or across architectures, using NVIDIA Nsight Compute [45].

Across all architectures, TC13 (orange in Figure 1) is among the worst performing TCs. For Volta and Turing, cuTENSOR's implementation consists of two kernels: a GEMM kernel of which the launch configuration contains a lot more thread blocks than our optimal kernel (thus exploiting parallelism more), and an additional reduction kernel. This is likely a Split-K optimisation, specific for problems with small M and N, and large K [46]. A normal GEMM implementation on GPUs parallelises tiles in the M- and N-dimensions across thread blocks. Split-K additionally parallelises across the K-dimension, and then sums the resulting tiles using a separate reduction kernel, improving parallelism at the cost of a separate reduction step and additional memory to store intermediate results. Indeed, out of the suite's 48 TCs, TC13 is the one for which $M \cdot N$ is the smallest, and K is the largest. For Ada Lovelace, we do not notice a second reduction kernel, but cuTENSOR's launch configuration has 7.5x – 17.5x as many thread blocks as ours, hence increasing parallelism. This may again indicate a Split-K optimisation, but fused in the first GEMM kernel, or other similar optimisations that improve parallelism for these kinds of problem sizes.

TC14 (green in Figure 1) is similar to TC13 in many ways: same tensor dimensionality, similar resulting GEMM sizes, and cuTENSOR uses a separate reduction kernel for it. Nonetheless, our kernels' relative performance is much better for this TC. The reason lies in the different order of indices in the TC, and thus a different, suboptimal memory access pattern that results in two orders of magnitude more stalls related to global memory operations for TC14 than TC13 in cuTENSOR, slowing down the cuTENSOR kernel by a factor of $40\times$. We conjecture that the run time is dominated by this suboptimal memory access pattern in global memory, so that the Split-K optimisation is less important to determine performance, closing the gap between our kernels and cuTENSOR's.

The next interesting category of TCs consists of TC20, TC22, and TC24 (red in Figure 1), for which our kernels perform badly on Volta, but exhibit decent performance on other architectures. When comparing the ratio of bank conflicts when loading from shared memory between our kernels and cuTENSOR's, we notice that, for Volta, our kernels have

Table II
DETAILED SPECIFICATIONS FOR THE MACHINES USED FOR EVALUATION

Specification	Machine 1	Machine 2	Machine 3	Machine 4	Machine 5
CPU	Intel Xeon E5-2603 v4	Intel Xeon E5-2637 v2	AMD TR 2990WX	AMD 7800X3D	AMD TR PRO 7985WX
RAM	382 GB	64 GB	128 GB	32 GB	512 GB
GPU	V100, V100S	RTX 2080 Ti	RTX 4070	RTX 4080	RTX 6000 Ada
OS	Ubuntu 24.04.3	Debian 12	Debian 12	Arch Linux	Ubuntu 24.04.3

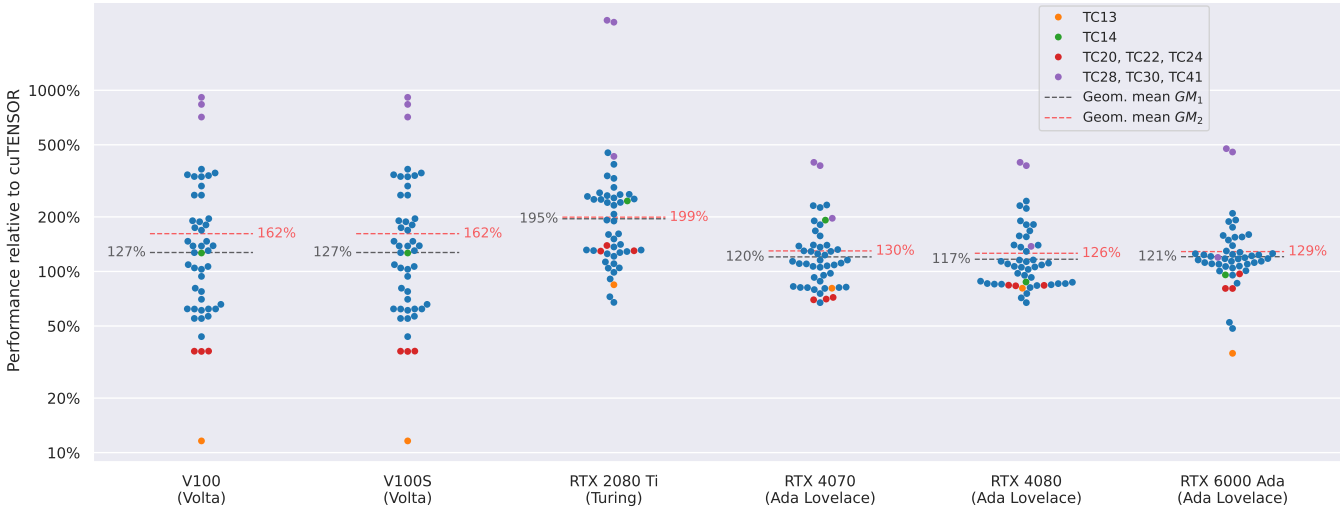

Figure 1. Swarmplot of the performance results of our kernels relative to cuTENSOR on different architectures, for the 48 TCs in the TCCG benchmark suite, along with the two geometric means GM_1 and GM_2 for each GPU architecture. Explicitly discussed groups of TCs are shown in different colours. Note that the y-axis, showing the relative performance $\frac{t_{\min,i,\text{cuTENSOR}}}{t_{\min,i,\text{ours}}}$, is logarithmic.

$199\times$, $353\times$, and $529\times$ as many bank conflicts. Compare this with the kernels for the same TCs on Turing having $3\times$ *fewer* bank conflicts compared to cuTENSOR, and on Ampere having at most an order of magnitude increase in bank conflicts. Again, this seems to indicate that the shared memory swizzling optimisation to reduce shared memory bank conflicts is more important for Volta than for Turing or Ampere. Indeed, we notice that the TCs for which our kernels perform the worst for Volta roughly correspond to the TCs for which the bank conflict ratio $\frac{\text{\# conflicts ours}}{\text{\# conflicts cuTENSOR}}$ is the largest. However, this is not a general rule: both TC28 and TC30 have high bank conflict ratios as well, and for these two TCs our kernels are among the best performing on Volta.

The final category of TCs to discuss is TC28, TC30, and TC41 (shown in purple in Figure 1). For these TCs our kernels achieve relative performances of 660% – 891% on Volta. The most prominent difference with cuTENSOR's kernels lies in memory-related metrics. cuTENSOR's have more global-memory-related stalls, and an increased number of transactions from device memory to the L2-cache, as well as from the L2-cache to the L1-cache. This might be because of a suboptimal memory access pattern for cuTENSOR. Given its closed-source nature, it is unclear whether this is the result of lacking heuristics in cuTENSOR, or because its kernels are not flexible enough to enable the use of a better memory access pattern like our kernels. Finally, we note that, while our kernels exhibit large relative performance on other architectures as well for these three TCs, the difference in the discussed metrics is less pronounced there. For Ampere, one possible explanation is that asynchronous copy allows for better latency hiding, thus reducing the effect of suboptimal access patterns.

D. Operator Fusion

One important advantage of our kernels over cuTENSOR's is the ability to fuse arbitrary elementwise operations. While cuTENSOR also allows for operator fusion, the supported elementwise operations are limited to a pre-defined list, whereas our kernels support arbitrary Julia expressions as operations. To study the impact of operator fusion on performance, we execute all 48 TCs in the TCCG benchmark suite for two additional cases. Both cases involve identical elementwise operations applied to $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, i.e. they are TCs of the form $\mathcal{C}_{\Pi^{\mathcal{C}}(I_m \cup I_n)} \leftarrow \Psi\big(\Psi(\mathcal{A}_{\Pi^{\mathcal{A}}(I_m \cup I_k)}) \cdot \Psi(\mathcal{B}_{\Pi^{\mathcal{B}}(I_k \cup I_n)})\big)$. For the first case, we chose $\Psi = \mathrm{ReLU}$, a common activation function in neural networks [47]. This activation function is supported by cuTENSOR, and is hence fused in the TC kernel. The second case involves $\Psi = \text{Leaky ReLU}$, a variant of ReLU that avoids zero gradient for negative inputs [29]. As this activation function is not supported by cuTENSOR, it cannot be fused, and requires an additional kernel launch and the reloading of data. If the original data needs to be kept, cuTENSOR's non-fused approach additionally requires extra storage for this configuration. For the evaluation in this section, we assume the original data can be overwritten, and perform the elementwise kernels in-place. Our kernels fuse all elementwise operations for both configurations.

We do not perform a separate sweep for the kernels here, but instead reuse the optimal parameters from Section IV-A.

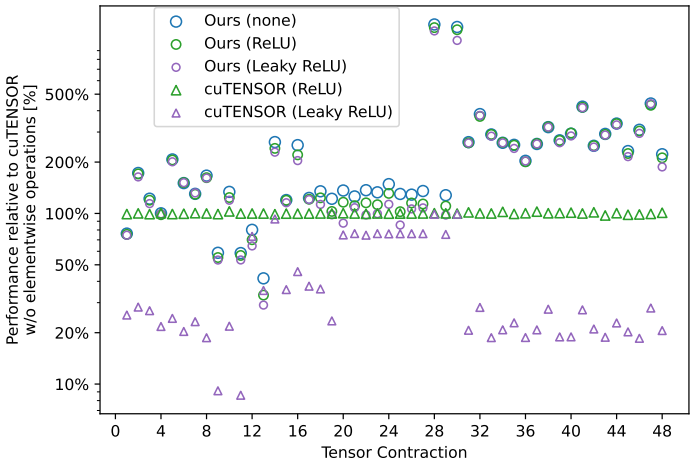

Figure 2. Performance results on the TCCG benchmarks of our kernels (◯) relative to cuTENSOR (△) for three configurations: without elementwise operations (blue); applying ReLU to $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ (green); and applying a leaky ReLU to $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ (purple). Performance is relative to cuTENSOR without elementwise operations. Except for Leaky ReLU for cuTENSOR, all elementwise operations are fused in the TC kernel. All results are obtained on the RTX 2080 Ti (Turing) by taking the minimum over 10 runs.

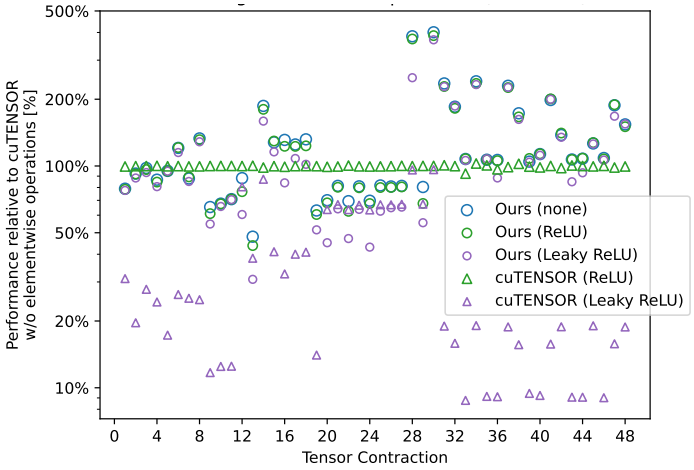

Figure 3. RTX 4070 performance results with elementwise operations.

The additional elementwise operation may increase register usage beyond the SM-register limit, resulting in the kernel no longer being able to be launched with the required number of threads. For that reason, we additionally instruct the PTX compiler to limit register usage by introducing spills. This was ultimately only necessary for one configuration (TC 28 with leaky ReLU on the RTX 4070). For the other configurations, this extra option has no influence on the generated instructions.

Figures 2 to 4 show performance results for the RTX 2080 Ti, RTX 4070, and RTX 6000 Ada, respectively, relative to cuTENSOR without elementwise operations. As expected, cuTENSOR with ReLU performs almost identically to cuTENSOR without elementwise operations, as this operation is fused. cuTENSOR with Leaky ReLU cannot use fusion, resulting in large slowdowns from additional kernel launches and data reloading. On the other hand, our kernels exhibit minimal slowdowns when introducing elementwise operations, as they are all fused in the TC kernel.

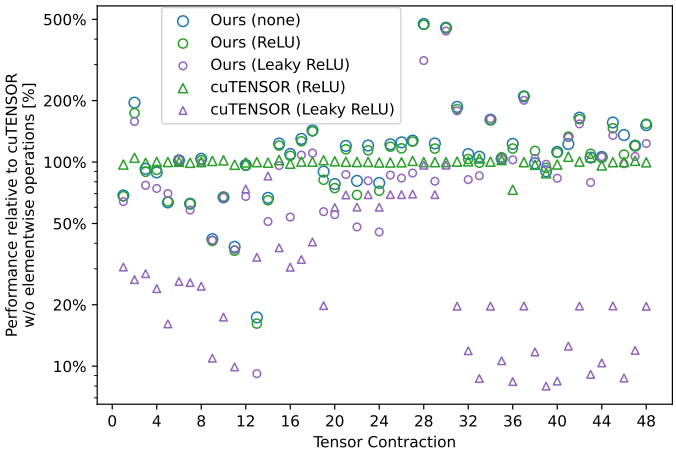

Figure 4. RTX 6000 Ada performance results with elementwise operations.

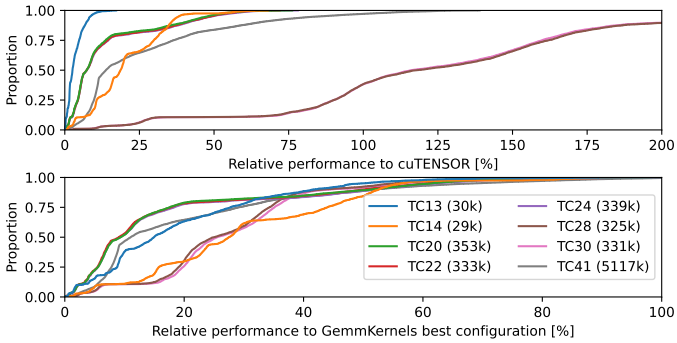

Figure 5. Cumulative distributions of the relative performance for randomly sampled, valid configurations for 8 TCs on RTX 6000 Ada. Top: performance relative to cuTENSOR. Bottom: performance relative to our best configuration. Each distribution samples 1% of the total search space for each TC. Parenthesised numbers indicate estimated total valid configurations.

E. Random Sampling Efficiency

Section III-D discussed that we determined the best TC configurations by randomly sampling. For the eight TCs analysed in Section IV-C, Figure 5 shows the cumulative distribution functions (CDFs) of the relative performance obtained with randomly sampled, valid configurations, covering around 1% of the total search space for each TC.

Well-performing configurations are sparse, so extensive sampling is needed to approach optimal performance. Our prototype sweep pipeline uses parallel compilation workers (limited by the number of CPU cores and RAM), and one measurement worker per GPU. The configuration space sampling time hence depends heavily on the used machine. On our RTX 6000 Ada machine (specifications in Table II), compilation workers averaged 0.18 valid configurations per second per worker, while the measurement worker achieved 0.30 configurations per second per GPU, meaning two compilation workers could saturate one measurement worker.

Measurement throughput has room for improvement, however. Our sweep pipeline filters out invalid configurations at multiple stages, such as before kernel generation by checking preconditions on parameter assumptions our GETT implementation makes, after compilation when, e.g., the PTX compiler failed to find sufficient registers, and during testing when a

kernel crashes, e.g., due to an illegal memory access. The later the filtering, the more time is wasted. This is particularly the case when a kernel crashes, because it requires a complete GPU worker restart. While we invested some effort in advancing the filtering in our research prototype pipeline by inserting additional preconditions, around 2% of the configurations still get filtered as they crash. As a result, startup and one-time operations (e.g. data allocation and initialisation) consumed up to 76% of measurement worker wall time. In our estimation, eliminating all such crashing configurations would speed up the measurement to around 1.03 configurations per second per GPU. Further improvements might come from reducing inter-measurement sleeping periods, currently used to prevent GPU throttling that adversely affects measurement accuracy.

These throughputs enable estimation of the sweep time required to reach a target performance. To that end, the legend of Figure 5 shows estimated total valid configurations (V) for each TC. For a TC with CDF $F(x) = \mathrm{Prob}[\mathrm{perf} \leq x]$, finding a configuration with performance $\geq x$ requires sampling at most $n = F(x) \cdot V$ configurations. The corresponding worst-case run time on a machine with g GPUs can be calculated using the measurement throughput as follows:

$$\frac{n\,\mathrm{configurations}}{g\,\mathrm{GPUs} \cdot 0.30\,\mathrm{configurations\,s^{-1}\,GPU^{-1}}} \tag{7}$$

However, sweeps can be terminated much earlier. The number of configurations tested before finding one with performance $\geq x$ follows a negative hypergeometric distribution $\mathrm{NHG}_{N,K,r}(k)$ with $N = V$, $K = F(x) \cdot V$, and $r = 1$. The expected number of tested configurations is $n = \mathbb{E}[k] + 1 = \frac{V+1}{V(1-F(x))+1}$, convertible to an expected run time using Formula (7).

Consider TC30 as a concrete example. Figure 5 (top) shows $F(100\%) = 0.38$ and $V = 331\,000$, i.e. 38% out of 331 000 valid configurations underperform cuTENSOR. On a single-GPU machine, finding a cuTENSOR-beating configuration requires sweeping at most $125\,780$ configurations in the worst case ($116\,\mathrm{h}$), but only 1.61 configurations on average ($5.4\,\mathrm{s}$).

As speed-ups over cuTENSOR can be obtained within minutes, running sweeps for frequently-executed TCs seems feasible in production. However, finding the optimal configurations remains time-consuming due to the flat tail of CDFs at high relative performances (Figure 5, bottom). Techniques to prune the search space or to identify promising configurations upfront are thus interesting future research goals.

F. Padding

Unlike cuTENSOR, our kernels require zero-padding input tensors to ensure that the resulting matrix multiplication dimensions are multiples of the thread block tile size. The preceding evaluations do not account for the overhead introduced by this padding requirement. This section analyses how padding affects both run time performance as well as memory consumption, and discusses potential strategies to eliminate the padding requirement altogether.

Figure 6 shows the impact of padding on memory consumption. The top graph presents the memory overhead at the individual tensor level, for all 48 TCs on Turing and Ada Lovelace GPUs. It treats each individual tensor as a separate data point, i.e each TC $\mathcal{C} \leftarrow \mathcal{A} \cdot \mathcal{B} + \mathcal{C}$ contributes three distinct samples to the distribution. This approach reveals how padding affects each tensor independently, showing the variation in overhead across all tensors in the TCCG benchmark suite, and across different GPUs. Across the entire benchmark suite, 27% of tensors require no padding. 54% of tensors experience a modest padding overhead, ranging from +0.93% to +33%. The remaining 19% of tensors incurred a more substantial memory overhead, with storage increases from +37% to +255%.

The per-tensor memory overhead can thus be quite significant in some cases. However, these large relative storage overheads only occur for tensors that have dimensions of small extent, and for which a limited amount of such dimensions are mapped to a single GEMM dimension. For a given TC, this can only occur for a single tensor, that also has the smallest overall size of all tensors involved in the TC. The other, bigger tensors will require less padding relative to their own size, but because they are larger, they will dominate the overall memory consumption in absolute terms. For example, consider TC3 abcd-dbea-ec, with unpadded extents $(a, b, c, d, e) = (72, 72, 24, 72, 72)$. With a tile size $(M, N, K) = (128, 64, 32)$, the $\mathcal{B}$-tensor requires padding $K = e$ from 72 to 96, and $N = c$ from 24 to 64, yielding a substantial +255.55% overhead. On the other hand, the $\mathcal{A}$-tensor only faces a +33.33% overhead (as a result of padding $K = e$ from 72 to 96), and the $\mathcal{C}$-tensor a +166.66% overhead (due to padding N). However, since the substantially padded tensor $\mathcal{B}$'s contribution to the overall memory consumption is negligible, the overall additional memory consumption is dominated by $\mathcal{A}$ and $\mathcal{C}$, which have lower padding overhead. Thus, the total extra memory consumption due to padding is actually much less than displayed in the top graph of Figure 6.

To evaluate the total memory impact of padding per TC, the bottom graph in Figure 6 presents the memory overhead at the TC level, treating each TC as a single data point. Here, the overhead is calculated by summing the padded sizes of all three tensors and dividing by the sum of their unpadded sizes. This provides a more holistic view of the memory overhead of padding, reflecting the total additional memory required to execute each tensor contraction. The per-TC overheads range from +0% to +77.8%, significantly lower than the per-tensor overheads. 9% of TCs require no padding for any of the input or output tensors. Half of the TCs have a per-TC padding overhead below 4%, and 72% of TCs maintain overheads lower than +10%. These results demonstrate that although individual small tensors may incur substantial padding overhead, the overall memory impact on complete TC operations remains relatively low.

Figure 7 shows the run time impact of padding on our four newest GPUs. It shows the CDF of the performance of our kernels relative to cuTENSOR. Each subplot shows all four combinations of including/excluding the padding and unpadding kernels in the run time of our kernels. The results reveal that padding can substantially affect run time, with the majority of the overhead stemming from padding input tensors rather than output tensor unpadding.

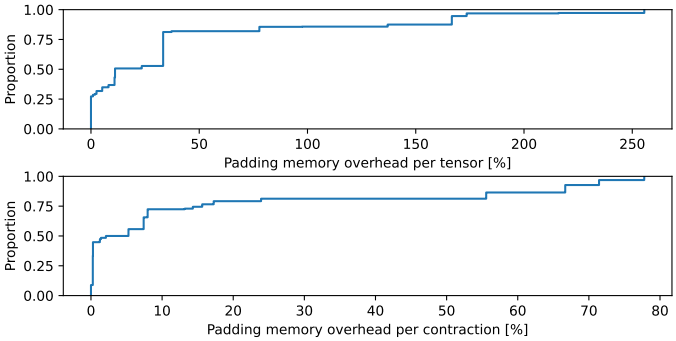

Figure 6. Cumulative distributions of the relative memory overhead of padded vs. unpadded tensors on Turing and Ada Lovelace. The top graph shows the overhead distribution per tensor, i.e. each TC $\mathcal{C} \leftarrow \mathcal{A} \cdot \mathcal{B} + \mathcal{C}$ results in three samples $\frac{|\mathcal{A}_{\text{padded}}|}{|\mathcal{A}_{\text{unpadded}}|}$, $\frac{|\mathcal{B}_{\text{padded}}|}{|\mathcal{B}_{\text{unpadded}}|}$, $\frac{|\mathcal{C}_{\text{padded}}|}{|\mathcal{C}_{\text{unpadded}}|}$. The bottom graph shows the distribution of the overhead per tensor contraction, i.e. each tensor contraction only results in a single sample $\frac{|\mathcal{A}_{\text{padded}}|+|\mathcal{B}_{\text{padded}}|+|\mathcal{C}_{\text{padded}}|}{|\mathcal{A}_{\text{unpadded}}|+|\mathcal{B}_{\text{unpadded}}|+|\mathcal{C}_{\text{unpadded}}|}$.

Consequently, for TCs requiring significant padding, our library only offers improved performance over cuTENSOR in specific scenarios: when input tensors are stored pre-padded or can be generated thusly, when padding costs can be amortised across multiple TCs that reuse the same input tensors, or within computational pipelines consisting of a chain of TCs, where intermediate results remain in padded form such that padding overhead is only incurred at pipeline initialisation.

Note that we reused the TC parameters from earlier rather than conducting a separate sweep. Consequently, the parameter selection process did not prioritise configurations with reduced padding requirements to minimise padding overhead. A dedicated sweep accounting for the additional run time costs incurred by padding could potentially yield improved results. Additionally, our padding and unpadding kernels use a naive implementation using two broadcast operations: one for data copying, and another for zeroing the padded region. Combining these into a single kernel that is additionally specialised on the extents of the involved tensor would likely result in a noticeable performance improvement, though such optimisations are beyond the scope of this work.

Our kernels require padding for two main reasons. First, to avoid handling partial tiles at tensor boundaries, all problem dimensions must be multiples of the thread block tile size. This ensures that the nested computation loops only process complete tiles, avoiding the need for epilogue logic. Second, our kernels use vectorised memory operations, which necessitate proper alignment: for 128-bit vector loads of FP16 data (8 elements per load), tensor extents and strides must maintain 128-bit alignment. Both requirements could potentially be eliminated through various approaches. For partial tile handling, techniques such as predicated loads/stores where out-of-bounds accesses are masked by thread predicates, or the Tensor Memory Accelerator (TMA) on Hopper-generation GPUs, which provides hardware-accelerated boundary handling, allow kernels to process tensors of arbitrary dimension without padding. For alignment requirements, hybrid loading schemes could be used that combine vectorised loads for aligned addresses with scalar loads for the remaining elements.

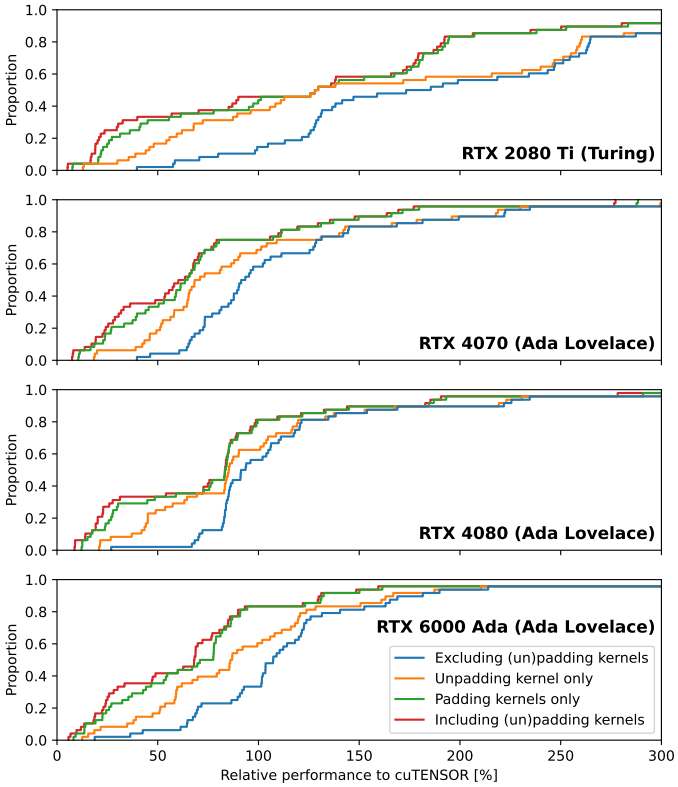

Figure 7. Cumulative distributions of TC run times across four GPUs of two generations, showing performance relative to the cuTENSOR baseline. Each of the four subplots shows four configurations: (1) our kernels excluding both padding and unpadding operations, (2) our kernels with unpadding the output tensor $\mathcal{C}$ only, (3) our kernels with padding the input tensors $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ only, and (4) our kernels including both padding and unpadding.

However, implementing these padding-elimination techniques introduces additional code complexity, and potential performance regressions due to the additional control flow overhead, placing them beyond the scope of the current work.

G. Holistic Performance Analysis

Section IV-C analysed the performance of TCs with outlying performance. This section focuses on a more holistic performance analysis across all TCs to answer two questions: (1) Which aspects of our kernels and cuTENSOR's explain our performance improvement or regression, and (2) For which types of TCs (extents, aspect ratio, padding requirements) should an end user prefer our library over cuTENSOR.

For the first question, we ran all TCs under the Nsight Compute profiler [45] on all our GPUs, gathering all metrics for both libraries, with the best parameters found during the sweep. We calculated the correlation of metrics and the $\frac{\text{ours}}{\text{cuTENSOR}}$-ratios on the one hand, with the run times and their ratios on the other hand. We use the Pearson correlation p to detect linear relationships, and the Spearman correlation s to uncover monotonic relationships. We looked at metrics with the highest maximum absolute correlation $m = \max(|p|, |s|)$, both overall across all GPUs, as well as individually per GPU. We filter out metrics that are by definition (in)directly correlated with run time, such as number of elapsed cycles.

The results differ quite significantly over different GPU architectures. On the older architectures (Volta and Turing), the highest correlation (around 0.75) occurred for metrics related to "no instruction stalls". On these architectures, 80% of cuTENSOR's kernels suffer more from these types of stalls than our kernels. These stalls occur when warps are stalled waiting for an instruction to be fetched, e.g. due to instruction cache misses. Potential reasons for this could be a larger instruction footprint, worse instruction locality due to excessive branching, or more divergence between threads. To exclude that this could be caused by cuTENSOR's kernels having to include extra logic to handle the non-padded layouts (which would bias the performance comparison in Section IV-A), we re-ran cuTENSOR's kernels with padded inputs. With the exception of TC29 on the RTX 2080 Ti, all TCs remained equally performant or suffered a slowdown on cuTENSOR for padded inputs. This hints that the potentially extra logic due to unpadded inputs is not the reason for the extra stalls.

On Ada Lovelace, these same metrics only reach correlations of 0.18 – 0.39. For these GPUs, the most correlated metrics (0.68 – 0.89) are related to shared memory operations. Our kernels typically have more shared memory instructions and memory requests to the load-store unit than cuTENSOR's, and are stalled much more often on memory dependencies. Based on the distribution of bank conflicts over different GPU generations, it is clear that this cannot stem from an increased number of bank conflicts. Instead, integrating support for hardware features that perform global-to-shared memory copies more efficiently, such as Ampere's asynchronous copy, are integral to improve performance on Ada Lovelace.

For the second question, we perform a similar correlation analysis, now with metrics related to the TC's aspect ratio and extents. The considered metrics include the M, N, K dimension and arithmetic intensity $\frac{MNK}{MN+NK+KN}$ of the equivalent GEMM and, for each combination x and y of two GEMM-dimensions, their product xy, their quotients $\frac{x}{y}$ and $\frac{y}{x}$, and their "squareness" $\max(\frac{x}{y}, \frac{y}{x})$. We calculated these metrics using both the unpadded and the padded extents. We also include the padding ratios $\frac{x_{\text{padded}}}{x_{\text{unpadded}}}$ per dimension x, and per tensor.

Generally, the unpadded variant of a metric has a slightly higher correlation than the padded one, both across all GPUs and for each GPU individually. This, in combination with the relatively low correlations of padding ratio metrics, aligns with our intuition that the padding overhead is a worse predictor for relative performance than the order of magnitude of the dimensions. Still, due to the need for (un)padding kernels, for TCs requiring padding, our library should only be preferred over cuTENSOR in the cases discussed in Section IV-F.

Furthermore, K and $M \cdot N$ are the metrics most correlating with relative performance. The higher $M \cdot N$, the better our kernels perform compared to cuTENSOR. This makes sense, since the underlying GEMM kernels in GemmKernels are designed to exploit the massive parallelism of large matrices, and contrary to cuTENSOR do not include optimisations for small matrices. Surprisingly, a higher value of K corresponded to worse performance of GemmKernels compared to cuTENSOR. On first sight, this would indicate that one should prefer our kernels over cuTENSOR for small K only. This is unexpected because of the above observation on GemmKernels' optimisation for large, not small matrices, and because large K increases the number of stages and thus the efficiency of our kernels through software pipelining.

This spurious correlation turned out to be due to the way the tensor extents are chosen in the TCCG benchmark suite. In our benchmark suite, the tensor extents are chosen in such a way that the resulting tensors are roughly the same size for all TCs. This means that a TC with low K will inevitably be associated with a high M and/or N, increasing parallelism. Indeed, the K dimension and the product $M \cdot N$ have a large negative Spearman correlation of -0.80. Additionally, most TCs in TCCG have a relatively low K: 69% of TCs have a K that only allows at most three software pipelining stages. Thus, the reason that increasing K lowers the relative performance of our kernels, is because this decreases $M \cdot N$, and thus the potential parallelism of the GEMM kernel. The increase in K should in theory also result in higher performance due to the increase in pipeline stages, but this effect is not noticeable in the benchmark suite due the involved K-values being so low so as to not make much of a difference.

V. Discussion

High-level languages like Julia boost productivity by abstracting away low-level details such as memory management. Programmers must still understand compiler optimisations, however, as minor source code changes can dramatically affect generated machine code and performance. This requirement undermines productivity gains and necessitates low-level understanding even when using high-level languages.

A. Compiler Issues

In our research, we have discerned two main categories of such issues. The first category involves thresholds used in the compiler's cost functions and static analyses. These thresholds sometimes prevented optimisations like loop unrolling or inlining when exceeded, and caused compilation failures when static array sizes hit heuristic limits in type inference.

The second category concerns the phase ordering problem in compilers. Modern compilers apply transformations sequentially, but these can be destructive—one optimisation may prevent others from being applicable. While compiler developers optimise the transformation order using sample programs, we encountered cases where this ordering caused missed opportunities. For instance, when explicitly vectorising memory operations for FP16 Tensor Core operations in our kernels, we found that type changes introduced by LLVM's instcombine pass reduced vectorisation from 128-bit to 64-bit loads because the NVPTX back-end limits i16-vector loads differently from half-vector loads.

These two categories are not mutually exclusive. While experimenting with shared memory swizzling, we compared the generated code for two kernel versions. In one version, we had manually restructured the code, implementing a form of software pipelining to prefetch data during the loading from shared memory. As data then gets loaded and operated on in different iterations, this restructuring resulted in slightly

longer data flow paths in the generated LLVM IR. The data and address computations themselves did not change. The compiler middle-end's known-bits analysis was not hampered by the restructuring, so for both versions, it converted additions to bitwise ORs for canonicalisation, assuming this might enable further optimisations. The back-end featured its own known-bits analysis, however, and this one imposed a threshold limit on how deep information is propagated from producers to consumers in the IR. Because of this limit, the back-end analysis lacked sufficient precision to convert the ORs back to additions, preventing the use of efficient register-plus-immediate addressing modes, resulting in explicit ADD and OR instructions instead of optimised memory access patterns and in increased register pressure. Ultimately, this caused a large performance hit that could not have been explained without detailed knowledge or analysis of the compiler's intricacies and its sensitivity to code structure.

B. Underlying Reasons

Compiler developers set thresholds and optimisation orders by tuning them on what they consider and collect as representative program sets, optimising for compilation time and code quality. However, these settings may not generalise well beyond the original tuning set. Our kernels likely differ from typical tuning programs for two reasons. First, tuning sets usually contain manually written code fragments rather than automatically generated kernels. Second, dense linear algebra kernels are often written in assembly to bypass compilers entirely, making them less relevant to compiler developers and unlikely to be included in tuning sets. Consequently, compilers are poorly optimised for the patterns in our automatically generated kernels. This problem is amplified in high-level languages like Julia, which involve complex compilation pipelines spanning multiple projects—from Julia-to-LLVM compilation through LLVM middle-end passes to NVPTX backend code generation. Each component is maintained by different teams with distinct priorities and tuning sets, making it highly unlikely that our kernel patterns have been tested across the entire compilation flow.

C. Consequences for Flexible Library Developers and Users

Compiler sensitivity to code structure poses unique challenges for flexible library developers. Unlike fixed libraries where developers can verify all kernels before shipping binaries, flexible libraries generate kernels based on user configurations, making comprehensive pre-verification impossible.

The discussed issues complicate automatic tuning by introducing performance discontinuities from minor code changes. This noise hinders tuning algorithms from learning the kernel parameter-to-performance mapping. Even though some models such as Bayesian optimisation [48] can deal with discontinuities, the flakiness still seems hard for black-box optimisation algorithms to deal with if it becomes too prevalent.

Paradoxically, autotuning might help detect flakiness. Major deviations between predicted and measured performance could potentially signal missed optimisations, alerting developers to investigate. Verifying this hypothesis is future work.

One can argue whether eliminating this flakiness is the responsibility of the library users or of its developers. Users will likely have to verify their specific kernel instantiations, increasing usage complexity and expertise requirements compared to fixed libraries. Yet library developers should also bear some responsibility, by designing with compiler behaviour in mind, potentially trading flexibility for optimisation reliability.

For the flakiness as a result of thresholds, a potential solution might be for the library developers to make use of attributes such as `clang::always_inline` [49] (to force inlining, ignoring the inlining cost), or metadata such as `!{!"llvm.loop.unroll.full"}` [50] (which forces full loop unrolling). We think two requirements need to be met for this to be viable. First, the library developers have to be sure that the used attribute or metadata is not just a (strong) hint to the compiler (such as the `inline` keyword), but actually forces the optimisation (such as the `clang::always_inline` attribute; although even that attribute does not guarantee in *all* cases that inline substitution actually occurs [49]). Second, there needs to be some feedback (in terms of a compiler diagnostic such as a warning or error) to the user of the library when a given optimisation that was requested, was not applied for some reason. Ideally, the library developers can rely on the compiler for this. For example, when Clang notices a `clang::always_inline` attribute in a context where it is not applicable, it emits a warning.

VI. Related Work

In 2018, Springer and Bientinesi classified the state of the art for TCs in three categories [5]. Loop nesting approaches apply loop optimisations such as loop fusion and unrolling to the mathematical formulation of TCs [9], [10], [11]. Loop-over-GEMM approaches decompose TCs into a sequence of two-dimensional matrix multiplications, for which a GEMM kernel is re-used [8], [12]. TTGT also re-uses GEMM kernels, but combines them with tensor transposition kernels such that only one GEMM invocation suffices per TC. Several TC frameworks use TTGT, such as the Cyclops Tensor Framework [13], Tensor Toolbox [14], Tensorlab [15], and libtensor [16]. While these three approaches perform well for some TCs, their performance suffers on certain classes of TCs due to strided accesses, additional kernel launch overhead, or extra storage requirements. Springer and Bientinesi's alternative GETT does not suffer from these issues [5]. GETT relies on the ability of flexible GEMM kernels to customise the storage layout of the matrices to perform arbitrary TCs without extra transposition kernels. It has been adopted by later work as well [6]. Independently, Matthews proposed TBLIS [17], which also reuses flexible GEMM kernels for TCs. It builds on the BLIS framework's components to instantiate BLAS kernels and variants thereof [51]. Contrary to our work, TBLIS considers only TCs on CPUs.

NVIDIA's cuTENSOR library builds on the principles of GETT, but focuses on TCs on NVIDIA GPUs [3] and exploiting the latest hardware features such as Tensor Cores. cuTENSOR relies on the flexible kernels in NVIDIA's CUTLASS library [52], which contains a large set of C++ templates

that can be combined and customised arbitrarily to instantiate GEMM-like kernels. While CUTLASS is open-source, the source code of cuTENSOR is not (yet) available. As discussed in the introduction, this is problematic in cases requiring flexibility beyond the extent that cuTENSOR offers.

Recent CUTLASS releases incorporate CuTe, a C++ template library for kernel development for multidimensional data. Developers can combine CUTLASS and CuTe components to implement general TCs [53]. However, this approach still faces significant limitations compared to our work: the current implementation relies heavily on the Tensor Memory Accelerator (TMA) for address arithmetic, restricting compatibility to Hopper GPUs and excluding older generations. Furthermore, despite CUTLASS and CuTe being open-source, customisation requires deep understanding of their complex template machinery, which can present a substantial barrier to developers seeking to adapt these tools for specific TC computations or alternative hardware targets.

Compiler-based approaches such as XLA [54], TVM [55], and others built on infrastructures like MLIR [56], along with specialised tools like Triton [57] and TACO [58], have emerged for optimising TCs on GPUs. Some, like Triton [57], operate on similar abstractions as our work (individual fusion-capable kernels), but differ in their approach: Triton provides a Python DSL to express kernels that are then compiled, whereas our work provides a flexible library of pre-structured kernel templates with configurable fusion capabilities. Both approaches support autotuning, but target different use cases: Triton enables custom kernel development with reduced programming complexity, while our templates provide high-performance solutions for standard TC patterns without requiring kernel programming. Others, such as TVM [55], are end-to-end ML compilers, and optimise entire ML models represented by a computational graph. After graph-level optimisations each node is lowered to efficient kernels through techniques that automatically handle the complexities of memory hierarchy, thread scheduling, and instruction-level optimisations. Like our work, they support auto tuning and operator fusion. However, they differ in scope (ML models' entire pipelines of TCs vs. a single TC) and abstraction level (computational graph vs. a single TC node, possibly fused with an elementwise operation), and thus feature fundamentally different design spaces. Since these compilers are primarily accessed through ML frameworks like TensorFlow and PyTorch, they are predominantly tuned for standard ML operations, and may exhibit the same sensitivity issues outlined in Section V, where minor modifications to the source code can lead to dramatically different machine code and hence performance. Additionally, some of these systems employ aggressive optimisations that impact numerical accuracy [59], which would likely disqualify them for scientific computations.

The techniques that lower computational nodes to efficient kernels complement or replace pre-optimised, vendor-provided libraries where those libraries lack the necessary support for arbitrary operation fusion [55]. Our work offers the necessary flexibility, however, and could be used as an alternative. For example, TVM supports different types of fusion, some of which could be achieved using our kernels [55]: fusing TCs with injective nodes (i.e. elementwise operations) is supported through the use of GemmKernels's transform abstraction, and fusing TCs with reductions can be achieved with a custom epilogue. By combining TVM and our TC kernels, one could benefit from the holistic optimisation offered by these frameworks, and from the improved numerical accuracy and potentially improved performance of our kernels. Additionally, TVM's global optimisation takes into account preferred data layouts for each node in the computational graph, and finds a globally optimal assignment of layouts, inserting the necessary layout transformations between producers and consumers with mismatching layout requirements. This could be utilised to find a global optimum taking into account the run time overhead of the necessary (un)padding kernels.

More recent work on GPU tensor computations focuses mostly on optimising specific (classes of) computations in a certain scientific domain [40], [41], but flexibility remains largely unaddressed. To the best of our knowledge, our work is the first that discusses flexibility and its impact on performance and the productivity of library developers and end users.

VII. Availability

Our artefacts are released under an open-source license as a branch in the GemmKernels.jl package at https://github.com/JuliaGPU/GemmKernels.jl/tree/tensor-contractions. This includes our extensions to support tensor contractions, our new GETT-related components, and the scripts used for autotuning and evaluation on the TCCG benchmark suite.

VIII. Conclusion and Future Work

We presented the requirements, design, and implementation of a flexible, performant TC library in Julia built on the GemmKernels library. We discussed optimisations and necessary changes to GemmKernels, our kernel design space, and how we select the kernel to be used for each TC. After illustrating the library's usage on a practical example, we conducted a comprehensive performance and flexibility evaluation and analysis against NVIDIA's state-of-the-art cuTENSOR library. It shows that our kernels achieve on average higher or comparable performance, often matching and occasionally surpassing cuTENSOR's speed, while avoiding the flexibility constraints that limit cuTENSOR's general applicability. We discussed the impact of the compiler on programmer and end-user productivity in the context of flexible libraries and high-level programming languages, and proposed potential alleviations.

We see several promising avenues for extending this work. First, we have yet to evaluate our kernels on more recent GPU architectures, such as Hopper or Blackwell, which could reveal additional optimisation opportunities. Secondly, while our kernels demonstrate strong overall performance, there are still some TCs or architectures for which they perform poorly. Follow-up work could explore GPU-architecture-specific optimisations such as Volta's `mma.sync` combined with shared memory swizzling [19], Ampere's asynchronous copy [20], or Hopper's Tensor Matrix Accelerators [21]. Such optimisations can have a large impact on performance. For example, on

the V100, using `mma.sync` instructions instead of WMMA has resulted in speedups of $1.65\times$ and $1.73\times$ for GEMM and transformer networks, respectively [19]. Problem-specific optimisations such as Split-K or Sliced-K [46], or predicated loads to avoid the need for padding also warrant exploration.

Finally, our current random-sampling-approach to kernel parameter selection remains suboptimal given the enormous search space involved. One potential solution is to use a performance model to either select the optimal kernel parameters, or to prune the search space, as done by some TC libraries [5], [60]. Other possibilities include using machine learning models such as decision trees or neural networks, which have shown promise in the literature for tuning GEMM kernels [61], and could potentially be adapted for TCs.

Acknowledgements

This work was funded by the Research Foundation Flanders (FWO), grant number 11I1123N. The authors also thank Nora Dossche for her valuable feedback.

References

[1] NVIDIA. 'cuBLAS: Basic linear algebra on NVIDIA GPUs.'

[2] NVIDIA. 'cuDNN: CUDA deep neural network.'

[3] NVIDIA. 'cuTENSOR: Tensor linear algebra on NVIDIA GPUs.'

[4] T. Faingnaert, T. Besard and B. De Sutter, 'Flexible performant GEMM kernels on GPUs,' *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 9, pp. 2230–2248, 2022.

[5] P. Springer and P. Bientinesi, 'Design of a high-performance GEMM-like tensor–tensor multiplication,' *ACM Trans. Math. Softw.*, vol. 44, no. 3, pp. 1–29, 2018.

[6] J. Kim et al., 'A Code Generator for High-Performance Tensor Contractions on GPUs,' in *Proc. CGO*, IEEE, Feb. 2019, pp. 85–95.

[7] J. Chen, R. G. Edwards and W. Mao, 'Graph Contractions for Calculating Correlation Functions in Lattice QCD,' in *Proc. PASC*, ACM, 2023, pp. 1–10.

[8] E. Di Napoli et al., 'Towards an efficient use of the BLAS library for multilinear tensor contractions,' *Appl. Math. Comput.*, vol. 235, pp. 454–468, May 2014.

[9] E. Apra et al., 'Efficient Implementation of Many-Body Quantum Chemical Methods on the Intel Xeon Phi Coprocessor,' in *Proc. CS*, IEEE, 2014, pp. 674–684.

[10] W. Ma et al., 'GPU-Based Implementations of the Noniterative Regularized-CCSD(T) Corrections: Applications to Strongly Correlated Systems,' *J. Chem. Theory Comput.*, vol. 7, no. 5, pp. 1316–1327, 10th May 2011.

[11] T. Nelson et al., 'Generating Efficient Tensor Contractions for GPUs,' in *2015 44th International Conf. on Parallel Processing*, IEEE, Sep. 2015, pp. 969–978.

[12] J. Li et al., 'An input-adaptive and in-place approach to dense tensor-times-matrix multiply,' in *Proc. CS*, ACM, 2015, pp. 1–12.

[13] E. Solomonik et al., 'Cyclops Tensor Framework: Reducing Communication and Eliminating Load Imbalance in Massively Parallel Contractions,' in *Proc. IPDPS*, IEEE, 2013, pp. 813–824.

[14] B. W. Bader and T. G. Kolda, 'Algorithm 862: MATLAB tensor classes for fast algorithm prototyping,' *ACM Trans. Math. Softw.*, vol. 32, no. 4, pp. 635–653, 2006.

[15] N. Vervliet et al. 'Tensorlab 3.0. '[Online]. Available: https://www.tensorlab.net.

[16] E. Epifanovsky et al., 'New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations,' *J. Comput. Chem*, vol. 34, no. 26, pp. 2293–2309, 5th Oct. 2013.

[17] D. A. Matthews, 'High-Performance Tensor Contraction without Transposition,' *SIAM J. Sci. Comput.*, vol. 40, no. 1, pp. C1–C24, Jan. 2018.

[18] NVIDIA. 'CUDA C++ programming guide.'

[19] A. Kerr et al., 'Programming Tensor Cores: Native Volta Tensor Cores with CUTLASS,' NVIDIA GPU Technology Conference, 2019.

[20] A. Kerr, 'Developing CUDA Kernels to Push Tensor Cores to the Absolute Limit on NVIDIA A100,' NVIDIA GPU Technology Conference, 21st May 2020.

[21] P. Ramani and C. Cecka, 'Developing Optimal CUDA Kernels on Hopper Tensor Cores,' 22nd Mar. 2023.

[22] NVIDIA. 'CUDA C programming guide – features and technical specifications.'

[23] C. Cecka and M. Awatramani, 'Programming Blackwell Tensor Cores with CuTe and CUTLASS,' 2025.

[24] J. Bezanson, A. Edelman, S. Karpinski and V. B. Shah, 'Julia: A Fresh Approach to Numerical Computing,' *SIAM Rev.*, vol. 59, no. 1, pp. 65–98, Jan. 2017.

[25] C. Lattner and V. Adve, 'LLVM: A compilation framework for lifelong program analysis & transformation,' in *Proc. CGO*, IEEE, 2004, pp. 75–86.

[26] T. Besard et al., 'Effective Extensible Programming: Unleashing Julia on GPUs,' *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 4, pp. 827–841, 2019.

[27] NVIDIA. 'cuTENSOR user guide.'

[28] D.-A. Clevert, T. Unterthiner and S. Hochreiter. 'Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs).' arXiv: 1511.07289 [cs].

[29] A. L. Maas, A. Y. Hannun, A. Y. Ng et al., 'Rectifier nonlinearities improve neural network acoustic models,' in *Proc. ICML*, Atlanta, GA, vol. 30, 2013, p. 3.

[30] NVIDIA. 'CUTLASS documentation: Threadblock rasterization.'

[31] *Hyperopt.jl*, 2022. [Online]. Available: https://github.com/baggepinnen/Hyperopt.jl.

[32] M. D. McKay et al., 'A comparison of three methods for selecting values of input variables in the analysis of output from a computer code,' *Technometrics*, vol. 42, no. 1, pp. 55–61, 2000.

[33] L. Li et al., 'Hyperband: A novel bandit-based approach to hyperparameter optimization,' *Journal of Machine Learning Research*, vol. 18, no. 185, pp. 1–52, 2018.

[34] S. Falkner et al., 'BOHB: Robust and efficient hyperparameter optimization at scale,' in *Int'l Conf. on Machine Learning*, PMLR, 2018, pp. 1437–1446.

[35] M. E. Ozturk et al., 'A Performance Portability Study Using Tensor Contraction Benchmarks,' in *Proc. IPDPS Workshops*, 2023, pp. 591–600.

[36] A. Olivry et al., 'IOOpt: Automatic derivation of I/O complexity bounds for affine programs,' in *Proc. International Conf. on Programming Language Design and Implementation*, 2021, pp. 1187–1202.

[37] R. Li et al., 'Analytical cache modeling and tilesize optimization for tensor contractions,' in *Proc. CS*, ACM, 2019, pp. 1–13.

[38] K. Stock et al., 'Using machine learning to improve automatic vectorization,' *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 1–23, Jan. 2012.

[39] G. Baumgartner et al., 'Synthesis of High-Performance Parallel Programs for a Class of ab Initio Quantum Chemistry Models,' *Proc. of the IEEE*, vol. 93, no. 2, pp. 276–292, Feb. 2005.

[40] R. Hu et al., 'BCB-SpTC: An Efficient Sparse High-Dimensional Tensor Contraction Employing Tensor Core Acceleration,' *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 12, pp. 2435–2448, Dec. 2024.

[41] X.-Y. Liu et al., 'High-Performance Tensor Learning Primitives Using GPU Tensor Cores,' *IEEE Trans. Comput.*, vol. 72, no. 6, pp. 1733–1746, 1st Jun. 2023.

[42] J. Finkelstein et al., 'Quantum Perturbation Theory Using Tensor Cores and a Deep Neural Network,' *J. Chem. Theory Comput.*, vol. 18, no. 7, pp. 4255–4268, 2022.

[43] J. Finkelstein et al., 'Quantum-Based Molecular Dynamics Simulations Using Tensor Cores,' *J. Chem. Theory Comput.*, vol. 17, no. 10, pp. 6180–6192, 2021.

[44] NVIDIA, *NVIDIA V100S datasheet*, 2020.

[45] NVIDIA, *Nsight Compute*, 2025.

[46] NVIDIA, *Efficient GEMM in CUDA: Parallelized Reductions*, 2025.

[47] V. Nair and G. E. Hinton, 'Rectified Linear Units Improve Restricted Boltzmann Machines,' 2010.

[48] J. Mockus, 'The bayesian approach to local optimization,' in *Bayesian approach to global optimization: Theory and applications*, Springer, 1989, pp. 125–156.

[49] Clang team. 'Attributes in Clang: Clang documentation.'

[50] Clang team. 'Code transformation metadata.'

[51] F. Van Zee and R. Van De Geijn, 'BLIS: A Framework for Rapidly Instantiating BLAS Functionality,' *ACM Trans. Math. Softw.*, vol. 41, no. 3, pp. 1–33, 2015.

[52] NVIDIA. 'CUDA templates for linear algebra subroutines.'

[53] NVIDIA. 'CUTLASS example 51: Hopper GETT.'

[54] Google. 'OpenXLA project.'

[55] T. Chen et al., 'TVM: An automated End-to-End optimizing compiler for deep learning,' in *Proc. OSDI*, USENIX, Oct. 2018, pp. 578–594.

[56] C. Lattner et al., 'MLIR: Scaling Compiler Infrastructure for Domain Specific Computation,' in *Proc. CGO*, 2021.

[57] P. Tillet et al., 'Triton: An intermediate language and compiler for tiled neural network computations,' in *Proc. Int'l Workshop on Machine Learning and Programming Languages*, 1029, pp. 10–19.

[58] F. Kjolstad et al., 'The tensor algebra compiler,' *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, 77:1–77:29, 2017.

[59] Z. Xia et al., 'Detecting numerical deviations in deep learning models introduced by the tvm compiler,' in *Proc. ISSRE*, 2024, pp. 73–83.

[60] A. Abdelfattah et al., 'High-performance Tensor Contractions for GPUs,' *Procedia Computer Science*, vol. 80, pp. 108–118, 2016.

[61] Y. Yu et al., 'Tailoring CUTLASS GEMM using Supervised Learning,' in *International Conf. on Computer Design (ICCD)*, 2023, pp. 465–474.

Thomas Faingnaert is a PhD student at Ghent University in the Computer Systems Lab. He obtained his MSc degree in Computer Science Engineering from Ghent University's Faculty of Engineering and Architecture in 2020. His research focuses on software protection, and high-level abstractions for GPU programming in Julia.

Ward Vermeulen currently works at TechWolf. He obtained his MSc degree in Computer Science Engineering from Ghent University's Faculty of Engineering and Architecture in 2023. His master thesis research project focused on high-level abstractions for tensor contractions in Julia.

Tim Besard is a software engineer at JuliaHub. He obtained his MSc in Computer Engineering from University College Ghent in 2011, and his PhD in Computer Science Engineering from Ghent University in 2019. He is currently the lead maintainer of several GPU back-ends for the Julia programming language.

Bjorn De Sutter is full professor at Ghent University in the Computer Systems Lab. He obtained his MSc and PhD degrees in Computer Science from Ghent University's Faculty of Engineering in 1997 and 2002. His research focuses on the use of compiler techniques to aid programmers with non-functional aspects of their software, such as performance and mitigation of various security threats.