A Code Generator for High-Performance Tensor Contractions on GPUs

Jinsung Kim*, Aravind Sukumaran-Rajam*, Vineeth Thumma*, Sriram Krishnamoorthy†,
Ajay Panyala†, Louis-Noël Pouchet‡, Atanas Rountev*, P. Sadayappan*
*The Ohio State University, Ohio, USA
{kim.4232, sukumaranrajam.1, thumma.6, rountev.1, sadayappan.1}@osu.edu
†Pacific Northwest National Laboratory, Washington, USA
{sriram, ajay.panyala}@pnnl.gov
‡Colorado State University, Colorado, USA
pouchet@colostate.edu

***Abstract*—Tensor contractions are higher dimensional generalizations of matrix-matrix multiplication. They form the compute-intensive core of many applications in computational science and data science. In this paper, we describe a high-performance GPU code generator for arbitrary tensor contractions. It exploits domain-specific properties about data reuse in tensor contractions to devise an effective code generation schema, coupled with an effective model-driven search, to determine parameters for mapping of computation to threads and staging of data through the GPU memory hierarchy. Experimental evaluation using a set of tensor contraction benchmarks demonstrates performance improvement and/or significantly reduced code generation time over other state-of-the-art tensor contraction libraries and code generators.**

***Index Terms*—Code Generation, Tensor Contractions, GPU Computing**

I. Introduction

Tensors, higher dimensional analogs of matrices, represent a key data abstraction for many applications in computational science and data science [1]–[7]. Tensor contractions are higher dimensional analogs of matrix-matrix multiplication. In comparison to matrix multiplication, a significant challenge in creating an efficient library for tensor contractions is the wide range of usage scenarios to be considered. For matrix-matrix multiplication, library implementations need to only address four cases: each of the input matrices may be specified as non-transposed or transposed for the matrix product. This corresponds to the two possible index positions for the summation index in the two input tensors: normal non-transposed matrix multiplication has the summation index in the second (column) index for the left matrix and the first (row) index for the right matrix. For contractions of two 3D tensors, and a single contraction (summation) index, there are 846 ($3! \times 3! \times 4!$) possible cases. If the contraction has two contraction indices, there are 72 cases. With higher dimensional tensors, the number of cases grows exponentially. Tensors of up to six dimensions are commonly encountered in ab initio quantum chemistry models such as the CCSD(T) coupled cluster method [8].

Artifact available at: https://doi.org/10.6084/m9.figshare.7403732

The common solution in use today for performing tensor contractions is to first perform suitable index permutation, i.e., layout transformation of one or both input tensors, to move all contraction indices to the left end or the right end, so that the needed sums of products for the contraction can be performed using matrix-matrix multiplication. A further final index permutation of the result from the matrix multiplication may also be required. However, as elaborated in the next section, this approach, referred to as the *TTGT* (Transpose-Transpose-GEMM-Transpose), incurs several disadvantages.

In this paper, we present an approach to perform *direct* contraction of tensors on GPUs without the creation of any transposed intermediate matrices. In addition to reducing memory usage by avoiding the use of intermediate tensors, we show that this approach results in improved performance. We employ a code generation strategy that utilizes properties about reuse directions in the iteration space of a multi-dimensional tensor contraction to devise an effective strategy for mapping of computations and orchestration of data movement through the GPU memory hierarchy. The code generation schema is designed to address key performance considerations for GPUs: utilization of significant concurrency to tolerate global memory access latency, coalesced access to global memory to maximize achieved bandwidth, and maximization of data reuse within registers and shared memory. An effective cost model is developed for the amount of data movement from/to GPU global memory and is used to rapidly identify mapping parameters from among a large set of alternatives.

The quality of the generated GPU code is assessed on the 48 tensor contractions in the TCCG benchmark suite [9], which has been used in other recent studies. We compare the achieved performance against TAL_SH [10], a state-of-the-art tensor contraction framework for GPUs that uses the TTGT approach, and with several recently developed CPU-based tensor contraction frameworks. We demonstrate significant performance improvement over available alternatives. On an Nvidia Volta V100 machine, we achieve up to 5.1× speedup, with a geometric mean of 1.7×, over the NWChem code generator for tensor contractions, and up to 19.3× speedup, with a geometric mean of 4.4×, over TAL_SH. On an Nvidia

978-1-7281-1436-1/19/$31.00 © 2019 IEEE

Pascal P100 machine we achieve up to 4.0× speedup, with a geometric mean of 1.69×, over NWChem's code generator, and up to 13.7× speedup, with a geometric mean of 4.0×, over TAL_SH.

II. Background and Motivation

Consider the following contraction of two 4D tensors to produce another 4D tensor:

$$C[a,b,c,d] = A[a,e,b,f] * B[d,f,c,e] \quad (1)$$

The notation for tensor contractions uses the so-called Einstein index convention, whereby indices that do not appear in the left-hand side tensor are all summation (contraction) indices. So the above contraction represents the computation:

$$C[a,b,c,d] = \sum_{e,f} A[a,e,b,f] * B[d,f,c,e]$$

Indices e and f are the *contraction* indices. The remaining indices (a,b,c,d) appear in the left-hand tensor and are called the *external* indices. This computation can be performed by first creating two temporary tensors TA and TB by performing index permutation as follows:

$$TA[a,b,e,f] = A[a,e,b,f];\ TB[e,f,c,d] = B[d,f,c,e]$$

Consecutive indexes in a tensor can be logically treated as a single virtual index with range equal to the product of the ranges of the original tensor indices. Thus, the 4D tensor $TA[a,b,e,f]$ may be equivalently viewed as a 2D matrix $MA[i,j]$, where the range of index i in the matrix MA is the product of the ranges of a and b in the tensor TA, and j has a range equal to the product of the ranges of e and f. Similarly, $TB[e,f,c,d]$ may be viewed as $MB[j,k]$, where index k in MB is a virtual index combining c and d in TB. MA and MB can be efficiently multiplied using a high-performance library routine to produce $MC[i,k]$, which is equivalent to $TC[a,b,c,d]$.

The above approach is widely used in practice today and is often referred to as the *TTGT* (Transpose-Transpose-GEMM-Transpose) approach: perform two transposes for the input tensors, followed by GEMM (GEneral Matrix Multiplication) and a final transpose of the result. However, it has some disadvantages:

- Index permutation can incur non-trivial overheads, especially if done naively by a set of nested loops to perform the data movement, due to high-stride access to either source or destination.
- The resulting matrices may be highly rectangular with one dimension being much smaller than the other. Library matrix-multiplication routines often achieve much lower performance for such matrices than with square matrices.
- It requires extra temporary space to hold the transposed matrices.

We present an approach that address all these challenges. The tensor contraction in Eq. 1 can be implemented as a 6D nested loop over the indices $\{a,b,c,d,e,f\}$. However, such a direct implementation will suffer from low performance because of high-stride access to tensors and insufficient exploitation of data reuse. Since tensor contractions represent a sub-class of affine loop computations, polyhedral optimizing compilers like PPCG (The Polyhedral Parallel Code Generator) [11] can be used to generate transformed and optimized code for GPUs. However, the very large number of possible tiled code configurations and the simplified linear cost models incorporated in general-purpose polyhedral compilers makes it infeasible to generate code that achieves very high performance. The Tensor Comprehensions framework from Facebook [12] incorporates a powerful polyhedral optimizer and GPU code generator along with an auto-tuner to perform extensive search in the huge configuration space of GPU programs corresponding to different ways of mapping the elementary arithmetic operations in a tensor contraction onto GPU threads.

In contrast, our approach to GPU code generation for an arbitrary tensor contraction exploits a key property that is true of any arbitrary tensor contraction: each loop index variable will occur in exactly two out of the three tensors: in the above example, indices $\{e,f\}$ occur in the input tensors, $\{a,b\}$ occur in A and C, and $\{c,d\}$ occur in B and C. A consequence of this property is that each loop index represents a reuse dimension for exactly one of the three tensors, the tensor that is not indexed by it. For different iterations of that loop, exactly the same set of elements of that tensor will be repeatedly accessed. This key property allows a grouping of an arbitrary number of loop indices in a high-dimensional tensor contraction into three groups, based on the tensor for which a loop index is a reuse direction. As we explain in greater detail in the rest of the paper, the exploitation of this property enables the development of a greatly pruned configuration space and an effective domain-specific work-to-thread mapping strategy. Further, we develop a cost-model that can quantify needed data movement in the GPU's memory hierarchy for any specific mapping choice, thereby enabling a model-driven pruning of code configurations. Since the peak floating-point performance of a GPU is orders of magnitude higher than the peak memory bandwidth, effective exploitation of data reuse in registers and shared-memory is essential to achieve the best performance.

For tensors that are much larger than shared-memory and register capacity, judicious blocking (tiling) of the computation to work on slices of the multi-dimensional tensor at a time is necessary. In summary, the GPU tensor contraction problem can be formulated as follows: *Given an arbitrary tensor contraction, how should the total work be partitioned among a suitably sized collection of thread blocks and how should the data movement in the memory hierarchy be orchestrated?*

In our experimental evaluation, we contrast the significant benefits from such a domain-specific code generation strategy to that using a general-purpose polyhedral optimizer in conjunction with auto-tuning, as done by Facebook's Tensor Comprehensions framework [12].

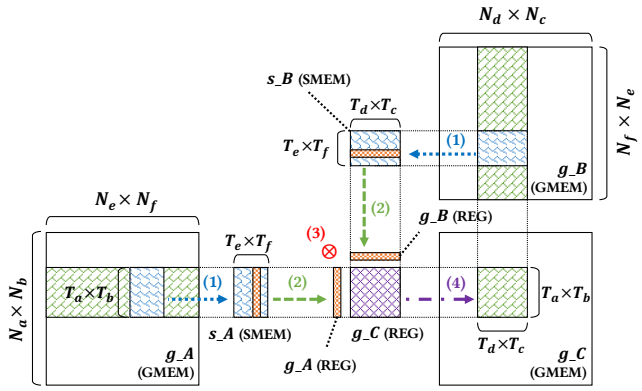

Fig. 1. Illustration of tiled execution of tensor contraction in Eq. 1

III. GPU Kernel Execution Strategy

In this section, we describe our approach to mapping a tensor contraction to GPU resources (shared memory, registers, and thread blocks). We illustrate our approach using the tensor contraction in Eq. 1. The input to the code generator is an ordered list of indices for the input and output tensors, and a representative extent (size) for each index. While the generated code will execute correctly for any problem size, a representative problem size is used for performance modeling that drives specific choice of tilesize/blocksize parameters for mapping the computation to GPU threads and thread blocks. Output CUDA code is generated[1] with parametric extents for the tensors.

Fig. 1 illustrates the overall approach to mapping the computation to the GPU for the tensor contraction example in Eq. 1. The generated GPU kernel incorporates these four steps:

1) Load slice of input tensors from global memory (GMEM) to shared memory (SMEM);
2) Load a subset of input tensor slices from SMEM to registers (REG);
3) Compute contributions to a slice of the output tensor in registers, using an *outer-product scheme* (explained below)
4) Store the finalized elements of the slice of the output tensor from REG to GMEM

The first three steps are repeated until all contributions to a given data slice of the output tensor are completed. Alg. 1 shows the general structure of the GPU kernel code, with some details of index arithmetic being elided (shown in "...") to simplify the pseudo-code.

The definition of each symbol used in the following sections is provided in Table I.

A. Register Tiling

The trends in GPU architectures point to increasingly large register files. With current Nvidia GPUs, the total storage capacity available in registers is much larger than the shared memory in each streaming multiprocessor (SM). Because accumulating values in registers is faster than read-modify-writes into shared memory, each thread holds a set of elements of the output tensor in its registers and uses shared-memory to buffer slices of the input tensors. Specifically, each thread block handles a hyper-rectangular slice of the output tensor, organizing the complete computation as a sequence of tiles in the tensor contraction's iteration space.

Let N_i and T_i denote the extent of a dimension i and the tile size along dimension i, respectively, $(1 \leq T_i \leq N_i)$. For Eq. 1, each thread block computes a portion of C of size $\prod T_s$, where $s \in \{a, b, c, d\}$, held in registers. In order to do so, the thread block requires $T_a \times N_e \times T_b \times N_f$ and $T_d \times N_f \times T_c \times N_e$ elements of A and B, respectively. The number

[1]OpenCL code generation is planned for the future, but is not done at present.

TABLE I
Terminology

Symbol	Description
N_i	extent of dimension i
T_i	tile size along dimension i
TB_x, TB_y	dimensions mapped to thread block X and Y[2]
TB_k	product of extent of internal indices
REG_x, REG_y	X and Y dimensions of the 2D register array
Blk_x	X dimension of grid[3]

Algorithm 1: Pseudo Code of a Tensor Contraction

Data: g_C, g_A and g_B are global memory for A, B and C, respectively.

```
__global__ void kernel_tc(g_C, g_A, g_B)
    __shared__ double s_A[T_a × T_b][T_e × T_f];
    __shared__ double s_B[T_e × T_f][T_c × T_d];
    double r_A[T_b]; // T_b × 1 column vector
    double r_B[T_c]; // 1 × T_c row vector
    double r_C[T_b][T_c]; // T_b × T_c register tile
    // (N_e × N_f)/(T_e × T_f) Steps
    // i is an index used for internal indices
    for i = 0 to ⌈(N_e × N_f)/(T_e × T_f)⌉ do
        // (1) Load Inputs from GMEM to SMEM
        for j = 0 to T_b do
            s_A[—][—] = g_A[—];
        for j = 0 to T_c do
            s_B[i][j] = g_B[—];
        __synchronize();
        for j = 0 to (T_e × T_f) do
            // (2) Load Inputs from SMEM to REG
            for k = 0 to T_b do
                r_A[k] = s_A[k][—];
            for k = 0 to T_c do
                r_B[k] = s_B[—][k];
            // (3) Outer-Product
            for k = 0 to T_b do
                for l = 0 to T_c do
                    r_C[k][l] = r_A[k] × r_B[l];
        __synchronize();
    // (4) Store the Output from REG to GMEM
    for j = 0 to T_b do
        for k = 0 to T_c do
            g_C[—] = r_C[j][k];
```

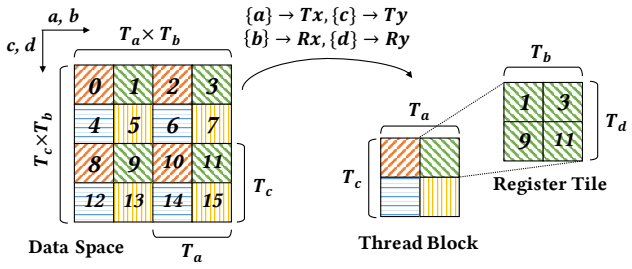

Fig. 2. Illustration of mapping of elements of result tensor C to threads and registers for contraction in Eq. 1

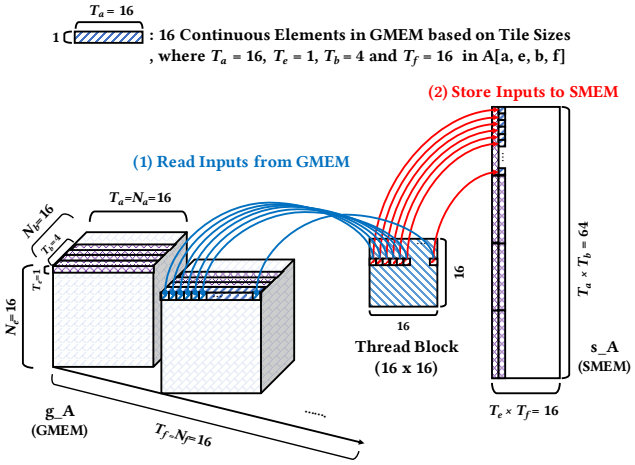

Fig. 3. Illustration of mapping of dimensions to thread block and register tiles

of thread blocks to execute the entire contraction is given by $\prod \lceil N_i/T_i \rceil$.

B. Mapping to Thread Blocks and Threads

The threads in a thread block are organized into a 2D block of size $TB_x \times TB_y$. Threads operate on a logical 2D array of registers, $REG_x \times REG_y$. Then, each thread computes multiple elements, $REG_x \times REG_y$, through *register tiling*. Thus, a thread block deals with $(TB_x \times TB_y) \times (REG_x \times REG_y)$ elements. For Eq. 1, from the data-space perspective, a thread block is in charge of $T_a \times T_b \times T_c \times T_d$ elements of C, which should be mapped onto the 2D space of threads and 2D register tiles. Then, the tile sizes and the mapping of dimensions to thread blocks and register tiles determines the size of the thread block and the size of the register tiles.

Fig. 2 illustrates an example of a mapping choice and tile-sizes, where the mapping is $\{a\} \rightarrow Tx$, $\{c\} \rightarrow Ty$, $\{b\} \rightarrow Rx$, $\{d\} \rightarrow Ry$, and tile sizes are $T_a = T_b = T_c = T_d = 2$. Then, the output data space of a thread block will be 16 elements, where a and c are the fastest varying indices (FVI) along x-axis and y-axis, respectively. According to the mapping and the tile sizes, a thread block is composed of 4 threads ($T_a \times T_c$) and each thread is associated with a register tile of size 2×2 ($T_b \times T_d$).

C. Efficient Data Movement

Coalesced Loading of Input Tensors: For Eq. 1, each thread block requires slices of both input tensors A ($T_a \times N_e \times T_b \times N_f$) and B ($T_d \times N_f \times T_c \times N_e$), as shown in Fig. 1. Because of the limited size of shared memory, instead of loading a slice of size N_e and N_f, a thread block loads $T_a \times T_e \times T_b \times T_f$ and $T_d \times T_f \times T_c \times T_e$ elements of A and B, respectively, at each step. The choice of these tile sizes determines how much shared memory is required by each thread block, and the choice must be made in a manner that balances data reuse and occupancy of SMs.

For example, in Eq. 1, each thread block requires a hyper-rectangle of A with $T_a \times T_e \times T_b \times T_f$ elements, which means that T_a elements are contiguous in global memory because a is the fastest varying index (FVI) in A. This leads to coalesced memory access for A, as shown in Fig. 3. In the figure, we assume that $N_a = N_e = N_b = N_f = 16$, $T_a = 16$, $T_e = 1$, $T_b = 4$, $T_f = 16$ and we have 16×16 threads in a thread block. Thus 16 contiguous threads (1) load 16 (T_a) contiguous elements from global memory (g_A) and then (2) store them to shared memory (s_t2). In contrast, for tensor B, which requires $T_d \times T_f \times T_c \times T_e$ space, T_d elements are contiguous in global memory because d is the FVI. Based on the above assumption, 16 contiguous threads (1) load four groups of four (T_d) contiguous elements from global memory(g_v2) and then (2) store them to shared memory(s_v2). Global memory access for B can be expected to be les efficient than the access to A – the tile sizes along FVI in the input tensors is an important factor for efficient coalesced global memory access.

Transfer to Registers and Tensor Contraction: After a thread block loads slices of the input tensors to shared memory, each thread loads two sub-slices of the data from shared memory to registers. Specifically, each thread loads a column-vector of the left input tensor A and a row-vector of the right input tensor B to perform an outer-product to contribute additive updates to the elements of the output tensor assigned to the thread. For example, in Eq. 1, each thread loads a $T_b \times 1$ column-vector of A and a $1 \times T_c$ row-vector of B to contribute to $T_b \times T_c$ output elements in a register tile, as shown in Fig. 1.

Coalesced Storing of Output Tensor: After contracting input tensors along the full extents of the contraction indices ($N_e \times N_f$), each thread stores the results from registers to global memory. For Eq. 1, each thread block stores $T_a \times T_b \times T_c \times T_d$ elements of C in Fig. 1. Furthermore, as shown in Fig. 2, the mapping $\{a\} \rightarrow TB_x$, where a is the FVI in the output C, indicates that T_a contiguous threads in a thread block handle T_a contiguous elements of C in global memory. Thus, the tile size on the FVI along TB_x determines the number of elements of C that are coalesced in the store from registers to global memory.

TABLE II
PARAMETERS OF THE GENERATED GPU KERNEL

Name	Description
l_TB_x	external indices mapped on TB_x
l_TB_y	external indices mapped on TB_y
l_TB_k	internal indices mapped on TB_k
l_Tiles	tile sizes of indices

IV. PARAMETRIZED CODE GENERATION AND OPTIMIZATION

For a given tensor contraction expression and a selected set of tile sizes and mappings, the code generator creates a GPU kernel with structure as shown in Algorithm 1. Table II enumerates the code generator's kernel parameters. l_TB_x, l_TB_y, and l_TB_k indicate lists of mappings. l_Tiles is a list of index tile sizes. Specifically, according to the inputs—a tensor contraction expression and its representative problem size—the code generator determines the set of kernel parameters in Table II before generating the CUDA kernel and its driver codes. A challenge in generating efficient GPU kernels for tensor contractions is to determine the tile sizes and mappings, which impact performance by determining coalesced memory access and occupancy.

A naïve approach to select the kernel parameters would involve auto-tuning the parameters across the full search space. Typically, such a search begins by running the kernel using a selected random configuration to measure the actual performance. The next configuration to be selected is guided by an algorithm (e.g., genetic algorithms) or an ensemble of algorithms that drive the search direction towards the configuration with the best performance. However, for tensor contractions, this search space can be very large. For example, let us assume that there are five choices of tile sizes (e.g., $\{1, 2, 4, 8, 16\}$) and five dimensions for mapping: TB_x, TB_y, TB_k, REG_x, and REG_y, where TB_k is for an internal index and TB_x, TB_y, REG_x, and REG_y are for external indices. The total number of configurations possible can be computed as $|mapping| \times |tilesize|$, where $|mapping|$ is the number of mapping choices and $|tilesizes|$ is the number of tile-size choices. For Eq. 1, the number of mapping choices is $|mapping| = 4^4 \times 2^1$, because there are four external indices and two internal indices and they can be mapped onto several dimensions. The number of tile-size choices is $|tilesize| = 6^5$ for six indices. Then, Eq. 1's total space will be $(4^4 \times 2) \times (6^5) = 3,981,312$. The latter does not even consider the possibility of merging dimensions (helps to achieve coalescing if the extent of each dimension is very small), splitting each dimension into multiple dimensions (helps ensure that there are enough thread blocks), or thread coarsening; adding these choices will further increase the search space exponentially.

Instead of executing a large number of code versions and selecting the best, our approach relies on a cost model that efficiently rank orders the search space without running the code. The cost model is based on various factors, such as total number of memory transactions. First, we enumerate all possible combinations of kernel parameters that do not violate any hardware constraints (e.g., using more shared memory than available) or performance constraints (e.g., very low parallelism). Such configurations are identified using a set of rules described in Section IV-A. Each configuration's cost is then estimated using a cost model based on DRAM data movement from inputs and output. The cost model is explained in detail in Section IV-B.

A. Enumeration with Pruning

The entire search space consists of different tile sizes and choices. First, we explain the constraints we apply to choose the tile sizes and mapping to reduce the search space. Hardware constraints reduce the search space by eliminating configurations that do not satisfy the hardware limits and the performance constraints reduce the search space by eliminating configurations that are expected to achieve low performance.

1) Hardware Constraints: The tile size choice affects the amount of shared memory and registers required. Larger tile sizes allow better reuse; however, they are constrained by available hardware resources. The shared memory capacity required for the inputs is $(TB_x \times TB_k \times REG_x) + (TB_y \times TB_k \times REG_y)$ and the register capacity for storing a slice of the output tensor is $(REG_x \times REG_y)$. For a configuration to be valid, the shared memory and registers required should be less than hardware capacity.

2) Performance Constraints: To maximize reuse, external indices are preferentially mapped to TB_x and TB_y. However, if the fastest varying index (FVI) of either input tensors corresponds to an internal index, excluding it will lead to uncoalesced accesses that significantly reduce performance. Hence, while choosing indices mapped to TB_x or TB_y, we always include the FVI of the input tensor. Similarly, for output coalescing, we also include the output FVI to the dimensions mapped to shared memory. In addition to coalescing, we ensure that the number of thread blocks is above a threshold. A low number of thread blocks adversely affects load balancing. If the number of thread blocks is too low, then some SMs may be work starved (low occupancy). For example, consider a tensor contraction where the size of tensors is small. In this case, using the maximum possible register tile size will achieve better reuse. However, higher register tile sizes reduce the number of thread blocks, potentially resulting in reduced performance. The shared memory size and number of registers per thread affects achievable occupancy; hence the tile sizes are also constrained to ensure good occupancy.

3) Enumeration Algorithm: In order to maintain good occupancy for double precision, the tile size choices for TB_x, TB_y, and TB_k are limited to $\{4, 8, 16\}$ and the choices for REG_x and REG_y are limited to $\{2, 4, 6, 8\}$. For each combination of TB_x, TB_y, TB_k, REG_x, and REG_y, different mapping and tile-size choices are explored that respect the hardware and performance constraints.

In order to determine possible configurations, given a tensor contraction and a representative problem size, we first explore different choices of external indices in the input tensor that is indexed by the output tensor's FVI, for TB_x mappings and tile sizes of indices mapped onto TB_x. We then explore choices of the other external indices that will be mapped on REG_x. After mapping TB_x and TB_y, if there exist unmapped external indices, they will be mapped on Blk_x. Similarly, for the another input tensor, we explore choices of external indices for Ty and then Ry.

All remaining external indices are mapped to thread blocks Bx. They are technically mapped on TB_x or TB_y with tile-size of 1. Because each thread is responsible for fully processing $REG_x \times REG_y$ elements, all the remaining internal indices are mapped on the serial dimension, TB_k. The code generator ensures that an external index is only mapped to one dimension.

Algorithm 2 shows the detailed description of TB_x mapping and tile-size choices. Because of the performance constraints, we first map the output's FVI on TB_x as TB_x's FVI (Line 11 to Line 21). When we map an index to a dimension in Line 10, we determine its tile-size by using its representative problem size. So we calculate a candidate for the dimension size in Line 11. However, the candidate size might exceed the target TB_x dimension size, TB_size. Then, we make its tile size as big as possible in Line 14. These choices are stored in kernel parameters in Table II. For mapping TB_x, we use two kernel parameters such as l_TB_x and l_Tiles. Indices mapped to TB_x are stored in l_TB_x and the tile sizes of indices mapped to TB_x are stored in l_Tiles.

After mapping an external index to TB_x, the candidate size can be less than the target TB_x size, which means that other external indices can be mapped on TB_x. In order to enumerate all possible cases, we start from the input's FVI to the Slowest Varying Index (SVI) (Line 3) using s_idx. The other external indices are handled in two parts: from s_idx to the SVI and from 0 to $s\_idx - 1$, in Line 24 and Line 44, respectively.

After determination of TB_x, choices for REG_x are made by using unmapped-external indices in the input in Line 47. In enum_X_REG(), we enumerate all possible mapping cases for REG_x with different target REG_x dimension sizes such as {2,4,6,8}, based on a specific mapping case for TB_x, as determined in Line 18 or 37, with a specific target TB_x dimension size determined in Line 2. The information composed of the mappings, TB_x, REG_x and tile sizes for the indices mapped on TB_x and REG_x are stored in $l\_partial\_config\_X$.

Algorithms to determine the mapping choices for REG_x, TB_y, REG_y, and TB_k are similar to Algorithm 2, except for handling the output's FVI. There are three different partial configurations information such as (1) TB_x and REG_x, (2) TB_y and REG_y, and (3) TB_k. All possible configurations can obtained by taking the Cartesian product of them. Before running the cost-model for all possible configurations, they are pruned using our constraints. In the current code generator, for the benchmarks evaluate, around 97% of the configurations were pruned

Algorithm 2: Enumerating Mapping and Tile-Sizes for TB_x

```
   Input: A, C, l_sizes_TB
   Output: l_partial_config
1  def enum_X_TB(l_size_TB, l_size_REG, A, C):
2      for TB_size in l_size_TB :
3          for s_idx in range(0, A.len) :
4              v_TBx = 1
5              v_TBx_prev = 1
6              l_TBx = []
7              l_Tiles = []
8              is_mapped = False
9              // A is assumed to have Output Tensor's FVI
10             l_TBx.append(C[0])
11             v_TBx *= C[0].size
12             if v_TBx >= TB_size :
13                 if v_TBx > TB_size :
14                     blk_size = TB_size / v_TBx_prev
15                     l_Tiles.append([C[0].name, blk_size])
16                 else:
17                     l_Tiles.append([C[0].name, C[0].size])
18                 is_mapped = True
19             else:
20                 l_Tiles.append([C[0].name, C[0].size])
21             v_TBx_prev *= C[0].size
22             if is_mapped == False :
23                 // from s_idx to A.len
24                 for t_idx in range(s_idx, A.len) :
25                     if A[t_idx] is internal :
26                         continue
27                     if A[t_idx] == C[0] :
28                         continue
29                     v_TBx *= A[t_idx].size;
30                     if v_TBx >= TB_size :
31                         l_TBx.append(A[t_idx])
32                         if v_TBx > TB_size :
33                             blk_size = TB_size /
                                v_TBx_prev;
34                             l_Tiles.append([A[t_idx].name,
                                blk_size]])
35                         else:
36                             l_Tiles.append([A[t_idx].name,
                                A[t_idx].size]])
37                         is_mapped = True
38                         break
39                     else:
40                         l_TBx.append(A[t_idx])
41                         l_Tiles.append([A[t_idx].name,
                            A[t_idx].size]])
42                     v_TBx_prev *= A[t_idx].size;
43                 // the remaining indices from 0 to s_idx
44                 for t_idx in range(0, s_idx) :
45                     // Similar from Line 24 to Line 42
46             if is_mapped == True :
47                 l_partial_config_X =
                    enum_X_REG(l_size_REG, A, l_TBx,
                    l_Tiles)
48     return l_partial_config_X;
```

B. Cost Model

The purpose of a cost model is to quickly predict the performance of a configuration with good accuracy. Even though pruning helped to reduce the search space, running each remaining configuration is still expensive. Hence our

cost model is based on estimated amount of data moved from/to the global memory (DRAM). The data movement traffic is calculated using analytical models parametrized by kernel parameters such as tile sizes and mapping choices for indices to thread blocks and register tiles. Along with tile size, the model also considers the mapping choice to determine the amount of coalesced memory access. The cost predicted by our analytical model is well co-related with the actual performance.

The best performing tile sizes and mappings are not only dependent on the contraction but also on the problem size. In order to guide tile size selection, the user is expected to provide a representative problem size as an input. When the code generator receives a set of representative problem sizes, it can generate different code versions targeted at each representative problem size. We note that the code generator does not require the exact problem size at compile time but only a representative size, for the purpose of performance modeling and parameter optimization. Although the kernel is selected at runtime based on the closest representative for the performance, generated kernels can support arbitrary problem sizes.

The cost model includes the number of memory transactions to load both input tensors and to store the output tensor. The cost model assumes that the size of each global memory transaction is 128 bytes (16 double precision elements) and each transaction is aligned to a 128 byte boundary.

According to kernel parameters such as mappings and tile-sizes, the input and output data spaces of a thread block and the number of threads in a thread block. According to the GPU kernel execution strategy in Section III, when the generated CUDA kernels load an input tensor, threads along the x-axis will load elements along an input's FVI in order to enable coalesced memory access. The number of transactions from a row of threads within a thread block is determined as shown in Algorithm 3. In Line 2, helper function cal_Cont() returns the number of contiguous elements in a hyper-rectangle of A in global memory. The hyper-rectangle of A is based on tile sizes of indices in A. In Line 3, helper function cal_Size_TB_x() returns the number of threads along the x-axis. Then, $min(size\_Cont, size\_TB_x)$ will be the maximum number of contiguous elements in the hyper-rectangle of A in global memory. Each row of threads in a thread block reads $size\_TB_x$ elements with $size\_Tx/min(size\_Cont, size\_TB_x) = numTransTx$ transactions, in Line 8. We can load $size\_Tx \times size\_TB_y$ elements at once with $numTransTB_x \times size\_TB_y = numTransTB$ transactions (Line 9). $TB_x \times TB_y \times REG_x$ is equal to a partial slice of a hyper-rectangle of an input tensor, as per the approach described in Section III. At each step, we need $numTransTB \times size\_REG_x = numTransStep$ transactions (Line 10). For all contributions, a thread block requires $numTransStep \times numSteps$ (Line 11). Finally, based on the representative problem size and tile-sizes, we also calculate the number of thread blocks (Line 7), resulting in the total number of the estimated transactions for loading an input tensor (Line 12). Similarly, the number of transactions to store the output is estimated.

Algorithm 3: Calculating DRAM Transaction Costs for Tensor A

Input: $config$: a configuration
Output: $numTransTotal$: the number of estimated transactions to load A

```
1  def calculate_A_cost(A, C, config):
2      size_Cont = cal_Cont(A)
3      size_TB_x = cal_Size_TB_x(config, A)
4      size_REG_x = cal_Size_REG_x(config, A)
5      size_TB_y = cal_Size_TB_k(config, A)
6      numSteps = cal_Steps(config)
7      numTBs = cal_Num_TBs(config, C)
8      numTransTx = size_TB_x / min(size_Cont, size_TB_x)
9      numTransTB = numTransTB_x × size_TB_y
10     numTransStep = numTransTB × size_REG_x
11     numTransFullStep = numTransStep × numSteps
12     numTransTotal = numTransFullStep × numTBs
13     return numTransTotal
```

V. Experimental Results

In this section, we present an experimental evaluation of the code generated by the model-driven code generator (COGENT: COde GENerator for Tensors). The experiments were carried out on two GPUs: an Nvidia Pascal P100 GPU (56 Pascal SMs, 64 cores/MP, 16GB global memory) and an Nvidia V100 GPU (80 Volta SMs, 64 cores/MP, 16GB global memory). All the codes were compiled using CUDA 9.0 and GCC 6.2. All benchmarks were run 3 times and the average is reported.

We used the benchmarks in the TCCG benchmark suite [13] compiled by Springer et. al [14], using the same problem sizes as used by them in their paper. The TCCG benchmark suite is comprised of 48 different tensor contractions, representing a number of specific contractions that arise in real applications. It includes several compute-intensive use-cases from quantum chemistry applications: 18 contractions from the CCSD(T) method (31st to 48th in Fig. 4 and Fig. 5), 19 contractions from the CCSD method (12th to 30th in Fig. 4 and Fig. 5), 3 contractions used to transform a set of two-electron integrals from an atomic orbital basis to a molecular orbital basis (9th to 11th in Fig. 4 and Fig. 5), and a set of 8 contractions involving tensor-matrix multiplication, representing computations in machine learning (1st to 8th in Fig. 4 and Fig. 5).

We compared COGENT with TAL_SH [10] and the code generator used to synthesize GPU kernels for tensor contractions in the production computational chemistry suite NWChem [15]. TAL_SH is based on the TTGT (Transpose-Transpose-GeMM-Transpose) approach. We configured TAL_SH to use CuTT to perform transposition as it achieved the best performance. NWChem's code generator is based on direct tensor contractions on GPUs. The CCSD(T) implementation in the production NWChem distribution is generated using this code generator.

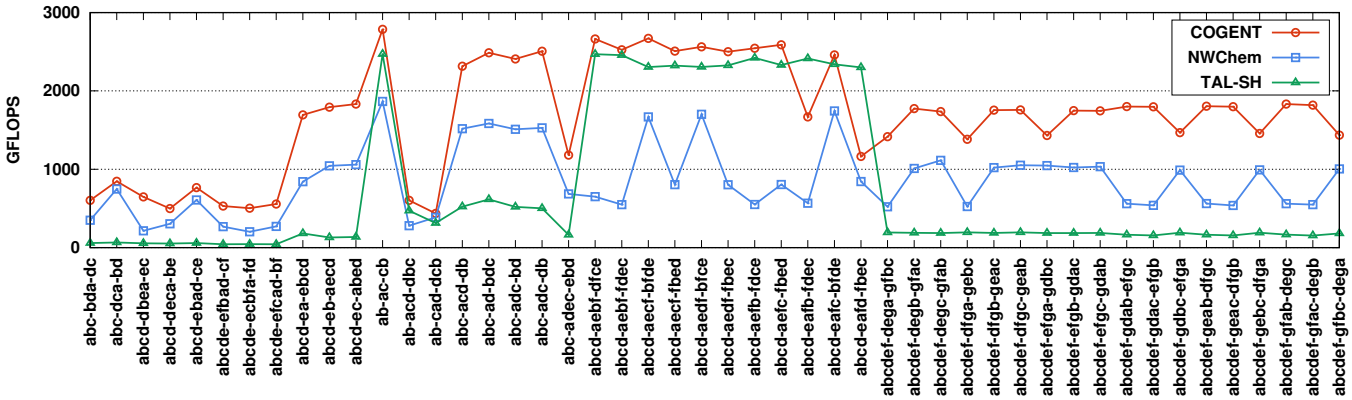

Fig. 4. TCCG Benchmark on P100 (Pascal)

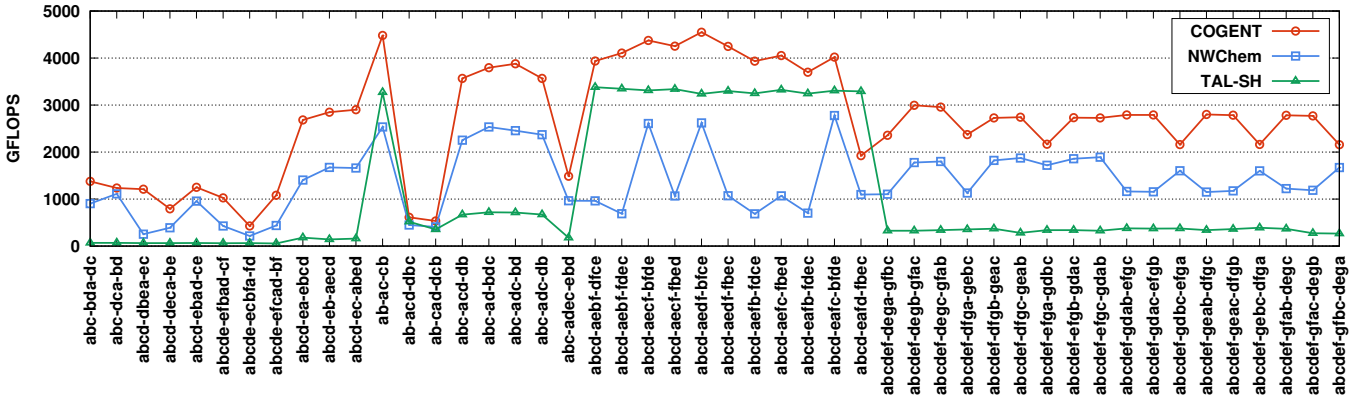

Fig. 5. TCCG Benchmark on V100 (Volta)

Fig. 4 and Fig. 5 compare performance on an Nvidia Pascal P100 GPU and a V100 GPU, respectively. For the 18 CCSD(T) contractions, the time spent to transpose the input tensors and output tensors slows down TAL_SH. Hence, TAL_SH only achieves around 200 GFLOPS and 390 GFLOPS on the P100 and V100, respectively. The NWChem Code Generator achieves 520 to 1050 GFLOPS on the P100 and 1100 GFLOPS to 1900 GFLOPS on the V100. However, due to superior mapping and tile size selection, COGENT attains 1050 GFLOPS to 1300 GFLOPS on the P100 and from 1800 GFLOPS to 2100 GFLOPS on the V100. For the 4D = 4D * 4D contractions (the 12th and 20th to 30th benchmarks), the transposition time is very much lower than compute time; hence TAL_SH achieves very good performance by using the highly tuned matrix multiplication primitives provided by cuBLAS [16]. On the Pascal P100, COGENT generates faster code than TAL_SH for five cases and is competitive in other cases. However, on the Volta V100, COGENT consistently outperforms TAL_SH. For the remaining cases, COGENT is usually faster than the NWChem code generator and TAL_SH.

Fig. 6 and Fig. 7 compare COGENT with Facebook's Tensor Comprehensions (TC) [12] on the P100 and V100, for the SD2 tensor contractions for the CCSD(T) method, for single precision. TC uses a polyhedral optimizer for GPU code generation, in conjunction with an auto-tuner using genetic algorithms. We present results with only a small subset of the TCCG benchmarks since the time taken to tune the TC code for each benchmark ranged between several hours to several days. We also could not compare double precision results since TC could not generate the corresponding code. All TC experiments were run with the population size set to 100 and the generation size set to 20. COGENT's model-driven code consistently and often significantly outperforms the extensively auto-tuned code from TC. Fig. 8 shows the performance trend (in GFLOPS) as a function of the number of auto-tuning iterations with TC, for the SD2_1 (bcdef-gdab-efgc) benchmark. The blue and green lines show the performances without tuning and with tuning, respectively. Without any tuning TC achieves less than 1 GFLOP on both P100 and V100. With tuning, TC achieves between 400 to 1000 GFLOPS on the P100 and between 900 to 1500 GFLOPS on the V100. The total tuning time was $\approx$ 8514 seconds.

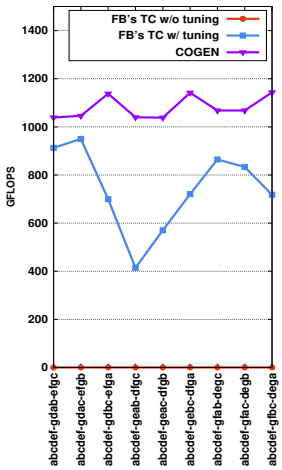

Fig. 6. TCCG Benchmark (40th - 48th) on P100 (Pascal)

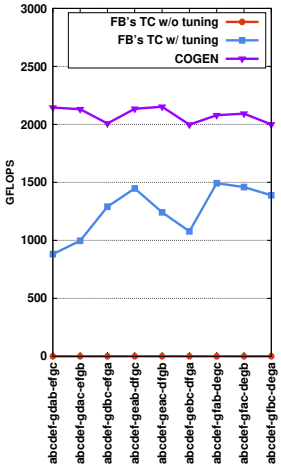

Fig. 7. TCCG Benchmark (40th - 48th) on V100 (Volta)

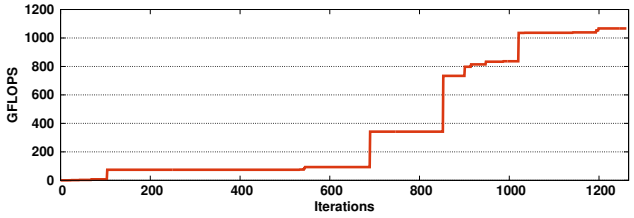

Fig. 8. GFLOPS vs number of code versions for tensor comprehensions on V100 (Volta) for SD2_1 (abcdef-gdab-efgc)

VI. Related Work

Several other efforts have addressed the problem of efficient tensor contractions on multicore CPUs. The state-of-the-art techniques can be broadly grouped into two classes: 1) the TTGT (Transpose-Transpose-GEMM-Transpose) approach based and 2) *direct* contraction based (without explicit transposition).

TTGT: Quantum chemistry codes that implement any members of the coupled cluster family of models require tensor contractions, which dominate their execution time. These codes typically use the TTGT approach, relying on the availability of efficient matrix-multiplication routines on all platforms. Although the number of arithmetic operations for performing the matrix-multiplication is generally much larger than the number of data elements moved for the transpose operations, the time for the latter can be very significant due to the much lower memory bandwidth relative to computational peak, as well as the inefficiency of high-stride data access. Therefore, several efforts have targeted the development of efficient tensor transposition routines for multicore CPUs [17], [18] as well as GPUs [10], [18], [19]. HPTT [17] is an optimized tensor transposition library for multi-core CPUs. TTC [18] is a compiler that can generate high-performance transpose routines for both GPUS as well as multicore processors. TAL_SH [10] is a software framework for efficient tensor contraction on GPUs that uses the TTGT approach. It links with the cuTT library [19], [20] for efficient GPU tensor transposition. In our experimental evaluation, we perform comparisons with TAL_SH on GPUs. We also benchmark achievable performance for TTGT using HPTT on a multicore CPU.

Direct Contraction: While the TTGT approach has the benefit of leveraging efficient vendor matrix multiplication libraries, the extra transposition imposes both space and time overheads. So several efforts have developed implementations for tensor contractions that avoid the use of transpose. Two recent efforts have independently developed such an approach for multicore CPUs. TBLIS [21] uses the BLIS (BLAS-like Instantiation Software [22]) framework to implement arbitrary tensor contractions by essentially fusing transposition of slices of tensors on-the-fly and invoking BLIS kernels for efficient matrix-matrix multiplication. GETT [14] implements the operations for an arbitrary tensor contraction as a loop over slices that are computed using a highly tuned "macro kernel" whose operands reside in a certain level of the cache hierarchy. GETT has been distributed as part of the TCCG (Tensor Contraction Code Generator) [13] framework, which includes several alternatives for performing tensor contractions, including TTGT, GETT, a loop-over-GEMM approach (LoG) and a direct nested-loop implementation for tensor contractions. Ma et al. [15], [23] developed a code generator for implementing direct tensor contractions for GPUs, with a special focus on the most expensive triples computation for the CCSD(T) method in the NWChem computational chemistry suite. In our experimental evaluation, we present performance for the publicly available options: TTCG and GETT. Shi et al. [24] use a new strided batched BLAS functionality in Nvidia's cuBLAS as a means of implementing direct tensor contractions for a set of contractions of significance for machine learning. Tensor Comprehensions (TC) [12] is a high level domain-specific language to express tensor operations. The high level expression is then converted to a polyhedral representation, and is subject to a set of polyhedral transformations. The framework is capable of producing both CPU and GPU code. It offers a JIT based autotuner that uses a genetic algorithm to prune the search space. The framework allows specification of a representative problem size, which is used to evaluate the quality of the generated code during auto-tuning.

Parameter Search Optimization: Tensors are core data structures used in machine learning (ML) applications. Until recently, much of the work in optimizing tensor computations for machine learning was done via manual implementation of libraries such as Nvidia's cuDNN. A recent effort [25] has sought to automate the optimization of tensor computations in ML by developing a learning framework that learns to choose among the options in a space of transformed configurations for the code. As pointed out by the authors, they focus on effective search among the possible configurations in a pre-defined space, and the choice of that space is beyond the scope of the learning strategy for selection. We believe that model-driven frameworks like the one proposed here complement learning-

based optimizers by defining an effective configuration space of high-performance configurations from which the best ones are to be selected. While we have only model-driven selection of a set of configurations and auto-tuned across a selected set of configurations, our model-driven approach could be enhanced by using a learning-based approach to perform the selection among the top set of candidate configurations based on our analytical modeling.

VII. Conclusions

This paper has presented a CUDA code generator for arbitrary tensor contractions. It uses a code generation schema based on domain-specific properties about data reuse in tensor contractions, along with a model-driven pruning strategy for rapid determination of parameters for mapping of computation to threads and staging of data through the GPU memory hierarchy. Experimental results demonstrate significant performance gains over state-of-the-art tensor contraction code generators and libraries over a range of benchmarks.

Appendix
Artifact Appendix

A. Abstract

The artifact contains all the programs required to reproduce the experimental results in the CGO 2019 paper "A Code Generator for High-Performance Tensor Contractions on GPUs". The artifact is publicly available for download from https://doi.org/10.6084/m9.figshare.7403732. The latest version is available on the git repository: http://gitlab.hpcrl.cse.ohio-state.edu/jinsung/COGENT.

The downloaded package comes with

- The source code for COGENT
- The scripts to run benchmarks on all the compared frameworks
- Expected output in the form of text files
- A very detailed README file which contains detailed instructions to build and troubleshoot installation of all frameworks.

B. Artifact Check-List (Meta-Information)

- **Program:** (1) COGENT (COde GENerator for Tensor contractions), (2) NVIDIA CUDA kernels generated by NWChem's code generator, (3) TAL-SH benchmark, and (4) Tensor Comprehensions benchmark.
- **Compilation:** Detailed instructions to compile and scripts to run each framework is provided below and in the README.md.
- **Transformations:** The code generator accepts a tensor contraction expression and generates corresponding cuda kernels.
- **Binary:** Makefile is included in the package to generate the executable.
- **Data set:** 48 Tensor Contractions in TCCG Benchmark.
- **Software:** Linux version 3.10.0 (tested), CPU code: g++ with C++11 support (GCC 6.3.0 tested) and python (3.5 tested); GPU code: NVCC (9.0 tested).
- **Hardware:** Linux platform such as Ubuntu, and a GPU device with compute capability $\geq$ 6.0. The benchmarks reported on the paper were ran on an Nvidia Tesla P100 machine and an Nvidia Tesla V100 machine.
- **Output:** Text files containing the execution time and GFLOPS used for the Figure 4 - 8.
- **Publicly available?:** Yes.

C. Description

1) How Delivered: Our artifact is available on both https://doi.org/10.6084/m9.figshare.7403732 and http://gitlab.hpcrl.cse.ohio-state.edu/jinsung/COGENT. All the files in the repository are licensed to The Ohio State University.

2) Hardware Dependencies: The generated code can be executed on any Nvidia device with compute capability $\geq$ 6.0. For reproducing the results reported in the paper, we suggest using Tesla P100 and Tesla V100 devices.

3) Software Dependencies: COGENT requires Python 3.5. CUDA kernels generated by COGENT and NWChem require GCC version ($\geq$ 6.3.0) with C++11 support and NVCC version ($\geq$ 9.0). TAL-SH requires cuTT (instructions to build cuTT are provided) and BLAS library. TC require conda.

4) Data Sets: The benchmark is based on 48 tensor contractions in TCCG benchmark. Since each framework accepts different formats, framework specific representation of benchmarks are provided in each each framework's directory.

For COGENT, expressions of TCCG benchmark are provided in ./cogent/input_strings/tccg directory. Source codes generated by NWChem's code generator are provided in NWChem directory. For TAL-SH, we provide ./tal-sh/test.cpp which includes all the TCCG benchmark. Finally, for TC, we provide python codes to evaluate the benchmarks.

D. Installation

After cloning the repository— http://gitlab.hpcrl.cse.ohio-state.edu/jinsung/COGENT, TAL-SH and TC should be built before evaluation (see *README.md* for detailed instructions). COGENT and NWChem can be installed using the Makefile provided.

1) TAL-SH: As TAL-SH is dependent on cuTT, install cuTT from the git-repository: https://github.com/ap-hynninen/cutt before building TAL-SH (https://github.com/DmitryLyakh/TAL_SH)

2) Tensor Comprehensions (TC): TC requires conda, build anaconda3 by following the instructions on https://conda.io/docs/index.html. TC can then be installed as follows:

```
$ conda install -y -c pytorch -c tensorcomp tensor_comprehensions
```

E. Experiment Workflow

Scripts are provided to run different frameworks.

1) COGENT:

To run the benchmark for Fig. 4 and 5:

```
$ cd cogent
$ bash ./bench_tccg.sh
```

To run the benchmark for Fig. 6 and 7:

```
$ cd cogent
$ bash ./cogent/bench_fb.sh
```

2) NWChem:

To run the benchmark for Fig. 4 and 5:

```
$ cd nwchem-tccg
$ bash ./bench_tccg.sh
```

3) TAL-SH:
To run the benchmark for Fig. 4 and 5:
(In the directory where you have built TAL-SH,)
$ bash ./test_talsh.x

4) Tensor Comprehensions:
For TC with tuning, to run the benchmark for Fig. 6, 7 and 8:
$ cd fb-tc/fb-w-tuning/
$ bash ./bench_fb_w_tuning.sh

For TC without tuning, to run the benchmark for Fig. 6 and 7:
$ cd fb-tc/fb-wo-tuning
$ bash ./bench_fb_wo_tunning.sh

F. Evaluation and Expected Result

We expect the performance results to be close to those reported in the paper (Fig. 4 - 8). The results of the benchmark will be printed out in text files.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback and suggestions that helped improve the paper. We are grateful to the Ohio Supercomputer Center for use of their hardware resources. This work was supported in part by the U.S. National Science Foundation (NSF) through awards 1440749, 1513120 and 1816793, by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under awards 71648 and DE-SC0014135, and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830.

REFERENCES

[1] C.Kavaklioglu and A. T. Cemgil, "Optimal contraction order of multiple tensors," in *2013 21st Signal Processing and Communications Applications Conference (SIU)*, April 2013, pp. 1–4.

[2] N. A. Rink, I. Huismann, A. Susungi, J. Castrillon, J. Stiller, J. Fröhlich, and C. Tadonki, "Cfdlang: High-level code generation for high-order methods in fluid dynamics," in *Proceedings of the Real World Domain Specific Languages Workshop 2018*, ser. RWDSL2018. New York, NY, USA: ACM, 2018, pp. 5:1–5:10. [Online]. Available: http://doi.acm.org/10.1145/3183895.3183900

[3] R. Poya, A. J. Gil, and R. Ortigosa, "A high performance data parallel tensor contraction framework: Application to coupled electro-mechanics," *Computer Physics Communications*, vol. 216, pp. 35 – 52, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465517300681

[4] A. A. Auer, G. Baumgartner, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov, "Automatic code generation for many-body electronic structure methods: the tensor contraction engine," *Molecular Physics*, vol. 104, no. 2, pp. 211–228, 2006. [Online]. Available: https://doi.org/10.1080/00268970500275780

[5] F. Huang, U. N. Niranjan, M. U. Hakeem, P. Verma, and A. Anandkumar, "Fast detection of overlapping communities via online tensor methods on gpus," *CoRR*, vol. abs/1309.0787, 2013. [Online]. Available: http://arxiv.org/abs/1309.0787

[6] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[7] A. Anandkumar, R. Ge, D. J. Hsu, S. M. Kakade, and M. Telgarsky, "Tensor decompositions for learning latent variable models," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 2773–2832, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2697055

[8] T. Crawford and H. Schaefer III, "An Introduction to Coupled Cluster Theory for Computational Chemists," in *Reviews in Computational Chemistry*. John Wiley & Sons, Inc., 2000, vol. 14, pp. 33–136.

[9] P. Springer and P. Bientinesi. Tensor contraction benchmark v0.1. [Online]. Available: https://github.com/HPAC/tccg/

[10] D. I. Lyakh, "Talsh," 2014. [Online]. Available: https://github.com/DmitryLyakh/TAL_SH

[11] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 54:1–54:23, Jan. 2013.

[12] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *arXiv preprint arXiv:1802.04730*, 2018.

[13] P. Springer and P. Bientinesi, "Tccg," 2018. [Online]. Available: https://github.com/HPAC/tccg

[14] P. Springer and P. Bientinesi, "Design of a high-performance gemm-like tensor–tensor multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 44, no. 3, p. 28, 2018.

[15] W. Ma, S. Krishnamoorthy, O. Villa, K. Kowalski, and G. Agrawal, "Optimizing tensor contraction expressions for hybrid cpu-gpu execution," *Cluster computing*, vol. 16, no. 1, pp. 131–155, 2013.

[16] C. Nvidia, "Cublas library," *NVIDIA Corporation, Santa Clara, California*, vol. 15, no. 27, p. 31, 2008.

[17] P. Springer, T. Su, and P. Bientinesi, "Hptt: a high-performance tensor transposition c++ library," in *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ACM, 2017, pp. 56–62.

[18] P. Springer, A. Sankaran, and P. Bientinesi, "Ttc: a tensor transposition compiler for multiple architectures," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ACM, 2016, pp. 41–46.

[19] A.-P. Hynninen and D. I. Lyakh, "cutt: A high-performance tensor transpose library for cuda compatible gpus," *arXiv preprint arXiv:1705.01598*, 2017.

[20] D. I. Lyakh, "An efficient tensor transpose algorithm for multicore cpu, intel xeon phi, and nvidia tesla gpu," *Computer Physics Communications*, vol. 189, pp. 84–91, 2015.

[21] D. A. Matthews, "High-performance tensor contraction without transposition," *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. C1–C24, 2018.

[22] F. G. Van Zee and R. A. Van De Geijn, "Blis: A framework for rapidly instantiating blas functionality," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 3, p. 14, 2015.

[23] W. Ma, S. Krishnamoorthy, O. Villay, and K. Kowalski, "Acceleration of streamed tensor contraction expressions on gpgpu-based clusters," in *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*. IEEE, 2010, pp. 207–216.

[24] Y. Shi, U. N. Niranjan, A. Anandkumar, and C. Cecka, "Tensor contractions with extended blas kernels on cpu and gpu," *arXiv preprint arXiv:1606.05696*, 2016.

[25] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," *arXiv preprint arXiv:1805.08166*, 2018.