Optimizing Tensor Contractions in CCSD(T) for Efficient Execution on GPUs

Jinsung Kim[1], Aravind Sukumaran-Rajam[1], Changwan Hong[1], Ajay Panyala[2],
Rohit Kumar Srivastava[1], Sriram Krishnamoorthy[2], P. Sadayappan[1]
[1] The Ohio State University,
{kim.4232, sukumaranrajam.1, hong.589, srivastava.141, sadayappan.1}@osu.edu
[2]Pacific Northwest National Laboratory,
{ajay.panyala, sriram}@pnnl.gov

ABSTRACT

Tensor contractions are higher dimensional analogs of matrix multiplications, used in many computational contexts such as high order models in quantum chemistry, deep learning, finite element methods etc. In contrast to the wide availability of high-performance libraries for matrix multiplication on GPUs, the same is not true for tensor contractions. In this paper, we address the optimization of a set of symmetrized tensor contractions that form the computational bottleneck in the CCSD(T) coupled-cluster method in computational chemistry suites like NWChem. Some of the challenges in optimizing tensor contractions that arise in practice from the variety of dimensionalities and shapes for tensors include effective mapping of the high-dimensional iteration space to threads, choice of data buffering in shared-memory and registers, and tile sizes for multi-level tiling. Furthermore, in the case of symmetrized tensor contractions in CCSD(T), it is also a challenge to fuse contractions to reduce data movement cost by exploiting reuse of intermediate tensors. In this paper, we develop an efficient GPU implementation of the tensor contractions in CCSD(T) using shared-memory buffering, register tiling, loop fusion and register transpose. Experimental results demonstrate significant improvement over the current state-of-the-art.

CCS CONCEPTS

• **Computing methodologies → Shared memory algorithms**; • **Computer systems organization → Single instruction, multiple data**; • **Applied computing** → *Chemistry*;

KEYWORDS

Tensor Contractions, coupled Cluster Methods, CCSD(T), Loop Fusion, GPU Computing

ACM Reference Format:
Jinsung Kim[1], Aravind Sukumaran-Rajam[1], Changwan Hong[1], Ajay Panyala[2], Rohit Kumar Srivastava[1], Sriram Krishnamoorthy[2], P. Sadayappan[1] [1] The Ohio State University, {kim.4232, sukumaranrajam.1, hong.589, srivastava.141, sadayappan.1}@osu.edu [2]Pacific Northwest National Laboratory, {ajay.panyala, sriram}@pnnl.gov . 2018. Optimizing Tensor Contractions in CCSD(T) for Efficient Execution on GPUs. In *ICS '18: 2018 International Conference on Supercomputing, June 12–15, 2018, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3205289.3205296

ICS '18, June 12–15, 2018, Beijing, China

ACM ISBN 978-1-4503-5783-8/18/06…$15.00
https://doi.org/10.1145/3205289.3205296

1 INTRODUCTION

Tensor contractions are higher dimensional analogs of matrix multiplication, used in many computational contexts such as high order models in quantum chemistry, deep learning, finite element methods, etc. While similar to matrix multiplication, their unique characteristics have motivated specialized optimization strategies.

In this paper, we present an in-depth description of the GPU optimization for the compute-intensive tensor contractions that dominate the execution time for the perturbative triples correction (T) in Coupled Cluster (CC) methods employed by electronic structure calculations [1]. CCSD(T) is a widely used application in computational chemistry suites like NWChem [2] since it effectively balances the trade-off between high computational costs and desired chemical accuracy. The dominant tensor contraction kernels that constitute (T) produce a six-dimensional tensor, a multidimensional matrix, from four-dimensional input tensors using $O(N^7)$ operations, where N is the tensor dimension. A significant fraction of supercomputer time in performing accurate electronic structure calculations is spent in these kernels [3–5].

The high computational complexity holds the potential to maximize computational efficiency and achieve high sustained FLOPS. However, several features of the (T) method make this a nontrivial challenge. First, the large six-dimensional output tensor restricts the tensor sizes that can be computed on memory-limited systems (e.g., GPUs). We present a strategy to efficiently execute these operations on NVIDIA GPUs.

Second, tensor contractions resemble highly rectangular matrix-matrix multiplication operations, which do not perform as well as multiplication of square matrices. For example, performing DGEMM (using GPUs) on Pascal GPUs achieves no more than 1.9TFLOPS for the problem sizes that can be executed on a GPU. To improve upon this, we develop a novel kernel fusion strategy to combine the operations across a sequence of contractions required in the (T) method due to permutational symmetry of the tensors (explained later). Full fusion across related contractions is achieved via a novel strategy to transpose data in registers by using shared memory as temporary buffers.

We evaluate the new algorithms on NVIDIA Pascal P100 and Volta V100 GPUs. We compare our new implementation against various current state-of-the-art libraries/code-generators/compilers for

GPUs and CPUs. The new implementation significantly improves upon any other current option, including the current production code in NWChem for all O(N^7) kernels in the (T) method. Our fully fused algorithm achieves up to 2.8TFLOPS on the Pascal GPU and 4.9TFLOPS on the Volta GPU, achieving up to 8× speedup as compared to the current GPU code in NWChem. In addition to the significant performance improvements enabled for an important method in the widely used NWChem application, we believe that the design and optimization strategies we describe in this paper can serve as a basis for the development of automated compiler optimizations that can deliver much higher performance than currently achievable by state-of-the-art general-purpose and domain-specific compilers.

2 BACKGROUND

Consider the following tensor contraction:

```
L[a,i,j] += R1[a,k,j] * R2[k,i]
```

Here, the three-dimensional output tensor L is produced by contracting three-dimensional tensor R1 and two-dimensional tensor R2. Tensor contractions are generalizations of matrix multiplication and also have the property that each index variable is present in exactly two out of the three tensors. The indices of the output tensor are referred to as external indices and each of them appears also in one of the two input tensors. The remaining indices occur in both the input tensors and are referred to as the summation indices. Each summation index appears exactly once in both input tensors of a tensor contraction expression. For the tensor contraction shown above, k is a summation index and indexes both R1 and R2, while {a,i,j} are external indices and index the output tensor L, with {a,j} also indexing R1 and k indexing R2.

Coupled cluster methods implement highly accurate ab initio quantum chemistry models [1, 6]. Due to permutation symmetry among subsets of dimensions of a tensor, only distinct elements need to be stored. Multi-dimensional tensors with symmetry are stored as a collection of fully dense "bricks" or data-tiles, where only distinct bricks are explicitly stored. The contraction of two tensors is implemented as a collection of contractions involving the set of bricks representing the tensor [7–10]. The tile sizes are chosen based on the available memory and to ensure efficient communication, maximize computation efficiency throughout the calculation, and enable dynamic load balancing.

The CCSD(T) (Coupled Cluster Singles and Doubles with perturbative triples correction) is considered the "gold standard" for accurate electronic structure calculations [3]. In this paper, we focus on optimizing the perturbative triples (henceforth referred to as just triples correction) in CCSD(T) methods for execution on NVIDIA GPUs. The triples correction consumes significant fractions of supercomputing time and has motivated several optimization efforts. Each tile of the output six-dimensional tensor can be computed in parallel, allowing the perturbative triples correction to scale to the largest supercomputers [3–5].

A consequence of the permutational symmetry of tensors is that many tensor contractions have to be symmetrized, leading to sets of contractions with permuted indexing patterns being accumulated to form result tensors with the stipulated symmetry. For example, a symmetrized 2D tensor A[i<j] produced from B[i] and C[j] would be expressed as:

```
A[i<j] = A[i,j] - A[j,i] = B[i].C[j] - B[j].C[i]
```

Table 1 shows the 18 tensor contractions that constitute the dominant computation for the CCSD(T) method. Each set of 9 contractions arises because of the symmetrization of the result tensor with respect to two sets of indices {i,j,k} and {a,b,c}. Computation of the distinct tiles for the output triples tensor t3[k>j>i,c>b>a] requires the nine-way symmetrization operations to produce the output with the desired anti-symmetry. Each of the contractions requires a 7D nested loop (6 external indices and one summation index) and therefore has a computational complexity of O(N^7).

Table 1: Expressions of the eighteen sd1 and sd2 functions

sd1_1	t3[k,j,i,c,b,a] −= t2[l,a,b,i] * v2[k,j,c,l]
sd1_2	t3[k,j,i,c,b,a] += t2[l,a,b,j] * v2[k,i,c,l]
sd1_3	t3[k,j,i,c,b,a] −= t2[l,a,b,k] * v2[j,i,c,l]
sd1_4	t3[k,j,i,c,b,a] −= t2[l,b,c,i] * v2[k,j,a,l]
sd1_5	t3[k,j,i,c,b,a] += t2[l,b,c,j] * v2[k,i,a,l]
sd1_6	t3[k,j,i,c,b,a] −= t2[l,b,c,k] * v2[j,i,a,l]
sd1_7	t3[k,j,i,c,b,a] += t2[l,a,c,i] * v2[k,j,b,l]
sd1_8	t3[k,j,i,c,b,a] −= t2[l,a,c,j] * v2[k,i,b,l]
sd1_9	t3[k,j,i,c,b,a] += t2[l,a,c,k] * v2[j,i,b,l]
sd2_1	t3[k,j,i,c,b,a] −= t2[d,a,i,j] * v2[d,k,c,b]
sd2_2	t3[k,j,i,c,b,a] −= t2[d,a,j,k] * v2[d,i,c,b]
sd2_3	t3[k,j,i,c,b,a] += t2[d,a,i,k] * v2[d,j,c,b]
sd2_4	t3[k,j,i,c,b,a] += t2[d,b,i,j] * v2[d,k,c,a]
sd2_5	t3[k,j,i,c,b,a] += t2[d,b,j,k] * v2[d,i,c,a]
sd2_6	t3[k,j,i,c,b,a] −= t2[d,b,i,k] * v2[d,j,c,a]
sd2_7	t3[k,j,i,c,b,a] −= t2[d,c,i,j] * v2[d,k,b,a]
sd2_8	t3[k,j,i,c,b,a] −= t2[d,c,j,k] * v2[d,i,b,a]
sd2_9	t3[k,j,i,c,b,a] += t2[d,c,i,k] * v2[d,j,b,a]

The construction of the six-dimensional output tensor from four-dimensional input tensors provides significant opportunities for optimization. A key challenge to efficient execution of these methods is to limit the data movement for the six-dimensional tensor while efficiently executing the highly rectangular tensor contractions. Ma et al. demonstrated pipelined execution of each kernel on the NVIDIA Tesla GPUs [11]. They further optimized it for the Fermi GPUs by retaining each tile of the six-dimensional tensor on the GPU until all its updates are performed [12]. Despite these optimizations, the triples correction has only been shown to achieve a small fraction of the peak floating point performance available on GPUs. In this paper, we develop an efficient fused, tiled GPU implementation that makes effective use of registers and shared memory to minimize data movement to/from global-memory.

3 OVERVIEW OF MAPPING STRATEGY

We first use a simpler example of matrix-matrix multiplication to explain the approach to mapping of data and computations. Figure 1 presents the steps involved in executing a matrix multiplication C[i,j] = A[i,k] * B[k,j]. A thread block computes a $T_i \times T_j$ slice of

the output C. The thread block requires $N_k \times T_i$ and $N_k \times T_j$ slices of the inputs A and B on global memory, respectively.

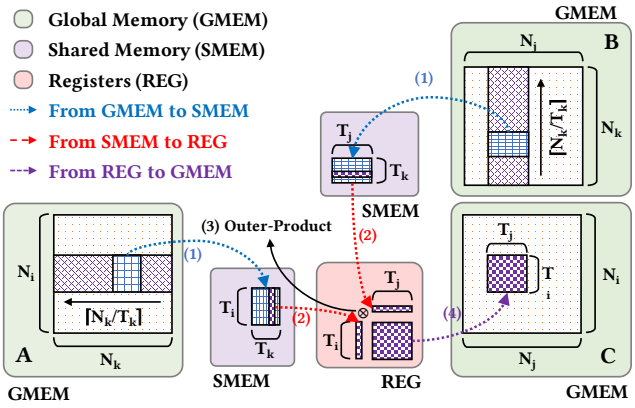


Figure 1: Overview of the computation of a matrix multiplication in a thread block

Each thread block loads $T_i \times T_k$ and $T_k \times T_j$ slices of A and B, respectively, because of the limited size of shared memory, resulting in $\lceil N_k/T_k \rceil$ steps. Then the threads load a row-vector and a column-vector of A and B of size $T_i \times 1$ and $1 \times T_j$, respectively, from shared memory. Then, the thread block computes an outer-product contribution to the output slice — $T_i \times T_j$ of C. Finally, after $\lceil N_k/T_k \rceil$ steps, the thread block stores the elements of the output tensor slice from registers to global memory.

In this and the following sections, we will use the following kernel, the first sd2 kernel listed in the Table 1, for illustration:

$$\text{t3[k, j, i, c, b, a] -= t2[d, a, i, j] * v2[d, k, c, b]} \quad (1)$$

Each thread block deals with a slice of the output t3. The output tensor's dimensions are partitioned by each index's tile for thread blocks. Each thread block handles a hyper-rectangle data tile. Let N_i and T_i be the size of an index i ($0 \le i \le N_i - 1$) and the tile size of an index i ($1 \le T_i \le N_i$), respectively. For Eq. (1), the size of t3 is $\prod N_s$, where $s \in$ {a,b,c,i,j,k}. The size of a portion of t3 computed by a thread block is given by $\prod T_s$, where $s \in$ {a,b,c,i,j,k}.

Let B_i be the number of distinct T_i along an index i, i.e., $B_i = \lceil N_i/T_i \rceil$. Executing the contraction in Eq. (1) requires $\prod B_s$ thread blocks, where $s \in$ {a,b,c,i,j,k}. Finally, let I be a tile-index of an index i, where $0 \le I < B_i$. Then, we can represent a hyperrectangle in t3[k,j,i,b,c,a] which each thread block handles as T3[K,J,I,B,C,A], where $J \times T_j \le j < (J+1) \times T_j$.

For example, consider $N_i = N_j = N_k = N_a = N_b = N_c = 16$ and $T_i = T_j = T_k = T_a = T_b = T_c = 4$. Then, each thread block produces $T_i \times T_j \times T_k \times T_a \times T_b \times T_c = 4096$ elements of t3 and $\lceil N_i/T_i \rceil \times \lceil N_j/T_j \rceil \times \lceil N_k/T_k \rceil \times \lceil N_a/T_a \rceil \times \lceil N_b/T_b \rceil \times \lceil N_c/T_c \rceil = 4096$ thread blocks are required. Furthermore, consider a thread block handling T3[0,1,2,0,3,0. Then, this thread block will produce t3[0:3,4:7,8:11,0:3,12:15,0:3].

3.1 Two-level Tiling

We organize the thread blocks into 2D arrays $TB_X \times TB_Y$ of threads. Threads operate on 2D arrays of registers (REG_X, REG_Y). Each thread computes multiple elements—$REG_X \times REG_Y$—through *register tiling*. Thus, a thread block deals with $(TB_X \times TB_Y) \times (REG_X \times REG_T)$ elements.

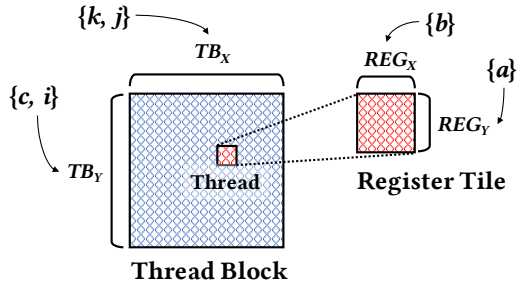


Figure 2: Illustration: mapping dimensions to thread block and register tiles for sd2_1

Because a thread block is in charge of $T_i \times T_j \times T_k \times T_a \times T_b \times T_c$ elements of t3, they should be mapped on 2D arrays of threads and 2D register tile in order to assign specific elements to threads.

We map four among the six external indices to thread blocks and the remaining two external indices are mapped to register tile in order for a thread block to handle $T_i \times T_j \times T_k \times T_a \times T_b \times T_c$. Fig. 2 illustrates a mapping for sd2_1, where $\{k, j\} \rightarrow TB_X$, $\{c, i\} \rightarrow TB_Y$, $\{b\} \rightarrow REG_X$ and $\{a\} \rightarrow REG_Y$, where i and k are the fastest varying index along TB_X and TB_Y, respectively.

The tile sizes and the mapping of dimensions to thread blocks and register tiles determines the size of the thread block. In Fig. 2 the thread block size is given by $(T_i \times T_j) \times (T_k \times T_c)$.

4 OPTIMIZED EXECUTION OF TENSOR CONTRACTIONS

Algorithm 1 shows pseudo-code for our optimized execution of a portion of a single tensor contraction operation mapped to a GPU thread block for the sd2_1 kernel. Some of the index arithmetic is elided (shown in "—") to simplify the pseudo-code.

Figure 3 illustrates the data movement through the different levels of storage for tiled execution for the tensor contraction example of Eq.1. There are 4 steps:

(1) Load slice of input tensors from global memory (GMEM) to shared memory (SMEM)
(2) Load a subset of the data from SMEM to registers (REG)
(3) Compute a contribution to the elements of the output tensor, in the form of an outer-product
(4) Store the elements of the output tensor from REG to GMEM

The first three steps are repeated until all contributions to the data slice of the output tensor for the thread block are completed.

Figure 3 illustrates the steps involved in executing a tensor contraction expression using our algorithm for the kernel in Eq. (1). We begin with a mapping $\{k, j\} \rightarrow TB_X$, $\{c, i\} \rightarrow TB_Y$, $\{b\} \rightarrow REG_X$ and $\{a\} \rightarrow REG_Y$. Then, we have $(T_k \times T_j) \times (T_c \times T_i)$ threads in a thread block and a thread computes $(T_b \times T_a)$ elements as *register tiling*, as mentioned in Sec. 3.1.

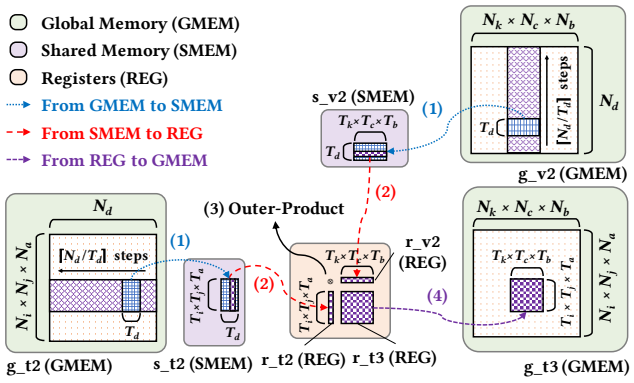


Figure 3: Overview of the computation of a tensor contraction for sd2_1 in a thread block

We assume that the input data is in global memory, say, using CUDA streams, before the kernel begins execution on the GPU. In this section, we present the details of each step.

4.1 Coalesced Loading of Input Tensors

For the sd2_1 function, each thread block requires $N_d \times T_a \times T_i \times T_j$ and $N_d \times T_k \times T_c \times T_b$ of t2 and v2, respectively. Instead of N_d, a thread block loads $T_d \times T_a \times T_i \times T_j$ and $T_d \times T_k \times T_c \times T_b$ of t2 and v2, respectively, at each step, because of the limited size of shared memory (line 9 to 16 in Alg 1).

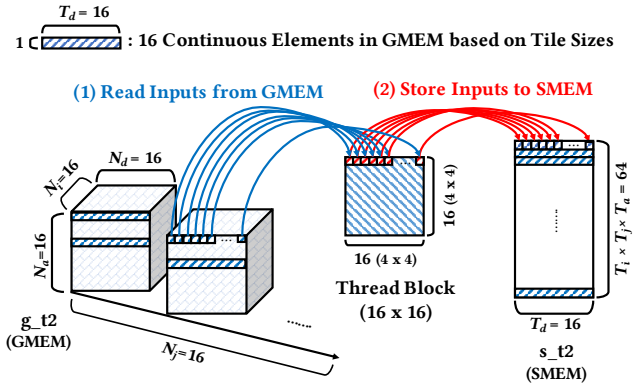


Figure 4: Coalesced loading of input tensor t2 from global memory to shared memory for sd2_1

For example, in the case of t2, in a hyperrectangle of $T_d \times T_a \times T_i \times T_j$, T_d elements are continuous on global memory because d is the fastest varying index (FVI) in t2. This results coalesced memory access for t2 as shown in Fig. 4. In the figure, we assume that $N_k = N_j = N_i = N_c = N_b = N_a = N_d = 16$, $T_k = T_j = T_i = T_c = T_b = T_a = 4$ and $T_d = 16$, resulting in a thread block size of 16 (4×4) along two dimensions as well as a register tile size of 4 along two dimensions. Thus, 16 continuous threads (1) load $16(T_d)$ continuous elements on global memory(g_t2) and then (2) store them to shared memory(s_t2).

On the other hand, in the case of v2 in sd1_1, T_k elements are continuous on global memory because k is the FVI. Based on the above assumption, 16 continuous threads (1) load four different $4(T_k)$ continuous elements on global memory(g_v2) and then (2) store them to shared memory(s_v2). Therefore, the tile size of the FVI in input tensors is an important factor when it comes to coalesced memory access.

Transfer to Registers and Tensor Contraction. Each thread loads a portion of the data in shared memory to registers. In particular, each thread loads a row-vector of the first tensor with a column-vector of the second tensor to perform an outer-product contribution to the elements of the output tensor assigned to this thread via register tiling from line 19 to 32 in Alg 1. The outer-product structure is chosen to maximum arithmetic intensity.

Coalesced Storing the Output Tensor to Global Memory. After contracting input tensors along N_l and N_d for sd_1 and sd_2, respectively, each thread stores the results from registers to global memory from line 36 to 40 in Alg 1. For the sd2_1 function, each thread block stores $T_k \times T_j \times T_i \times T_c \times T_b \times T_a$ elements of of t3.

As shown in Fig. 2, the mapping $\{k, j\} \rightarrow TB_X$, where k is the FVI, indicates that T_k continuous threads in a thread block handle T_k continuous elements of t3 on global memory. Thus, a tile-size mapped on the FVI along TB_X indicates how many elements of t3 are coalesced.

4.2 Efficient Index Calculation using Pre-computed Arrays

When we load the inputs from global memory to shared memory and store the output from register to global memory, we need to figure out the global memory addresses of the inputs and the output for each thread. Because of *register tiling*, each thread needs to calculate several addresses.

When each thread calculates the addresses from the index values, expensive integer division and modulo operations inevitably need to be used. Furthermore, some registers are required to keep the index values, resulting in lower occupancy, longer operation latency as well as register spilling.

To minimize this overhead, we use pre-computed arrays that are generated before kernels are launched to get the calculated addresses of global memory for inputs such as t2 and v2 and output tensor t3. These are stored in read-only texture memory.

The extra storage for the pre-computed arrays depends on tile-sizes and the problem size. In our implementation, because each tensor contraction has identical tile-sizes, the extra storage of the pre-computed arrays is independent of the specific tensor contraction (sd*). The size of pre-computed arrays is 1.64% of the tensor's size.

In Alg. 1, from line 8 to 16, the kernel loads inputs from global memory (g_t2 and g_v2) $T_d \times T_i \times T_a \times T_i \times T_j$ and $T_d \times T_k \times T_c \times T_b$, respectively, to shared memory (s_t2 and s_v2), by using the pre-computed arrays pre_t2 and pre_v2. Then, from line 17 to 29, a thread contracts tensors from shared memory. It repeats these steps $\lceil N_d/T_d \rceil$ times. Between line 36 to 40, the kernel stores the results from register (r_t3) $T_d \times T_a$ to global memory g_t3 by using the pre-computed array pre_t3.

Algorithm 1: Pseudo-code of a tensor contraction for sd2 function

Data: pre_t2, pre_v2 and pre_t3 are pre-computed arrays to find global memory addresses of $t2$, $v2$ and $t3$, respectively.

```
1  kernel tensor_contraction(g_t3,g_t2,g_v2,pre_t2,pre_v2,pre_t3)
2      __shared__ double s_t2[T_i × T_j × T_a][T_d];
3      __shared__ double s_v2[T_d][T_k × T_c × T_b];
4      double r_t2[T_a] ;                  // a column vector (T_a × 1)
5      double r_v2[T_b] ;                     // a row vector (1 × T_b)
6      double r_t3[T_a][T_b] ;              // register tile (T_a × T_b)
7      for i = 0 to ⌈N_d/T_d⌉ do
8          // (1) Load Inputs from Global Memory to Shared Memory
9          if threadIdx.x < T_d then
10             for j = 0 to T_a do
11                 s_t2[threadIdx.y + j × (T_i × T_j)][threadIdx.x] =
                       g_t2[pre_t2[—] + (threadIdx.x)];
12             end
13             for j = 0 to T_b do
14                 s_v2[threadIdx.x][threadIdx.y + j × (T_k × T_c)] =
                       g_v2[pre_v2[—] + (threadIdx.x)];
15             end
16         end
17         __syncthreads();
18         // (2) Load Inputs from Shared Memory to Registers
19         for j = 0 to T_d do
20             for k = 0 to T_a do
21                 r_t2[k] = s_t2[—][j];
22             end
23             for k = 0 to T_b do
24                 r_v2[k] = s_v2[j][—];
25             end
26             // (3) Contract Inputs to generate Output on Registers
27             for i = 0 to T_a do
28                 for j = 0 to T_b do
29                     r_t3[i][j] += r_t2[i] * r_v2[j];
30                 end
31             end
32         end
33         __syncthreads();
34     end
35     // (4) Store the Results from Registers to Global Memory
36     for i = 0 to T_a do
37         for j = 0 to T_b do
38             g_t3[pre_t3[—]] = r_t3[i][j];
39         end
40     end
```

5 FUSION FOR SYMMETRIZED TENSOR CONTRACTIONS

In this section, we discuss our approach to fusing tensor contractions in CCSD(T) according to the two-level tiling approach.

5.1 Constraints on Fusion

First of all, in Table 1, tensor contractions in CCSD(T) have the identical left-hand side (LHS). Therefore, it is technically possible to fuse tensor contractions with different parts of input tensors.

For the nine sd2 functions in CCSD(T), a thread block requires nine pairs of t2 and v2. Because a thread block exclusively produces a part of t3 such as $T_i \times T_j \times T_k \times T_a \times T_b \times T_c$, the nine pairs of t2 and v2 are as follows:

- sd2_1: (t2) $N_d \times T_a \times T_i \times T_j$, (v2) $N_d \times T_k \times T_c \times T_b$
- sd2_2: (t2) $N_d \times T_a \times T_j \times T_k$, (v2) $N_d \times T_i \times T_c \times T_b$
- sd2_3: (t2) $N_d \times T_a \times T_i \times T_k$, (v2) $N_d \times T_j \times T_c \times T_b$
- sd2_4: (t2) $N_d \times T_b \times T_i \times T_j$, (v2) $N_d \times T_k \times T_c \times T_a$
- sd2_5: (t2) $N_d \times T_b \times T_j \times T_k$, (v2) $N_d \times T_i \times T_c \times T_a$
- sd2_6: (t2) $N_d \times T_b \times T_i \times T_k$, (v2) $N_d \times T_j \times T_c \times T_a$
- sd2_7: (t2) $N_d \times T_c \times T_i \times T_j$, (v2) $N_d \times T_k \times T_b \times T_a$
- sd2_8: (t2) $N_d \times T_c \times T_j \times T_k$, (v2) $N_d \times T_i \times T_b \times T_a$
- sd2_9: (t2) $N_d \times T_c \times T_i \times T_k$, (v2) $N_d \times T_j \times T_b \times T_a$

Then, if a thread block has the above nine pairs of partial tiles of t2 and v2, then the thread block can fuse the nine sd2 functions without storing the results from registers to global memory after finishing each tensor contraction, as shown in Fig. 5. To be specific, for each tensor contraction, thread blocks load their partial inputs from global memory to shared memory and then contract them to produce the output tensor as an intermediate result. Thus, each step is as though a single tensor contraction is performed except for storing the results from registers to global memory.

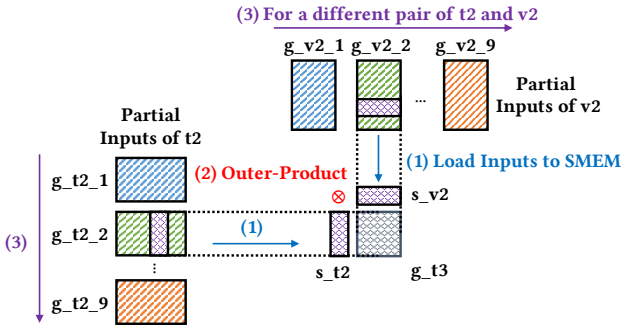


Figure 5: Overview of fusing tensor contractions in the nine sd2 functions in a thread block

However, there are two issues: (1) the size of shared memory and (2) arithmetic intensity. First, the size of shared memory used by a tensor contraction depends on the tile sizes.

In the case of sd2_1, each thread block requires $T_d \times T_a \times T_i \times T_j + T_d \times T_k \times T_c \times T_b$ shared memory space. To be specific, given tile sizes, different tensor contractions might require different amounts of shared memory space. Although we can fuse all of them if we allocate shared memory with the maximum size among tensor contractions, shared memory might be wasted to some tensor contractions which require the smaller shared memory size, resulting in lower occupancy.

Furthermore, let register tile be mapped as $\{a\} \rightarrow REG_Y$ and $\{b\} \rightarrow REG_X$. Among the nine sd2 functions, in the case of three tensor contractions—sd2_7, sd2_8, and sd2_9—one of the input tensors, v2, has indices a and b. Then, as for *register tiling*, if we have $T_a = T_b = 4$, each thread will load an element from t2 and 16 elements from v2 to produce 16 elements as the register tile.

However, for the remaining tensor contractions such as sd2_1 and sd2_2, each thread will load 4 elements from t2 and 4 elements from v2 and then produce 16 elements as the register tile.

Because of the above issues, given a set of tile sizes and a mapping dimensions to thread block and register tile, we impose the following *two constraints* on fusion:

- In a fused kernel, every tensor contraction requires an identical amount of shared memory.
- In a fused kernel, indices mapped to register tile should come from different input tensors.

Thus, tensor contractions which are compatible with the constraints can be fused in a kernel. With these simple constraints, we can extend the mapping strategy explained in Section 3 to support fusion.

With tile sizes such as $T_k = T_j = T_i = T_c = T_b = T_a = 4$ and $T_d = 16$ and a mapping $\{c, i\} \rightarrow TB_Y$, $\{k, j\} \rightarrow TB_X$, $\{a\} \rightarrow REG_Y$ and $\{b\} \rightarrow REG_X$, we can fuse 6 tensor contractions: sd2_1, sd2_2, sd2_3, sd2_4, sd2_5, sd2_6. Furthermore, with the same set of tile sizes and a different mapping such as $\{a, i\} \rightarrow TB_Y$, $\{k, j\} \rightarrow TB_X$, $\{c\} \rightarrow REG_Y$ and $\{b\} \rightarrow REG_X$, we fuse the remaining 3 tensor contractions: sd2_7, sd2_8 and sd2_9. Thus, for the nine sd2 functions, two different kernels fuse all sd2 kernels. According to the above two different mappings, we also can fuse the nine sd1 functions by two kernels such that the first kernel fuses sd1_1, sd1_2, sd1_3, sd1_7, sd1_8 and sd1_9 functions and the second kernel fuse sd1_4, sd1_5 and sd1_6 functions.

5.2 Fusing Tensor Contractions

Let us assume that we have a set of tile sizes such as $T_k = T_j = T_i = T_c = T_b = T_a = 2$ and $T_d = 4$ and a mapping such as $\{k, c\} \rightarrow TB_X$, $\{i, j\} \rightarrow TB_Y$, $\{b\} \rightarrow REG_X$ and $\{a\} \rightarrow REG_Y$. Then, given the set of tile sizes and the mapping dimensions to thread block and register tile, it will be determined which elements are produced by a thread. However, because external indices are permuted on input tensors through the nine sd2 functions, there are different ways to get inputs from shared memory for different sd2 functions.

Organize Inputs When Loading to Shared Memory. Among the sd1 and sd2 functions, both input tensors, t2 and v2, are four-dimensional tensors with 3 external indices and 1 internal index. The internal index is mapped on one dimension of 2D shared memory and the external indices are mapped on the other dimension as shown in Fig. 6.

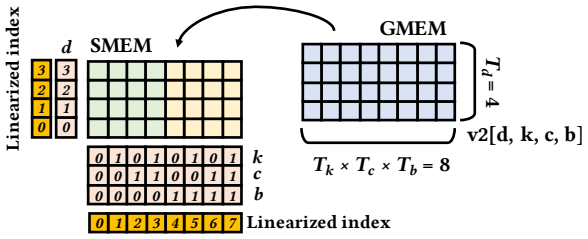


Figure 6: Illustration of organize of input tensors v2 of sd2_1

In our approach, when we map the external indices to a dimension, we choose the fastest-varying index (FVI) among the external indices as the FVI along the dimension and set the index mapped on the register tile to be the slowest-varying index (SVI) along the dimension. Figure 6 shows that, for v2[d, k, c, b] of sd2_1, where d is the FVI, d is mapped along y-axis and k, c, b are mapped along x-axis and $T_d \times T_k \times T_c \times T_b$ will be stored from global memory to shared memory.

Loading Inputs from Shared Memory via In-direction Array. According to the way to organize inputs in shared memory, each thread can load its different inputs along the nine sd2 functions on shared memory by using the nine different in-direction arrays. For example, let us look at a thread for $k = 1, c = 0, i = 0$ and $j = 0$ within a thread block, highlighted purple.

Figures 7 and 8 show how threads in a thread block load inputs for sd2_1 and sd2_2. First of all, in Figure 7, for t2[d, a, i, j] and v2[d, k, c, b] in the sd2_1 function, it requires two elements from t2 and v2 per step along T_d such as s_t2[d, 0, 0, 0] and s_t2[d, 1, 0, 0] and s_v2[d, 1, 0, 0] and s_v2[d, 1, 0, 1], where $0 \le d \le 3$, respectively.

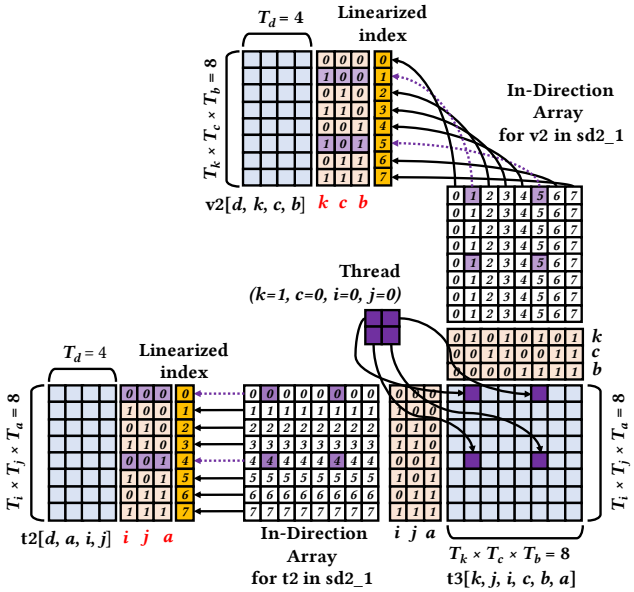


Figure 7: Loading inputs from shared memory via in-direction array for sd2_1

Secondly, in Figure 8, for t2[d, a, j, k] and v2[d, i, c, b] in the sd2_2 function, it requires two elements from t2 and v2 per step along T_d such as s_t2[d, 0, 0, 1] and s_t2[d, 1, 0, 1] and s_v2[d, 0, 0, 0] and s_v2[d, 0, 0, 1], where $0 \le d \le 3$, respectively.

5.3 Register Transposition

Register tile produced by a thread can be represented as $T_a \times T_b$, where $\{a\} \rightarrow REG_Y$ and $\{b\} \rightarrow REG_X$. However, tensor contractions in which one of inputs has a and b are not compatible to be fused by the given register tiles.

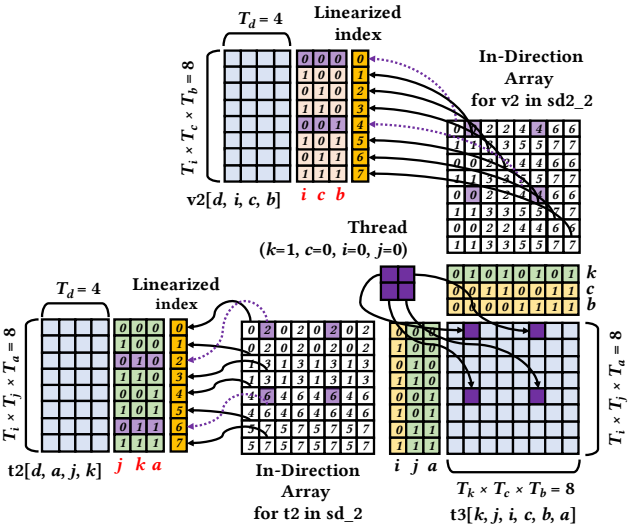


Figure 8: Loading inputs from shared memory via in-direction array for sd2_2

In this case, we use another two-level tiling such as $\{c\} \rightarrow REG_Y$ and $\{b\} \rightarrow REG_X$ to fuse tensor contractions incompatible with the given two-level tiling. Then, we can create a kernel corresponding to the another two-level tiling to fuse them.

However, under certain conditions, we can fuse the incompatible tensor contractions using *register transposition*. This condition is as follows:

(1) To keep arithmetic intensity, a target index's tile size should be equal to the given index's one.
(2) Because the given two indices come from an input tensor, the target index should come from another input tensor.

For example, let us assume that $T_i = T_j = T_k = T_a = T_b = T_c = 2$. Furthermore, let a given two-level tiling be $\{c, i\} \rightarrow TB_Y$, $\{k, j\} \rightarrow TB_X$, $\{a\} \rightarrow REG_Y$ and $\{b\} \rightarrow REG_Y$, as shown in Fig. 10. Then, we have 4×4 threads in a thread block and a thread produces 2×2 elements. To be specific, a thread block handles 8×8 elements of $t3$ such as $T_i \times T_j \times T_k \times T_a \times T_b \times T_c$.

Then, these 8×8 elements on registers through threads in a thread block will be stored to shared memory if the size of shared memory is enough to keep the elements. If not, we need to store partial elements to shared memory and then load them to the corresponding registers according to the another two-level tiling.

Fig. 9 presents how a register tile is stored to shared memory. For simplicity, let us assume that a thread block in Fig. 9 is responsible $t3[0, 0, 0, 0, 0, 0]$ such as $t3[0{:}1, 0{:}1, 0{:}1, 0{:}1, 0{:}1, 0{:}1]$. There are several ways to store and load register tiles, because we utilize shared memory, which threads in a thread block can access as temporary storage.

After storing register tiles to shared memory, each thread gets its register tiles from shared memory shown in Fig. 10. We refer to this as *register transposition* because each thread has different register tile as if register tile is transposed via shared memory.

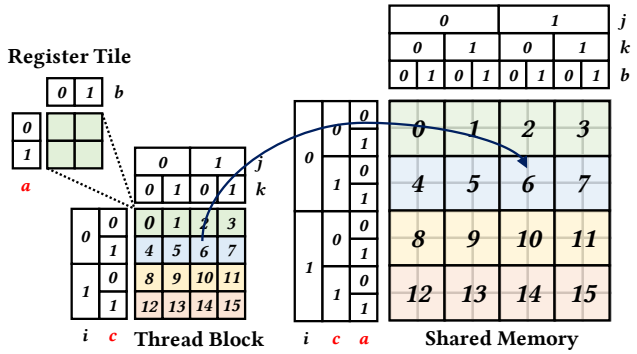


Figure 9: Example of storing register tiles to shared memory

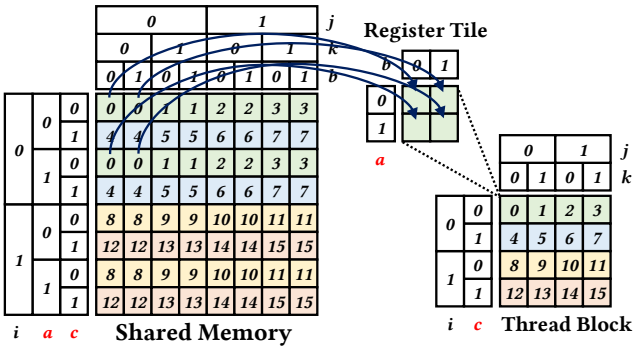


Figure 10: Illustration of loading register tiles from shared memory

Although there are two different ways to communicate values between threads such as using shared memory and the shuffle (SHFL) instructions which exchange data in a warp, in this paper, we utilized shared memory. The method to transpose register-tiles is summarized in Algorithm 2.

6 EXPERIMENTAL RESULTS

We evaluate the optimized CCSD(T) triples correction kernels on Pascal P100 and Volta V100 GPUs. Both GPUs have 16GB GPU memory and are connected to the host via PCI Express. The peak floating point performance of the Pascal and Volta GPUs are 4.7TFLOPS and 7.8TFLOPS, respectively. All codes were compiled using CUDA 9.0 and GCC 6.2. The OpenACC codes were compiled using PGI 17.9.

We consider various problem sizes shown in Table 2. These sizes are chosen based on those evaluated in prior optimization efforts and chosen in NWChem based on available memory on the GPUs. Typically, equal sizes are chosen for dimensions based on user input at the start of the computation. The block-sparse nature of the tensors (due to spin and point-group symmetries [13]) can preclude the construction of a full tile at the boundary of a block, leading to partial tiles. Among the chosen tile sizes, Size-D corresponds to such partial tiles.

Algorithm 2: Example of register transposition

```
Data: smem is shared memory for a thread block and results is
      register tile of a thread
sizeOutput = the size of a partial output produced by a thread block;
sizeSMEM = the size of shared memory used in a thread block;
for k = 0 to ⌈sizeOutput/sizeSMEM⌉ do
    // To Store the Intermediate Results from Register to Shared
       Memory
    for i = 0 to T_a do
        for j = 0 to T_b do
            smem[threadIdx.x×T_b + i][threadIdx.y×T_a + j] =
              results[i][j];
        end
    end
    __syncthreads();
    // To Load the Intermediate Results from Shared Memory to
       Register
    for i = 0 to T_c do
        for j = 0 to T_b do
            results[i][j] = smem[threadIdx.x×T_b + i] [threadIdx.y +
              j×T_c];
        end
    end
    __syncthreads();
end
```

Table 2: Problem sizes

	i, j, k, a, b, c, d(l)
Size-A	16, 16, 16, 16, 16, 16, 16
Size-B	64, 16, 16, 16, 16, 16, 16
Size-C	24, 24, 24, 24, 24, 24, 24
Size-D	15, 16, 16, 19, 20, 20, 20
Size-E	28, 28, 28, 28, 28, 28, 28

We evaluated the 18 kernels for sd_1 and sd_2, generated by PPCG (The Polyhedral Parallel Code Generator) with the problem sizes in Table 2. However, none of the variants achieved more than 10GFLOPS. We studied the code generated by PPCG and observed several reasons. Accesses to the v2 tensor were not coalesced by PPCG. Shared memory and register tiling were not exploited. We observed several `__syncthreads` operations that could have been removed. Many of the loops had loop-bound expressions involving min expressions, which increases branch cost and interferes with unrolling. Finally, PPCG does not explore fusion for these kernels. This demonstrates that traditional loop optimization techniques are not successful in optimizing the triples correction. Due to these reasons, we do not discuss PPCG further.

We compared our implementation with the kernels extracted from NWChem 6.6 [12, 14], those generated using OpenACC directives and those running by TAL-SH (Tensor Algebra Library for Shared Memory Computers). We evaluated all 18 O(N^7) contractions in the triples corrections to determine the benefits of the direct method of evaluation.

Table 3: Parameters used in Fully-Fused and Partially-Fused Kernels

	Functions	TB_X	TB_Y	REG_X	REG_Y	$T_i \ldots T_c$	$T_{d(l)}$
sd1	4,5,6,7,8,9	k,j	c,i	b	a	4	16
	1,2,3	k,j	a,i	b	c	4	16
sd2	1,2,3,4,5,6	k,j	c,i	b	a	4	16
	7,8,9	k,j	a,i	b	c	4	16

In addition to improving performance, the key benefit of the direct method is that it enables effective fusion across the 9 symmetrization kernels. We implemented two fused variants: Fully-Fused and Partially-Fused kernel versions based on the mapping in Table 3.

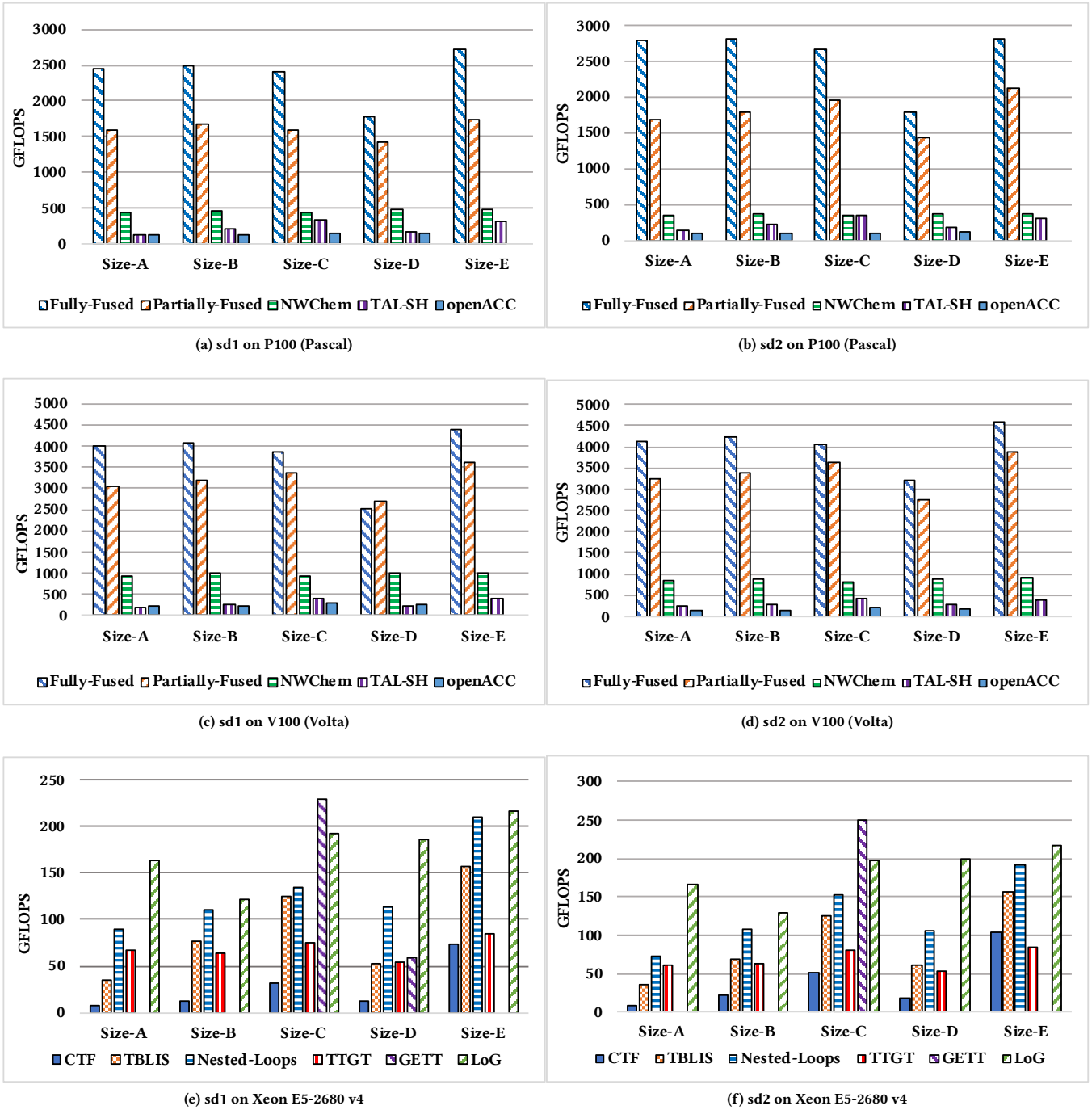
In the Partially-Fused version, for sd1, the kernels sd1_4,5,6,7,8,9 are fused to form the first kernel and kernels sd1_1,2,3 are fused to form the second kernel. Similarly for sd2 the two fused kernels correspond to sd2_1,2,3,4,5,6 and sd2_7,8,9 respectively (details in Sec 5.2). In the Fully-Fused kernel version, all sd* contractions are fused using techniques described in Sec 5.2 and Sec 5.3. The output tensor is only moved once from registers to global-memory, but it requires one register-tile transpose using shared-memory, as described earlier. The Partially-Fused kernel does not incur any register transpose overhead, but requires more global memory data movement to write and read back the result tensor elements between the partially fused sets of contractions. For both versions, we used $T_i = T_j = T_k = T_a = T_b = T_c = 4$ and $T_d = T_h = 16$ as tile-sizes.

Figure 11 shows the performance of the two fused variants with the NWChem kernels, TAL-SH and OpenACC implementations. We observe that all variants that employ specialized implementation strategies perform better than the implementation using OpenACC. In all cases, both fusion strategies considered perform better than the NWChem kernel. In addition, the 1-kernel version achieves the best performance in all cases, clearly demonstrating the need for fusion across the symmetrization operations.

We find that both fused variants achieve their worst performance for problem sizes with partial tiles (Size-D). This is due to inefficiency in handling incomplete thread blocks. Despite this inefficiency, we find that our approach is 2.3–4.8× faster than the NWChem kernels.

On Pascal, across all problem sizes considered, the NWChem kernels achieve a maximum performance of 1.0TFLOPS. The Fully-Fused Kernel and Partially-Fused Kernel versions achieve a maximum performance of 2.8TFLOPS and 2.1TFLOPS, respectively. In addition, the fused versions achieve more consistent performance across problem sizes. For example, while the NWChem version's performance varies in the 338–1004GFLOPS range, the Fully-Fused kernel version achieves performance in the 1.77–2.81TFLOPS range. This shows that the performance of the fused version is less sensitive to problem size.

Interestingly, on the newer Volta architecture, the NWChem kernels achieve more consistent performance (818–1004 GFLOPS across problem sizes). However, the fused versions perform better still, achieving between 2.5 and 4.5 TFLOPS, with a peak of 4.5



(a) sd1 on P100 (Pascal)

(b) sd2 on P100 (Pascal)

(c) sd1 on V100 (Volta)

(d) sd2 on V100 (Volta)

(e) sd1 on Xeon E5-2680 v4

(f) sd2 on Xeon E5-2680 v4

Figure 11: Performance (in GFLOPS) of sd1 and sd2 functions on on V100 (Volta),P100 (Pascal) GPUs and Xeon E5-2680 v4

TFLOPS. In general, the architectural improvements in Volta result in performance improvements for all variants evaluated, including OpenACC. However, only the fusion strategy presented in this paper achieves a significant fraction of the peak 7.8TFLOPS available on the GPU.

On the P100 machine, without full fusion (i.e., no register-transpose) the performance drops by 36%. Without any register tiling, the performance drops further by 54%, i.e. a performance loss of 70% relative to the register-tiled, fully-fused version.

Figure 11 (e) and (f) show the evaluation of the 18 kernels on a multi-core CPU using state-of-art alternative implementations. This is used to provide a comparative baseline for the GPU implementation. We used publicly available compilers/libraries for tensor contraction on CPUs. TCCG [15] is a Tensor Contraction Code Generator that offers several alternative direct implementations (without explicit transpose) for tensor contractions. *Nested-Loops* implements tensor contractions using a set of nested loops. TTGT implements tensor contractions using GEMM and transpose operations. TBLIS [16] and GETT [17] implement multicore kernels to directly perform the contraction by essentially fusing needed data layout transforms on the fly on slices of the tensors and then performing matrix-multiplication on the slices. LoG [15] implements tensor contractions as loops over GEMM calls. The experiments were performed using all cores of a 28-core 2.4GHz Xeon E5-2680v4 processor. CTF (Cyclops Tensor Framework) [18] is a distributed-memory framework for tensor contractions supports distributed processing, We evaluate it on all cores in 1, 2, 4, and 8 nodes and report the best performance. We note that none of these methods employ the fusion strategy presented in this paper, which likely is the reason that they achieve less than 250 GFLOPs, or 25% of the 1.075 TeraFlop peak of the multicore system. In contrast, our fully fused version achieves much higher absolute performance as well as >50% of peak GFLOPs.

7 RELATED WORK

Tensor contractions such as the ones in CCSD(T) can be described by perfectly nested loop nests. While automated optimization techniques are typically used to optimize such loop nests [19], two challenges arise when using these techniques to optimizes triples correction: (1) the high dimensionality and small dimension sizes make asymptotic cost models inaccurate (2) typical loop optimizers do not take symmetrization into account.

This has motivated the design of custom code generation approaches to optimize tensor contraction expressions. These approaches fall into two categories. In the first category, the approach involves permuting tensors into an index order that enables the use of GEMM routines from optimized BLAS libraries [2, 20–22]. This has led to the development and use of efficient tensor transposition libraries [23–30] and variants of GEMM that support batched and strided matrix-multiplication operations [31, 32]. In addition, several approaches to efficient execution of BLAS kernels have been developed [33, 34].

While these have been shown to be highly effective in optimizing general tensor contraction expressions in Coupled Cluster methods, the large number of symmetrization operations encountered in the triples correction put far greater burden on the transposition routines with low operation intensity. Our evaluation demonstrates performance improvements as compared to the GEMM-based approach in NWChem.

The other approach, referred to as the direct approach, involves the generation of custom optimized loop structures for tensor contractions. Ma et al. developed loop structures using CUDA to optimize the triples kernels [11, 12]. Matthews [35] presented a domain-specific compiler for tensor contraction expressions. These approaches do not exploit symmetry in tensor contractions.

Schatz et al. proved that exploiting symmetry in tensor contraction expressions can improve storage and computational requirements and present CPU implementations [36]. Our approach differs from this work in the following significant ways: (a) Schatz et al. focus on storage and computation optimizations for distributed tensors, rather than contractions of individual dense tensor tiles, (b) they do not present fusion across the symmetrization operations, (c) an efficient GPU algorithm is not presented, and (d) they do not consider or evaluate the high-dimensional triples correction kernels.

8 CONCLUSIONS AND FUTURE WORK

We presented a novel strategy for executing symmetric tensor contractions in CCSD(T) on GPUs. The approach involved kernel-level optimizations to efficiently utilize GPU resources, coupled with fusion across the symmetrization kernels to improve the number of compute operations per data moved. We also presented a novel register-level transpose operation to avoid moving data to GPU global memory.

Experimental evaluation demonstrated significant performance improvements as compared to existing alternatives, achieving over 60% of peak floating point performance on both Pascal P100 and Volta V100 GPUs. The algorithms presented can be directly integrated into a distributed-memory parallel implementation of the triples correction, where each tile of the six-dimensional tensor is computed in parallel.

As part of our future work, we plan to make the code publicly available and integrate these kernels into NWChem to evaluate end-to-end application performance.

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 and 1513120, by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under awards 71648 and DE-SC0014135, and 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] 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.

[2] M. Valiev, E. Bylaska, N. Govind, K. Kowalski, T. Straatsma, H. van Dam, D. Wang, J. Nieplocha, E. Apra, T. Windus, and W. de Jong, "Nwchem," 2017. [Online]. Available: http://www.nwchem-sw.org

[3] E. Aprà, A. P. Rendell, R. J. Harrison, V. Tipparaju, W. A. deJong, and S. S. Xantheas, "Liquid water: obtaining the right answer for the right reasons," in *Proceedings of the conference on high performance computing networking, storage and analysis*. ACM, 2009, p. 66.

[4] E. Aprà, M. Klemm, and K. Kowalski, "Efficient implementation of many-body quantum chemical methods on the intel® xeon phi coprocessor," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, 2014, pp. 674–684.

[5] K. Kowalski, S. Krishnamoorthy, R. M. Olson, V. Tipparaju, and E. Aprà, "Scalable implementations of accurate excited-state coupled cluster theories: application of high-level methods to porphyrin-based systems," in *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*, 2011, pp. 72:1–72:10.

[6] R. Bartlett and M. Musiał, "Coupled-cluster theory in quantum chemistry," *Reviews of Modern Physics*, vol. 79, no. 1, pp. 291–352, 2007.

[7] G. Baumgartner, A. Auer, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Hirata, S. Krishnamoorthy *et al.*, "Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 276–292, 2005.

[8] A. Hartono, Q. Lu, T. Henretty, S. Krishnamoorthy, H. Zhang, G. Baumgartner, D. E. Bernholdt, M. Nooijen, R. Pitzer, J. Ramanujam *et al.*, "Performance optimization of tensor contraction expressions for many-body methods in quantum chemistry," *The Journal of Physical Chemistry A*, vol. 113, no. 45, pp. 12 715–12 723, 2009.

[9] P.-W. Lai, K. Stock, S. Rajbhandari, S. Krishnamoorthy, and P. Sadayappan, "A framework for load balancing of tensor contraction expressions via dynamic task partitioning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 13.

[10] K. Z. Ibrahim, S. W. Williams, E. Epifanovsky, and A. I. Krylov, "Analysis and tuning of libtensor framework on multicore architectures," in *High Performance Computing (HiPC), 2014 21st International Conference on*. IEEE, 2014, pp. 1–10.

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

[12] 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.

[13] S. Hirata, "Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories," *The Journal of Physical Chemistry A*, vol. 107, no. 46, pp. 9887–9897, 2003.

[14] W. Ma, S. Krishnamoorthy, O. Villa, and K. Kowalski, "Gpu-based implementations of the noniterative regularized-ccsd (t) corrections: applications to strongly correlated systems," *Journal of chemical theory and computation*, vol. 7, no. 5, pp. 1316–1327, 2011.

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

[16] D. A. Matthews, "High-performance tensor contraction without transposition," *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. C1–C24, 2018. [Online]. Available: https://doi.org/10.1137/16M108968X

[17] P. Springer and P. Bientinesi, "Design of a high-performance gemm-like tensor–tensor multiplication," *ACM Trans. Math. Softw.*, vol. 44, no. 3, pp. 28:1–28:29, Jan. 2018. [Online]. Available: http://doi.acm.org/10.1145/3157733

[18] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel, "Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 813–824.

[19] S. Verdoolaege, "Ppcg," 2017. [Online]. Available: git://repo.or.cz/ppcg.git

[20] P. Springer and P. Bientinesi, "Design of a high-performance gemm-like tensor-tensor multiplication," *arXiv preprint arXiv:1607.00145*, 2016.

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

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

[23] 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*, ser. ARRAY 2017. New York, NY, USA: ACM, 2017, pp. 56–62.

[24] 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*, ser. ARRAY 2016. New York, NY, USA: ACM, 2016, pp. 41–46.

[25] A. Hynninen and D. I. Lyakh, "cuTT: A High-Performance Tensor Transpose Library for CUDA Compatible GPUs," *CoRR*, vol. abs/1705.01598, 2017. [Online]. Available: http://arxiv.org/abs/1705.01598

[26] Q. Lu, S. Krishnamoorthy, and P. Sadayappan, "Combining Analytical and Empirical Approaches in Tuning Matrix Transposition," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '06. New York, NY, USA: ACM, 2006, pp. 233–242.

[27] L. Wei and J. Mellor-Crummey, "Autotuning tensor transposition," in *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, May 2014, pp. 342–351.

[28] B. Catanzaro, A. Keller, and M. Garland, "A Decomposition for In-place Matrix Transposition," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: ACM, 2014, pp. 193–206.

[29] G. Mateescu, G. H. Bauer, and R. A. Fiedler, "Optimizing Matrix Transposes Using a POWER7 Cache Model and Explicit Prefetching," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 2, pp. 68–73, Oct. 2012. [Online]. Available: http://doi.acm.org/10.1145/2381056.2381073

[30] J. L. Jodra, I. Gurrutxaga, and J. Muguerza, "Efficient 3D Transpositions in Graphics Processing Units," *Int. J. Parallel Program.*, vol. 43, no. 5, pp. 876–891, Oct. 2015.

[31] J. Dongarra, S. Hammarling, N. J. Higham, S. D. Relton, and M. Zounon, "Optimized batched linear algebra for modern architectures," in *European Conference on Parallel Processing*. Springer, 2017, pp. 511–522.

[32] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "Libxsmm: accelerating small matrix multiplications by runtime code generation," in *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 2016, pp. 981–991.

[33] F. D. Igual, E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, R. A. Van De Geijn, and F. G. Van Zee, "The flame approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations," *Journal of Parallel and Distributed Computing*, vol. 72, no. 9, pp. 1134–1143, 2012.

[34] F. D. Igual, G. Quintana-Ortí, and R. A. Van De Geijn, "Level-3 blas on a gpu: Picking the low hanging fruit," in *AIP Conference Proceedings*, vol. 1504, no. 1. AIP, 2012, pp. 1109–1112.

[35] D. A. Matthews, "High-performance tensor contraction without blas," *arXiv preprint arXiv:1607.00291*, 2016.

[36] M. D. Schatz, T. M. Low, R. A. van de Geijn, and T. G. Kolda, "Exploiting symmetry in tensors for high performance: Multiplication with symmetric tensors," *SIAM J. Scientific Computing*, vol. 36, no. 5, 2014. [Online]. Available: https://doi.org/10.1137/130907215