A Sparse Tensor Benchmark Suite for CPUs and GPUs

Jiajia Li*, Mahesh Lakshminarasimhan†, Xiaolong Wu‡, Ang Li*, Catherine Olschanowsky§, Kevin Barker*
† *University of Utah*, Salt Lake City, UT, USA. maheshl@cs.utah.edu
‡ *Purdue University*, West Lafayette, IN, USA. wu1565@purdue.edu
§ *Boise State University*, Boise, ID, USA. catherineolschan@boisestate.edu
* *Pacific Northwest National Laboratory*, Richland, WA, USA.
{Jiajia.Li, Ang.Li, Kevin.Barker}@pnnl.gov

Abstract—**Tensor computations present significant performance challenges that impact a wide spectrum of applications ranging from machine learning, healthcare analytics, social network analysis, data mining to quantum chemistry and signal processing. Efforts to improve the performance of tensor computations include exploring data layout, execution scheduling, and parallelism in common tensor kernels. This work presents a benchmark suite for arbitrary-order sparse tensor kernels using state-of-the-art tensor formats: coordinate (COO) and hierarchical coordinate (HiCOO) on CPUs and GPUs. It presents a set of reference tensor kernel implementations that are compatible with real-world tensors and power law tensors extended from synthetic graph generation techniques. We also propose Roofline performance models for these kernels to provide insights of computer platforms from sparse tensor view. This benchmark suite along with the synthetic tensor generator is publicly available [1].**

Index Terms—**sparse tensors, benchmarking, data analysis, tensor decomposition, GPU**

I. Introduction

Tensors, multi-dimensional arrays that are often sparse, are utilized by a large number of critical applications that span a range of domain areas. These include quantum chemistry, healthcare analytics, social network analysis, data mining, signal processing, machine learning, and more. Operations on sparse tensors tend to dominate the execution-time of these applications. Understanding the performance characteristics of different implementation approaches is of paramount importance. This paper presents a benchmark suite specifically for that purpose. The suite provides implementations of common tensor kernels using state-of-the-art sparse tensor data structures and a variety of real and synthetic sparse tensors as its input dataset.

Given the heterogeneity in available hardware resources for high performance computing (HPC), it is non-trivial to answer questions about the potential for sparse tensor algorithms to be efficiently ported to various hardware. The difficulty of planning for the irregular parallelism that results from operating on sparse data structures is compounded by the availability of Graphics Processing Units (GPUs), vectorizing units, Field Programmable Gate Arrays (FPGAs), and potentially Tensor Processing Units (TPUs). A set of important tensor kernels with associated implementations eases the exploration of this space.

Optimizing the performance of tensor applications is challenging due to several application characteristics, named in the studies [1], [2], [3] and briefly outlined here for completeness: *the curse of dimensionality, mode orientation, tensor transformation, irregularity, and arbitrary tensor orders (or dimensions).*

Tensors are, by definition, multidimensional. The *curse of dimensionality* manifests itself as large computational and storage overheads required to accommodate the exponential growth of elements that occurs in some operations. For instance, a Kronecker product results in exponential expansion of space requirements. Compounding this issue is the increased interest in applications involving a large number of dimensions [4], [5], [6], [7], [8]. The data structures supporting sparse tensors and the required tensor operations are often mode specific, where each dimension of a tensor is referred to as a mode. Different data structures supporting sparse tensors favor iterating over specific mode order, *mode orientation*. There is a tradeoff that must be made between space requirements and enjoying good performance in multiple representations of various mode sequences. *Tensor transformation* is traditionally used to implement tensor operations by casting them as a set of matrix operations and utilizing highly tuned linear algebra libraries. However, the transformation process brings non-trivial overhead to the execution of a tensor operation. Mitigating this cost has become attractive for researchers in tensor linear algebra and their applications [9], [10], [11], [12], [13]. *Irregularity* in memory access patterns and in tensor shape makes poor use of memory subsystems and complicates code, especially for sparse data. Optimizations are typically best suited for a specific dimensionality, such as third-order, but most tensor operations are required to handle *arbitrary tensor orders*.

Beyond these, challenges associated with all benchmarks also apply, which include *completeness, diversity, extendibility, reproducibility, and comparability* across implementations. Comparisons across research groups are improved by using a standard set of kernels and inputs. Using that set as a starting point, optimizations can be applied and effectively compared.

Our benchmark suite consists of a set of reference imple-

[1]https://gitlab.com/tensorworld/pasta

mentations from various tensor applications, each of which show different computational behaviors. We keep the implementations simple yet effective; the benchmark represents a general case where the primary computation is not obfuscated by optimization attempts. Much like two-dimensional sparse matrices, the *data layout*, or the data structure used to hold a sparse tensor, has a significant impact on performance and storage [14], [15]. It also has a significant impact on how the control flow for a given operation must be executed and its memory footprints. We implement two sparse tensor formats: the most popular and mode-generic coordinate (COO) format and a newly proposed, more compressed hierarchical coordinate (HiCOO) format [16] and their variants to represent general or dense structured, arbitrary sparse tensors. Beyond the implementation diversity, platform and workload (or input) diversity is also critical to gain insights from a benchmark suite. We implement the same set of tensor kernels, by directly operating on non-zero entries to avoid tensor-matrix transformations, on CPUs and GPUs to provide a good understanding for users. Different inputs of an algorithm usually obtain different performance due to their diverse data sizes and patterns. This phenomenon is more obvious on sparse problems because their algorithm behavior largely depends on the features of data. Besides evaluating limited and hard-to-obtain real-world tensors, mimicking some application characters to generate more datasets is valuable for benchmarking. We create power law tensors extended from synthetic graph generation techniques. This suite provides a performance baseline and a starting point for new optimization strategies. It is easy to adopt new implementations, operations, and formats from users and to be adapted in a communication scheme.

This work is a continuous effort on the PASTA sparse tensor benchmark suite [17]. Our contributions, beyond the first sparse tensor benchmark suite for GPUs, include:

- reference and direct tensor-based implementations of five tensor kernels: TEW, TS, TTV, TTM, and MTTKRP, in COO format for GPUs and HiCOO format for CPUs and GPUs; (Sections II and III)
- application of HiCOO to more tensor operations and an extension of it to more flexible variations; (Section III)
- synthetic tensor generation based on Kronecker and power law generators; (Section IV)
- Roofline performance models for two Intel CPU and two NVIDIA GPU platforms to analyze the tensor kernels; and
- insights gained from thorough experiments and analysis of the performance. (Section V)

II. Tensor Benchmarks

Tensors are increasingly employed in computations across a spectrum of application areas. This benchmark suite represents a set of fundamental operations chosen by examining a range of composite operations commonly used in these applications and studied in the community. The following text provides the definition of each operation, the motivation for its inclusion, and its applications.

Notationally, we represent tensors as calligraphic capital letters, e.g., $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$; matrices by boldface capital letters, e.g., $\mathbf{U} \in \mathbb{R}^{I\times J}$; vectors by boldface lowercase letters, e.g., $\mathbf{x} \in \mathbb{R}^{I}$; and scalars by lowercase letters, such as x_{ijk} for the (i, j, k)-element of a third-order tensor $\mathcal{X}$. A *slice* is a two-dimensional cross-section of a tensor, obtained by fixing all indices but two, e.g., $\mathbf{S}_{::k} = \mathcal{X}(:, :, k)$. A *fiber* is a vector extracted from a tensor along a certain mode, selected by fixing all indices but one, e.g., $\mathbf{f}_{:jk} = \mathcal{X}(:, j, k)$.

A. Tensor Element-Wise Operations

Tensor element-wise (TEW) operations include addition, subtraction, multiplication, and division, that are applied to every corresponding pair of elements from two tensor objects. For example, element-wise tensor addition of $\mathcal{X}, \mathcal{Y} \in \mathbb{R}^{I_1\times\cdots\times I_N}$ is $\mathcal{Z} = \mathcal{X} + \mathcal{Y}$. Similarly for element-wise tensor subtraction $\mathcal{Z} = \mathcal{X} - \mathcal{Y}$, multiplication $\mathcal{Z} = \mathcal{X} \circ \mathcal{Y}$, and division $\mathcal{Z} = \mathcal{X} \oslash \mathcal{Y}$.

This operation is trivially implemented when the two input tensors having exactly the same non-zero pattern, tensor order (i.e., number of dimensions) and shape (i.e., dimension sizes). We also support more general cases those require iterating over both tensors and their matching non-zero elements as the execution proceeds for tensors in different tensor orders and/or shapes, where predicting the storage of the output $\mathcal{Z}$ is needed.

B. Tensor-Scalar Operations

A Tensor-Scalar (TS) operates between the non-zero values of a tensor and a scalar through addition (TSA), subtraction (TSS), multiplication (TSM), and division (TSD). For example, tensor-scalar multiplication of $\mathcal{X} \in \mathbb{R}^{I_1\times\cdots\times I_N}$ with scalar s is $\mathcal{Y} = \mathcal{X} \times s$. This benchmark suite implements TSA and TSM only, which are sufficient to support all the four operations.

TEW and TS are commonly used in machine learning, quantum chemistry, and so on. The tensor convolution operation in convolutional neural network (CNN) is a combination of TEW and TSM [18]; space mapping in quantum chemistry also involves these two. TEW and TS are simple operations and can be implemented along with other tensor operations. We consider them separately in this benchmark suite because of their different computational behavior (Will be shown in Table I).

C. Tensor-Times-Vector Product

The Tensor-Times-Vector (TTV) in mode n, $\mathcal{Y} = \mathcal{X} \times_n \mathbf{v}$, is the multiplication of a tensor $\mathcal{X} \in \mathbb{R}^{I_1\times\cdots\times I_n\times\cdots\times I_N}$ with a vector $\mathbf{v} \in \mathbb{R}^{I_n}$, along mode n. Element-wise,

$$y_{i_1\cdots i_{n-1}i_{n+1}\cdots i_N} = \sum_{i_n=1}^{I_n} x_{i_1\cdots i_{n-1}i_n i_{n+1}\cdots i_N} v_{i_n} \qquad (1)$$

This results in a $I_1 \times \cdots \times I_{n-1} \times I_{n+1} \times \cdots \times I_N$ tensor which has one less dimension. TTV is a critical computational kernel of the tensor power method [19], [20], an approach for orthogonal tensor decomposition, that decomposes a symmetric tensor into a collection of orthogonal vectors with corresponding weights. The tensor power method is used in machine learning and signal processing applications.

D. Tensor-Times-Matrix Product

The Tensor-Times-Matrix (TTM) in mode n, also known as the n-mode product, is the multiplication of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N}$ with a matrix $\mathbf{U} \in \mathbb{R}^{I_n \times R}$, along mode n, and is denoted by $\mathcal{Y} = \mathcal{X} \times_n \mathbf{U}$. [2] This results in a $I_1 \times \cdots \times I_{n-1} \times R \times I_{n+1} \times \cdots \times I_N$ tensor, and its operation is defined as

$$y_{i_1 \cdots i_{n-1} r i_{n+1} \cdots i_N} = \sum_{i_n=1}^{I_n} x_{i_1 \cdots i_{n-1} i_n i_{n+1} \cdots i_N} u_{i_n r}. \quad (2)$$

TTM is a special case of tensor contraction, a multiplication between two tensors in common mode(s). We consider TTM specifically because: 1) it is commonly used in popular tensor decompositions, such as the Tucker decomposition, for a variety of applications, including (social network, electrical grid) data analytics, numerical simulation, machine learning, recommendation systems, personalized web search, etc. [1], [21], [19], [22]; 2) the behavior of tensor contraction largely depends on which mode(s) to be contracted on; this creates difficulties for benchmarking. Also, note that R is typically much smaller than I_n in low-rank decompositions, typically $R < 100$.

E. Matriced Tensor-Times-Khatri-Rao Product

MTTKRP, matricized tensor times Khatri-Rao product, is a matricized tensor times the Khatri-Rao product of matrices. For an Nth-order tensor $\mathcal{X}$ and given matrices $\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}$, the mode-$n$ MTTKRP is

$$\tilde{\mathbf{U}}^{(n)} = \mathbf{X}_{(n)} \left(\mathbf{U}^{(N)} \odot \cdots \odot \mathbf{U}^{(n+1)} \odot \mathbf{U}^{(n-1)} \odot \cdots \odot \mathbf{U}^{(1)} \right), \quad (3)$$

where $\mathbf{X}_{(n)}$ is the mode-n matricization of tensor $\mathcal{X}$, $\odot$ is the Khatri-Rao product. The Khatri-Rao product is a "matching column-wise" Kronecker product between two matrices. Given matrices $\mathbf{A} \in \mathbb{R}^{I \times R}$ and $\mathbf{B} \in \mathbb{R}^{J \times R}$, their Khatri-Rao product is denoted by $\mathbf{C} = \mathbf{A} \odot \mathbf{B}$,

$$\mathbf{C} = \mathbf{A} \odot \mathbf{B} = \left[\mathbf{a}_1 \circ \mathbf{b}_1, \mathbf{a}_2 \circ \mathbf{b}_2, \ldots, \mathbf{a}_R \circ \mathbf{b}_R \right], \quad (4)$$

where $\mathbf{C} \in \mathbb{R}^{(IJ) \times R}$, $\mathbf{a}_r$ and $\mathbf{b}_r$, $r = 1, \ldots, R$ are columns of $\mathbf{A}$ and $\mathbf{B}$, $\circ$ is the outer product of vectors, a special case of Kronecker product. However, the Khatri-Rao and Kronecker products typically require redundant computation or extra storage to hold matrix operands, so in practice, these operations tend to be not implemented directly but rather integrated into sparse tensor operations.

MTTKRP is the most computational expensive kernel in CANDECOMP/PARAFAC decomposition (CPD), another popular tensor decomposition. CPD also has a wide application in (healthcare, social network, brain signal, electrical grid) data analytics, machine learning, recommendation systems, signal processing, personalized web search, quantum chemistry, and other domains [1], [21], [19], [22].

Because of the varying computational behavior (will be shown in Table I) of the above operations, we integrate them as a benchmark suite to help evaluate underlying hardware.

III. Tensor Formats and Kernel Implementations

Much like sparse matrices, sparse tensors are expressed using different formats. The best choice of format depends on the sparsity pattern of a tensor, operations applied, and the time required to translate between them. The common default format for sparse tensors is coordinate (COO) format. New formats have been developed including compressed sparse fiber (CSF) [23], balanced and mixed-mode CSF (BCSF, MM-CSF) [24], [25], flagged COO (F-COO) [26], and hierarchical coordinate (HiCOO) [16] for general sparse tensors, and mode-generic and -specific formats for structured sparse tensors [27]. PASTA suite [17] has realized the five tensor operations based on the COO format on multicore CPUs. Our benchmark suite further supports HiCOO for general sparse tensors and their variants for semi-sparse tensors with dense dimension(s), and extends to GPUs for COO- and HiCOO-based operations.

We chose COO and HiCOO formats because of their *mode generic* property, described in the work [16]. For a mode generic format, only one tensor representation is needed for all tensor computations in different modes, which is commonly required by tensor methods (e.g., CPD and Tucker decompositions). COO is the most popular format used in many tensor libraries, e.g., Tensor Toolbox [28], Tensorlab [29], TACO [30], [31], and ParTI [32]. HiCOO, a newly proposed format, obtains good compression and state-of-the-art tuned performance on CPUs [16]. Other formats especially CSF will be considered for our benchmark suite in the near future. This section overviews our supported formats and their corresponding parallel implementations for tensor kernels.

Our implementations directly operate on sparse tensor nonzero entries to avoid tensor-matrix transformations. Besides, we use more pre-processing to trade for less kernel computation time, which keeps this suite more efficient and relatively easy-to-analyze compared to pillar libraries, e.g., Tensor Toolbox [28] and TensorLab [29]. Table I presents the operational intensity of each kernel using a cubical third-order sparse tensor, while all the implementations in the benchmark suite support arbitrary tensor orders. Operational intensity (OI) is the ratio of bytes required per floating point operation for a given computation. The "memory access" column counts the maximal bytes needed (upper-bound) because of irregular and unpredictable memory access. Data reuse could happen if its access has or gains a good localized pattern naturally or from reordering techniques [23], [33]. Thus, their performance could be improved due to reductions in memory pressure. We name all tensor algorithms in the pattern of "[Tensor Format]-[Tensor Kernel]-[Parallelization]" in the following context.

[2]Our convention for the dimensions of $\mathbf{U}$ differs from that of Kolda and Bader's definition [1]. In particular, we transpose the matrix modes $\mathbf{U}$, which leads to a more efficient TTM under the row-major storage convention of the C language.

TABLE I

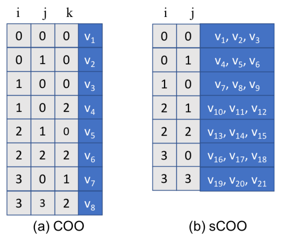
THE ANALYSIS OF KERNEL ALGORITHMS FOR THIRD-ORDER CUBICAL TENSORS ($\mathcal{X} \in \mathbb{R}^{I\times I\times I}$). WE CONSIDER ALL INPUT TENSORS WITH M NON-ZERO ENTRIES AND M_F FIBERS, $I \ll M_F \ll M$. THE INDICES USE 32 BITS, AND VALUES ARE SINGLE-PRECISION FLOATING-POINT NUMBERS WITH 32 BITS AS WELL.

Kernels	Work (#Flops)	Upper-Bound Memory Access (#Bytes) COO	HiCOO	OI
TEW	M	$12M$	$12M$	$1/12$
TS	M	$8M$	$8M$	$1/8$
TTV	$2M$	$12M + 12M_F$	$12M + 12M_F$	$\sim 1/6$
TTM	$2MR$	$4MR + 4M_FR$ $+8M + 8M_F$	$4MR + 4M_FR$ $+8M + 8M_F$	$\sim 1/2$
MTTKRP	$3MR$	$12MR + 16M$	$12Rmin\{n_bM_B, M\}$ $+7M + 20n_b$	$\sim 1/4$

A. Coordinate Format (COO)

Coordinate format is commonly used to store sparse matrices and tensors. It does not require or guarantee any particular ordering of data. Data values are stored in a one-dimensional array, no matter how many dimensions are represented in the data. For each dimension an additional index array is added to indicate the position of the value in that dimension. Figure 1(a) gives an example that a general third-order sparse tensor requires three index arrays. The storage space of an Nth-order COO tensor $\mathcal{X} \in \mathbb{R}^{I_1\times\cdots\times I_N}$ with M non-zeros is $4(N+1)M$ bytes consisting of 32-bit indices and single-precision floating-point values.

We also describe a variant of COO format (semi-sparse COO, sCOO) for a semi-sparse sparse tensor with dense modes [13], [27], which will be used in TTM. A dense mode means the fibers on it are all dense vectors. sCOO stores the dense mode(s) as dense array(s) and the rest modes remain the same as in the COO format, as shown by another example tensor in Figure 1(b), where the mode k is dense.



i	j	k	
0	0	0	v_1
0	1	0	v_2
1	0	0	v_3
1	0	2	v_4
2	1	0	v_5
2	2	2	v_6
3	0	1	v_7
3	3	2	v_8

(a) COO

i	j	
0	0	v_1, v_2, v_3
0	1	v_4, v_5, v_6
1	0	v_7, v_8, v_9
2	1	v_{10}, v_{11}, v_{12}
2	2	v_{13}, v_{14}, v_{15}
3	0	v_{16}, v_{17}, v_{18}
3	3	v_{19}, v_{20}, v_{21}

(b) sCOO

Fig. 1. COO format for a general sparse tensor and sCOO format [13] for a semi-sparse tensor.

B. Implementations based on COO

The implementations of TEW and TS directly follow their definitions: one loop over all non-zero values to do the corresponding computation. In the pre-processing stage, we allocate and set indices for the output tensor due to its easy-to-predict non-zero pattern [3]. TEW and TS have the smallest operational intensity: $1/12$ and $1/8$. We will use TTV algorithms on CPUs and GPUs as representatives to explain the similar TTM algorithms; TTM algorithms can be found in [13], [34], and MTTKRP algorithms can be found in literatures and libraries [16], [28], [29], [32]. For all operations except MTTKRP, we have a pre-processing stage to allocate the space of the output tensor with their indices.

1) Multicore CPU: We use a *sparse-dense property* for a sparse tensor times a dense vector/matrix (TTV and TTM), introduced by Li et al. [13]. That is, if the computation is between a sparse mode of a tensor and a dense vector in TTV or from the dense matrix in TTM, this mode will become dense in the output; and the other modes remain the same non-zero distribution (a.k.a. sparsity) with the input tensor. This property makes pre-allocating space for the outputs of TTV and TTM possible, with the help of the sCOO format for semi-sparse tensors. Introducing this property is good for parallelization by avoiding output data race and dynamic memory allocation, which are especially useful for GPU implementations.

Algorithm 1 COO-TTV-OMP algorithm [17].

```
Input: A third-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$ with $M$ non-zeros in COO
    format, dense vector $\mathbf{V} \in \mathbb{R}^K$, and an integer $n$ (= 3 in this pseudocode);
Output: Sparse tensor $\mathcal{Y} \in \mathbb{R}^{I\times J}$ in COO format;
                                                        $\triangleright\ \mathcal{Y} = \mathcal{X} \times_n \mathbf{v}$
 1: Pre-process to obtain $M_F$, the number of mode-n fibers of $\mathcal{X}$ and $f_{ptr}$,
    the beginning of each $\mathcal{X}$'s mode-n fiber, size $M_F$.
 2: Pre-allocate $\mathcal{Y}$ space with $M_F$ non-zeros and their indices;
 3: parfor $f = 1, \dots, M_F$ do
 4:     $f_X = f_{ptr}(f)$
 5:     $\text{inds}^1_Y(f) = \text{inds}^1_X(f_X)$
 6:     $\text{inds}^2_Y(f) = \text{inds}^2_X(f_X)$
 7:     $v = \text{val}_Y(f)$
 8:     for $m = f_X, \dots, f_{ptr}(f+1) - 1$ do
 9:         $k = \text{inds}^3_X(m)$
10:         $v += \text{val}_X(m) \times u(k)$
11:
```

The OpenMP-parallelized TTV algorithm using the COO format, **COO-TTV-OMP**, in Algorithm 1 is first illustrated in the PASTA suite [17]. We first pre-process the input tensor $\mathcal{X}$ to record the locations of mode-n fibers. According to the sparse-dense property, the output $\mathcal{Y}$ is pre-allocated with M_F non-zeros and its indices i, j are the same as of $\mathcal{X}$. The storage consists of $16M$ for $\mathcal{X}$, $4I$ for $\mathbf{v}$, and $12M_F$ for $\mathcal{Y}$ [4]. The number of floating-point operations (#Flops) is $2M$. The memory access in Table I counts $4M$ bytes for $\mathbf{v}$ because of its irregular and unpredictable memory access introduced by index-k. The operational intensity is approximately $1/6$ by ignoring less significant terms. **COO-TTM-OMP** is similar to COO-TTV-OMP with the output as a semi-sparse tensor stored in sCOO format. They are both parallelized for independent fibers, but work imbalance may exist because of different fiber lengths of sparse tensor $\mathcal{X}$.

COO-MTTKRP is widely used in Tensor Toolbox [28], Tensorlab [29]; and **COO-MTTKRP-OMP** is implemented the same way as in ParTI library [32]. Each row of $\tilde{\mathbf{A}}$ is updated

[3]For TEW with two input tensors having different non-zero patterns, we support it in the benchmark suite but not analyze them for performance perspective.

[4]$\mathcal{Y}$ is actually a matrix for a third-order TTV.

by scaling the dot product of two rows of matrices $\mathbf{B}$ and $\mathbf{C}$ by the non-zero value of $\mathcal{X}$. Its operational intensity is approximately $1/4$ again by ignoring less expensive terms. COO-MTTKRP-OMP is parallelized by non-zeros, but with atomic operations to protect output matrix. The data race may influence its performance differently depending on the non-zero distribution of an input tensor.

Algorithm 2 COO-TTV-GPU algorithm.

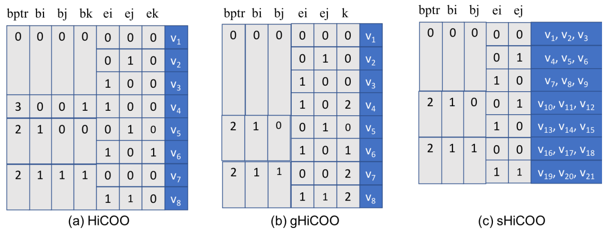
```
Input: A third-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ with $M$ non-zeros in COO
    format, dense vector $\mathbf{V} \in \mathbb{R}^{K}$, and an integer $n$ ($= 3$ in this pseudocode);
Output: Sparse tensor $\mathcal{Y} \in \mathbb{R}^{I \times J}$ in COO format;
                                                        ▷ $\mathcal{Y} = \mathcal{X} \times_n \mathbf{v}$
1: Pre-process to obtain $M_F$, the number of mode-n fibers of $\mathcal{X}$ and $f_{ptr}$,
   the beginning of each $\mathcal{X}$'s mode-n fiber, size $M_F$.
2: Pre-allocate $\mathcal{Y}$ space with $M_F$ non-zeros and their indices;
3: $\text{inds}^1_Y(tid) = \text{inds}^1_X(f_X)$
4: $\text{inds}^2_Y(tid) = \text{inds}^2_X(f_X)$
5: $v = \text{val}_Y(tid)$
6: for $m = f_{ptr}(tid), \dots, f_{ptr}(tid+1) - 1$ do
7:     $k = \text{inds}^3_X(m)$
8:     $v+ = \text{val}_X(m) \times u(k)$
```

2) GPU: **COO-TEW-GPU,COO-TS-GPU**, and **COO-TTV-GPU** use one-dimensional CUDA grids of one-dimensional thread blocks to parallelize non-zeros and fibers respectively. Algorithm 2 illustrates the **COO-TTV-GPU** algorithm, firstly proposed in this work, where M non-zeros are assigned to $M/256$ thread blocks with 256 threads for each. Again, due to the potential unbalanced fiber lengths, COO-TTV-GPU could suffer more performance drop. While **COO-TTM-GPU** and **COO-MTTKRP-GPU** use one-dimensional grids of two-dimensional thread blocks to parallelize the dense matrices, both of them were firstly implemented in ParTI library [32]. In COO-TTM-GPU algorithm, the x-dimension of thread blocks are used to represent matrix columns for GPU memory coalescing, while y-dimension represents non-zeros. (Refer to details in [34].) Be aware that the load imbalance still exists for COO-TTV and COO-TTM, and also data race for COO-MTTKRP, these GPU implementations could not be performance-efficient.

C. Hierarchical Coordinate (HiCOO)

Hierarchical Coordinate (HiCOO) [16] is derived from the COO format but further compresses tensor indices in units of sparse blocks with a pre-specified block size B. It represents tensor indices using two-level *block* and *element* indices, with element indices stored in only 8-bit. An extra block pointer array *bptr* is needed to store the starting locations of every fiber. Figure 2(a) shows HiCOO representation for the same tensor in Figure 1(a) in $2 \times 2 \times 2$ blocks. The same with COO format, HiCOO does not assume any mode order, and only one representation of a sparse tensor is sufficient for all its computations. While HiCOO saves the tensor storage from two aspects: 1) shorter bit-length for element indices; 2) shortened array length for block indices. Readers can refer to more details in the paper [16].

In this work we introduce two variants of the HiCOO format: gHiCOO and sHiCOO. gHiCOO is a generalization of HiCOO format for a general sparse tensor (Figure 2(b)) and sHiCOO is for semi-sparse tensors with dense mode(s) (Figure 2(c)). Concluded by the prior work [16], HiCOO could not be beneficial for hyper-sparse tensors where most tensor blocks only consist of one or few non-zeros. To conquer this problem, we propose gHiCOO where we can pick which modes to be compressed in units of blocks for HiCOO and which stay in COO format. Figure 2(b) chooses to compress modes i and j, leaving mode k in the same index array with Figure 1(a). gHiCOO also provide convenience to implement tensor operations where not all modes are needed during computation, such as TTV and TTM. sHiCOO is similar to sCOO but the extension of the HiCOO format. Figure 2(c) uses sHiCOO to compress the same semi-sparse tensor in Figure 1(b) with dense mode k. Our format variants could be useful in tensor methods and benchmarking for more efficient space and computation.



(a) HiCOO

bptr	bi	bj	bk	ei	ej	ek	
0	0	0	0	0	0	0	v_1
				0	1	0	v_2
				1	0	0	v_3
3	0	0	1	1	0	0	v_4
2	1	0	0	0	1	0	v_5
				1	0	1	v_6
2	1	1	1	0	0	0	v_7
				1	1	0	v_8

(b) gHiCOO

bptr	bi	bj	ei	ej	k	
0	0	0	0	0	0	v_1
			0	1	0	v_2
			1	0	0	v_3
			1	0	2	v_4
2	1	0	0	1	0	v_5
			1	0	1	v_6
2	1	1	0	0	2	v_7
			1	1	2	v_8

(c) sHiCOO

bptr	bi	bj	ei	ej	
0	0	0	0	0	v_1, v_2, v_3
			0	1	v_4, v_5, v_6
			1	0	v_7, v_8, v_9
2	1	0	0	1	v_{10}, v_{11}, v_{12}
			1	0	v_{13}, v_{14}, v_{15}
2	1	1	0	0	v_{16}, v_{17}, v_{18}
			1	1	v_{19}, v_{20}, v_{21}

Fig. 2. HiCOO, gHiCOO formats for general sparse tensors and sHiCOO for semi-sparse tensors.

D. Implementations based on HiCOO

HiCOO parallel implementations are all firstly proposed in this benchmark suite except MTTKRP on CPUs [16], [17]. Advanced techniques such as privatization, lock-avoiding parallel strategies, advanced scheduling [16] are not adopted since the purpose of this suite is to act as reference implementations for the community and also to avoid complicated tuning parameters.

1) Multicore CPU: Since the pre-processing phase has dealt with allocating space and setting indices for the output tensor in HiCOO format, the floating-point value computation of HiCOO-TEW-OMP and HiCOO-TS-OMP is the same with COO-TEW-OMP and COO-TS-OMP, respectively.

HiCOO-TTV-OMP and **HiCOO-TTM-OMP** also pre-allocate the output tensors according to the sparse-dense property. We use gHiCOO format to represent the input tensor $\mathcal{X}$ by leaving the mode doing the product uncompressed. Therefore, TTV and TTM can bypass the blocking nature of HiCOO and be performed without data race between blocks. Then the same computation will be implemented for HiCOO-TTV-OMP and HiCOO-TTM-OMP as in their COO counterparts, with pre-allocated output tensors and their indices in HiCOO or sHiCOO format respectively.

HiCOO-MTTKRP-OMP is more complicated because indices of all the tensor modes are used in this computation, different from TTV and TTM where the needed indices are all

Algorithm 3 HiCOO-MTTKRP-OMP algorithm in mode-1 [16].

Input: A sparse tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ with M non-zeros in HiCOO format, dense matrices $\mathbf{B} \in \mathbb{R}^{J \times R}$, $\mathbf{C} \in \mathbb{R}^{K \times R}$, block size B;
Output: Updated dense matrix $\tilde{\mathbf{A}} \in \mathbb{R}^{I \times R}$;

```
                                                    ▷ $\tilde{\mathbf{A}} \leftarrow \mathcal{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$
1: parfor $b = 1, \ldots, n_b$ do
2:     $bi = \text{binds}^1(b)$, $bj = \text{binds}^2(b)$, $bk = \text{binds}^3(b)$;
3:     $\mathbf{A}_b = \mathbf{A} + bi \cdot B \cdot R$; $\mathbf{B}_b = \mathbf{B} + bj \cdot B \cdot R$; $\mathbf{C}_b = \mathbf{C} + bk \cdot B \cdot R$;
4:     for $x = bptr(b), \ldots, bptr(b+1) - 1$ do                ▷ entry $x$
5:         $ei = \text{einds}^1(x)$, $ej = \text{einds}^2(x)$, $ek = \text{einds}^3(x)$
6:         $value = \text{val}(x)$
7:         for $r = 1, \ldots, R$ do
8:             $\tilde{A}_b(ei, r) += value \cdot C_b(ek, r) \cdot B_b(ej, r)$
9:
```

from only one mode. We first block matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ as $\mathbf{A}_b$, $\mathbf{B}_b$, and $\mathbf{C}_b$ to be reused for the non-zeros inside a tensor block. For each block, we update the values of a block of the output matrix $\tilde{\mathbf{A}}$ using corresponding blocks $\mathbf{B}_b$, and $\mathbf{C}_b$. Thus, we do not need to compute the actual indices i, j, k, and data locality is enhanced due to blocking and Morton order sorting implied by the HiCOO format [16]. Different from COO-MTTKRP-OMP, HiCOO-MTTKRP-OMP parallelizes in the units of tensor blocks rather than non-zeros.

The analysis of HiCOO algorithms are also listed in Table I. Since HiCOO-TEW, -TS, -TTV, -TTM all have the same value computation step with COO counterparts, so they have the same behavior except for the storage space, where HiCOO is usually beneficial. However, from our experiments, we still observe some benefits of HiCOO affected by the pre-processing stage. HiCOO-MTTKRP has smaller memory access than COO-MTTKRP due to its blocked feature.

2) GPU: HiCOO-GPU implementations are also the same with COO ones except HiCOO-MTTKRP-GPU. This unoptimized HiCOO-MTTKRP-GPU maps one tensor block to a CUDA thread block, thus the balanced workload by non-zero distribution for COO-MTTKRP disappears, while the atomic operation stays. Therefore, the work imbalance due to different numbers of non-zeros in tensor blocks could make its performance even worse than COO-MTTKRP-GPU.

IV. Tensor Dataset

This benchmark suite uses sparse tensors derived from real-world applications from online collections [35], [36], [37]. It also generates synthetic sparse tensors based on graph models that preserve the properties of real-world graphs. The benchmark suite can be run against any set of tensors provided that they are expressed using coordinate format.

A. Tensors From Real World Applications

The tensors taken from real-world applications are described in Table II(a), sorted by tensor order and decreasing density. They are taken from The Formidable Repository of Open Sparse Tensors and Tools (FROSTT) dataset [35], the HaTen2 [38] dataset, and one built from electronic medical records from Children's Healthcare of Atlanta [37]. These tensors were chosen to represent a wide range of domains: pattern recognition (*vast*, *nips4d*, *uber4d*), natural language processing (*nell2*, *nell1*), healthcare analytics (*choa*), social network analysis (*deli*, *deli4d*, *flickr*, *flickr4d*, *fb-m*, *fb-s*), crime detection (*crime4d*), and anomaly detection (*enron4d*, *darpa*).

B. Synthetic Tensor Generation

The Kronecker graph model [39] and the biased power law generator from the FireHose streaming benchmark [40] are extended to generate third- and fourth-order tensors. These methods were selected because the resultant graphs follow the power law distribution, exhibit a small diameter, and have a high average clustering coefficient.

The software to generate synthetic tensors is included in the benchmark suite and the synthetic tensors used in our experiments are described in Table II(b) in a period order of "small, medium, large". The regular tensors, which are equidimensional along each mode, are generated using the Kronecker generator. The irregular tensors are generated using the biased power law streaming generator. The third- and fourth-order irregular tensors have one or two modes completely dense and much smaller compared to the two other modes which are equidimensional and sparse.

1) The Stochastic Kronecker Graph Model: The Stochastic Kronecker graph model [39] is a fractal growth model that preserves the properties of real-world graphs. We extended this approach to generate sparse tensors of order N by accepting the initiator as a small tensor $\mathcal{X}_1$ with N modes. By taking the repeated Kronecker products of $\mathcal{X}_1$ for n times, a larger Nth-order tensor $\mathcal{X}_n$ is produced. With Bernoulli sampling, $\mathcal{X}_n$ can be considered as a large sparse tensor representing the resultant hypergraph that follows the properties of real-world networks. The exponential growth of Kronecker multiplication limits the sizes of generated Nth-order tensors to certain numbers. We overcome this by performing an additional iteration of Kronecker multiplication and strip off the tensor coordinates those fall outside the given dimension sizes to obtain the final sparse tensor $\mathcal{X}$.

2) The Power Law Generator: The generator produces a stream of edges those, when combined, form a graph respecting the power law distribution. Rooted from a graph, a.k.a. a sparse matrix, the generator combines graphs together to form slices of a hypergraph, a.k.a. a third order tensor. This process, when repeated on an $(N-1)$th-order tensor, will generate a sparse tensor with N modes. The power law generated graphs do not possess a high average clustering coefficient, thus tensors in arbitrary sizes can be directly generated.

V. Experimental Results

We test our tensor kernels on four parallel platforms including Intel CPUs and NVIDIA GPUs and build Roofline performance models to measure our performance bounds for detailed analysis. We use floating point operations per second (FLOPS) to compare among kernels and platforms.

A. Configurations

1) Platforms: We run the experiments on four parallel platforms, the parameters of which are listed in Table III. All

TABLE II
DESCRIPTION OF SPARSE REAL AND SYNTHETIC TENSORS GENERATED (GEN.) WITH KRONECKER AND POWER LAW GENERATORS INDICATED AS KRON. AND PL RESPECTIVELY.

No.	Tensors	Order	Dimensions	#Nnzs	Density
r1	*vast*	3	$165K \times 11K \times 2$	26M	6.9E-3
r2	*nell2*	3	$12K \times 9K \times 29K$	77M	2.4E-5
r3	*choa*	3	$712K \times 10K \times 767$	27M	5.0E-6
r4	*darpa*	3	$22K \times 22K \times 24M$	28M	2.4E-9
r5	*fb-m*	3	$23M \times 23M \times 166$	100M	1.1E-9
r6	*fb-s*	3	$39M \times 39M \times 532$	140M	1.7E-10
r7	*flickr*	3	$320K \times 28M \times 1.6M$	113M	7.8E-12
r8	*deli*	3	$533K \times 17M \times 2.5M$	140M	6.1E-12
r9	*nell1*	3	$2.9M \times 2.1M \times 25M$	144M	9.1E-13
r10	*crime4d*	4	$6K \times 24 \times 77 \times 32$	5M	1.5E-2
r11	*uber4d*	4	$183 \times 24 \times 1140 \times 1717$	3M	3.9E-4
r12	*nips4d*	4	$2K \times 3K \times 14K \times 17$	3M	1.8E-6
r13	*enron4d*	4	$6K \times 6K \times 244K \times 1K$	54M	5.5E-9
r14	*flickr4d*	4	$320K \times 28M \times 1.6M \times 731$	113M	1.1E-14
r15	*deli4d*	4	$533K \times 17M \times 2.5M \times 1K$	140M	4.3E-15

(a) real tensors

No.	Tensors	Gen.	Order	Dimensions	#Nnzs	Density
s1	*regS*	Kron.	3	$(65K)^3$	1.1M	3.72E-9
s2	*regM*	Kron.	3	$(1.1M)^3$	11.5M	9.97E-12
s3	*regL*	Kron.	3	$(8.3M)^3$	94M	1.61E-13
s4	*irrS*	PL	3	$(32K)^2 \times 76$	1M	1.26E-5
s5	*irrM*	PL	3	$(524K)^2 \times 126$	10M	1.43E-6
s6	*irrL*	PL	3	$(4.2M)^2 \times 168$	84M	2.86E-8
s7	*regS4d*	Kron.	4	$(8.2K)^4$	1M	2.26E-10
s8	*regM4d*	Kron.	4	$(2.1M)^4$	11.2M	5.83E-19
s9	*regL4d*	Kron.	4	$(8.3M)^4$	110M	2.23E-20
s10	*irrS4d*	PL	4	$(1.6M)^3 \times 82$	1.0M	2.90E-9
s11	*irrM4d*	PL	4	$(2.6M)^3 \times 144$	10.8M	4.17E-12
s12	*irrL4d*	PL	4	$(4.2M)^3 \times 226$	100M	6.0E-15
s13	*irr2S4d*	PL	4	$(1.0M)^2 \times 122 \times 436$	1.6M	2.81E-11
s14	*irr2M4d*	PL	4	$(4.2M)^2 \times 232 \times 746$	19.9M	6.53E-12
s15	*irr2L4d*	PL	4	$(8.3M)^2 \times 952 \times 324$	109M	5.1E-12

(b) synthetic tensors

TABLE III
PLATFORM PARAMETERS.

Parameters	Intel CPUs		NVIDIA GPUs	
	Bluesky	Wingtip	DGX-1P	DGX-1V
Processor	Intel Xeon Gold 6126	Intel Xeon E7-4850v3	NVIDIA Tesla P100	NVIDIA Tesla V100
Microarch	Skylake	Haswell	Pascal	Volta
Frequency	2.60 GHz	2.20 GHz	1.48 GHz	1.53 GHz
#Cores	24 (12 × 2)	56 (14 × 4)	3584	5120
Peak SP Perf.	1.0 TFLOPS	2.0 TFLOPS	10.6 TFLOPS	14.9 TFLOPS
LLC size	19 MB	35 MB	3 MB	6 MB
Mem. size	196 GB	2114 GB	16 GB	16 GB
Mem. type	DDR4	DDR4	HBM2	HBM2
Mem. freq.	2.666 GHz	2.133 GHz	0.715 GHz	0.877 GHz
Mem. BW	256 GB/s	273 GB/s	732 GB/s	900 GB/s
Compiler	gcc 7.1.0	gcc 5.5.0	CUDA Tkit 9.1	CUDA Tkit 9.0

Intel platforms are non-uniform memory access (NUMA) machines with 2-4 NUMA nodes. We calculate the peak single-precision (SP) floating point performance and main/global-memory bandwidth from these parameters. The peak SP performance of all machines is above 1 TFLOPS. GPUs show advantages in peak performance and memory bandwidth over CPUs by approximately $4-12\times$ and $3-7\times$ respectively.

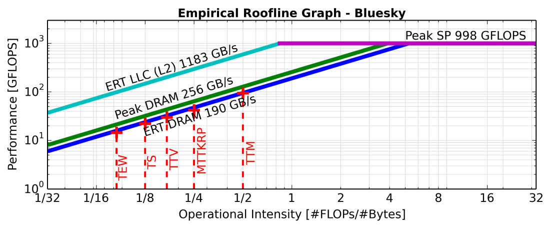
2) Kernel Implementation Details: Since our data is naturally sorted in a particular mode order, COO implementations could take some advantages from the better data locality. TEW and TS take addition and multiplication operations as representatives respectively; the performance using different operations is quite similar in our experiments. For multicore CPU implementations, we use OpenMP for parallelization under different scheduling strategies, with the number of threads set to the number of physical cores. "omp atomic" is used to handle data race in MTTKRP, and "omp simd" is for vectorization of TTM and MTTKRP. We use "numactl –interleave=all –physcpubind" to interleave memory allocation for better memory bandwidth usage and thread binding for lower scheduling overhead. For GPU implementations, "atomicAdd" is used in MTTKRP. For the HiCOO format, we fix the block size to 128 to fit into the last-level cache in all platforms and use only 8 bits to store element indices. We use 16 as the column size for matrices in TTM and MTTKRP, to reflect the low-rank feature in popular tensor methods. We run all kernels five times to get the average runtime; the time of TTV, TTM, and MTTKRP is further averaged among all tensor modes.

B. Roofline Performance Models

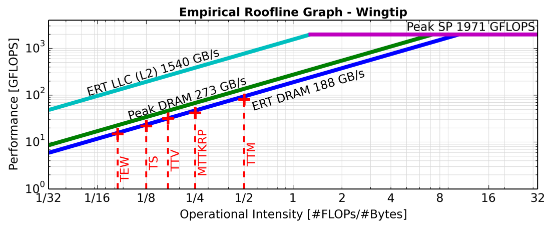
The Roofline performance model [41], [42] is a graphical representation of machine characteristics. It is employed for performance analysis in various application domains: digital signal processing, e.g., Spiral [43], sparse/dense linear algebra [42], [44], and Lattice Boltzmann Magnetohydrodynamics (LBMHD) [42]. The Empirical Roofline Tool (ERT) [45], included into Intel Advisor tool, automates the measurement of the target machine characteristics. ERT automatically generates Roofline data including the maximum bandwidth for various levels of memory hierarchy, obtained by testing a variety of micro-kernels [5]. ERT can utilize OpenMP and CUDA for parallelization on a single machine; we configure it to the corresponding compiler in Table III for tests.

Figure 3 plots the Roofline models for the four platforms in Table III with DRAM and last-level cache (LLC) bandwidth tested from ERT as obtainable bandwidth, and the theoretical peak SP performance and DRAM bandwidth (not cache-aware) for reference. We mark the operational intensities (#Flops/#Bytes, OIs) of our tensor kernels calculated from Table I overlying with Roofline models. "ERT-DRAM" bandwidth is the obtainable bandwidth from benchmarking micro-kernels, thus OIs of all the tensor kernels are marked on this line. From this figure, all the sparse tensor kernels we consider are main or global memory bound for CPUs and GPUs. Higher bandwidth will accelerate kernel execution, while other factors such as better data reuse (cache utilization) lowering memory access pressure will also lead to performance improvement by increasing the OI value. We use the calculated performance

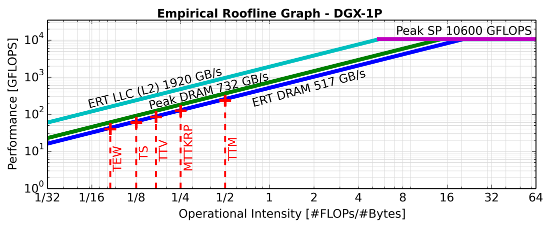
[5]The micro-kernels are similar to those in STREAM benchmark suite [46].



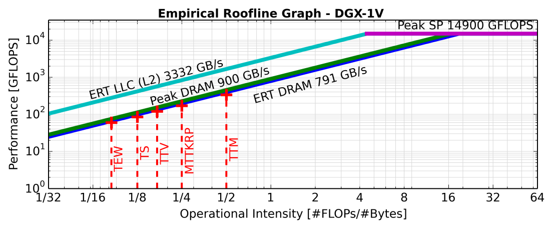
(a) Bluesky



(b) Wingtip



(c) DGX-1P



(d) DGX-1V

Fig. 3. Roofline models marked with the operational intensities of tensor kernels.

of all tensor kernels as the upper bounds in our performance figures below (called "Roofline performance"), calculated by timing the OI value with the "ERT-DRAM" bandwidth. The OI value is an accurate #Flops/#Bytes ratio by taking actual tensor features into account, especially for TTV and TTM because of the M_F term in Table I.

C. Performance

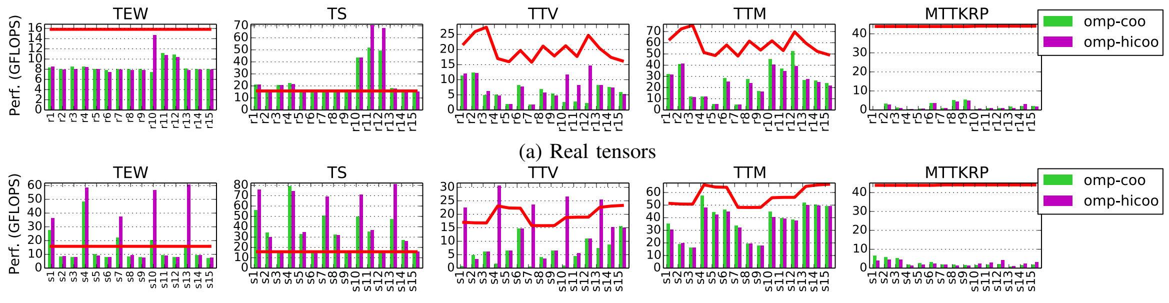
We present the performance of all the five tensor kernels using two datasets, real and synthetic, on four platforms along with five observations. The performance in GFLOPS of each tensor kernel is calculated from #Flops (in Table I) divided by the measured execution time. X-axis represents tensors using the numbers in Table II from different datasets.

Observation 1: *Achieved performance is diverse and difficult to predict, varying with the dimension sizes and non-zero patterns of tensors, data formats, and platforms.*

From Figures 4 to 7, the achieved performance in GFLOPS is extremely diverse among tensor kernels, data formats, platforms, and datasets. Take the synthetic dataset as an example, the performance varies from 0.8 GFLOPS (*regS* in COO) to 81 GLOPS (*irr2S4d* in HiCOO) in Figure 4 on the Bluesky CPU platform. Besides, the average performance of the five kernels ranges a lot as well. TEW, TS, TTV, TTM, MTTKRP kernels achieve an average of 14.6, 35.1, 6.3, 37.7, and 2.7 GFLOPS respectively for COO format, and 22.3, 40.8, 14.4, 35.8, and 2.6 GFLOPS respectively for HiCOO format. This also shows HiCOO on average behaves better than COO format for TEW, TS, and TTV and gets similar performance on TTM and MTTKRP. Even for the performance efficiency (or bandwidth efficiency), these kernels still vary largely from the lowest 2% (MTTKRP on *irr2S4d*) to 353% (TS on *regS*) for COO and 2% (MTTKRP on *irr2S4d*) - 479% (TS on *regS*) for HiCOO. (The over 100% efficiency will be explained below.) Also, different performance numbers are observed between real and synthetic datasets for the same tensor kernel. From platform perspective, TTV shows much lower GFLOPS numbers on Wingtip platform in Figure 5 than those on Bluesky in Figure 4. Though we observe some trend especially for synthetic dataset and TTM operation, generally it is hard to predict the performance of a sparse kernel, even operating with a dense matrix or vector.

Observation 2: *Performance is generally below the Roofline performance calculated from main/global memory bandwidth except for some small tensors fitting into caches or algorithms with good data locality thus making a good use of caches.*

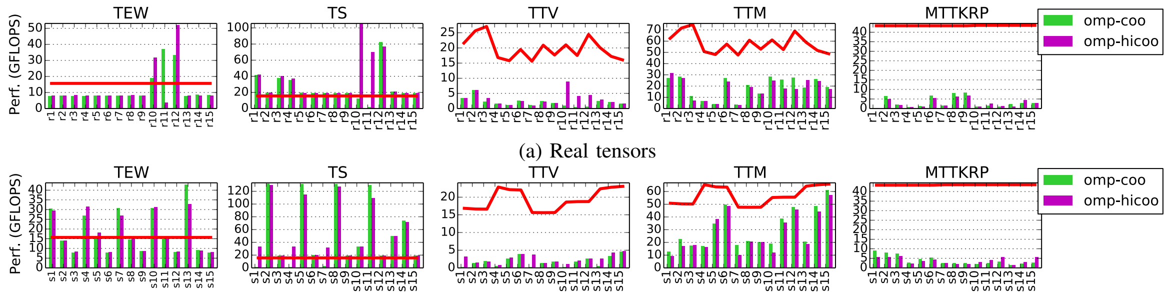
Most of cases in Figures 4 to 7 fall below the red "Roofline performance" line calculated using main/global memory bandwidth from Figure 3 except for some cases of TEW and TS on Bluesky and Wingtip CPU platforms and MTTKRP on DGX-1V GPU platform. Take Figure 4 as an example again, the tensors exceed Roofline performance are all small tensors with around 1M non-zeros: *regS*, *irrS*, *regS4d*, *irrS4d*, and *irr2S4d* for TEW, all small and medium synthetic tensors and small real tensors with 3-5M non-zeros: *crime4d*, *uber4d*, and *nips4d*. The last level cache size of Bluesky is 19 MB which could reside the value arrays of three tensors, each with around 1.6M non-zeros, for TEW, or of two tensors, each with around 2.4M non-zeros, for TS. These numbers match the sizes of the small synthetic and real tensors, while the medium tensors also gain some cache benefit. For MTTKRP on DGX-1V, COO-MTTKRP-GPU gets higher performance than Roofline more on irregular-shaped tensors in the synthetic dataset. And the tensors achieve high performance on DGX-1P are easier to break the upper bound on DGX-1V. One reason is that V100 GPU architecture has a twice larger LLC (6M) than P100; besides, V100 get improved atomic operation performance which could benefit MTTKRP; another reason is that they might have very good data reuse or small working-set size, e.g., tensors with a very short mode, so it is cache that offers the data injection rate rather than off-chip memory; lastly, the integer and floating-point operations have independent datapaths for instruction issuing on Volta architecture. Therefore,



(a) Real tensors

(b) Synthetic tensors

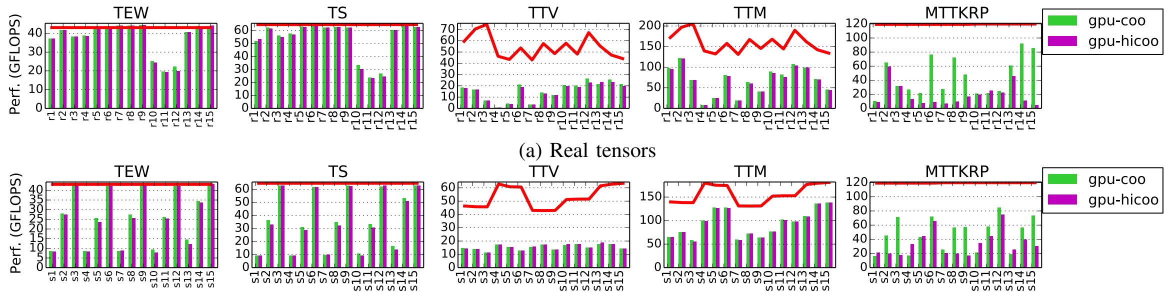
Fig. 4. Single-precision performance of the five tensor kernels on Bluesky with 24 CPU cores. Red line shows the calculated "Roofline performance".



(a) Real tensors

(b) Synthetic tensors

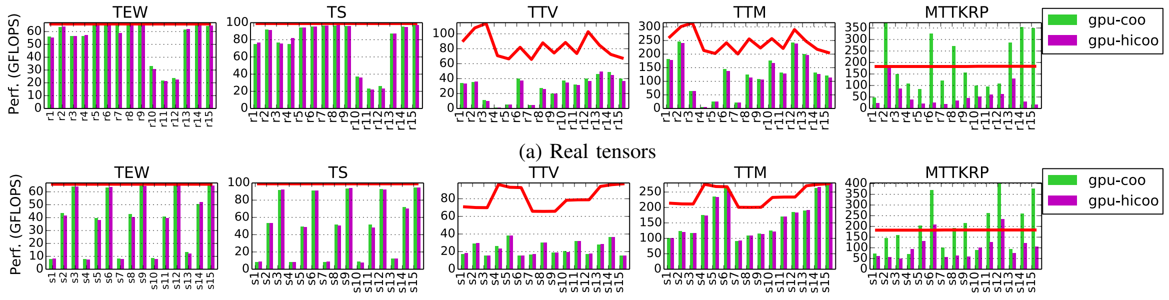
Fig. 5. Single-precision performance of the five tensor kernels on Wingtip with 56 CPU cores. Red line shows the calculated "Roofline performance".



(a) Real tensors

(b) Synthetic tensors

Fig. 6. Single-precision performance of the five tensor kernels on NVIDIA DGX-1P. Red line shows the calculated "Roofline performance".



(a) Real tensors

(b) Synthetic tensors

Fig. 7. Single-precision performance of the five tensor kernels on NVIDIA DGX-1V. Red line shows the calculated "Roofline performance".

address computation which is extensively used in MTTKRP can be overlapped with floating-point operations, which may mitigate the waiting time for address calculation compared with earlier GPU architectures.

Observation 3: *It is hard to obtain good performance efficiency for non-streaming kernels on multi-socket CPU machines because of NUMA effect, which could be even harder than on GPUs.*

We consider TTV, TTM, and MTTKRP are non-streaming kernels, versus the streaming kernels TEW and TS from the computation pattern. On Bluesky (Figure 4), the average performance efficiency of TTV, TTM, and MTTKRP is 31%, 64%, 6% for COO, and 73%, 61%, 5% for HiCOO; while the numbers are 9%, 52%, 9% for COO and 13%, 47%, 9% for HiCOO on Wingtip (Figure 5). Though MTTKRP behaves a little higher efficiency on Wingtip, its efficiency is still very low. The increment could come from better parallelism of Wingtip with 56 cores. On DGX-1P GPU (Figure 6), the average performance efficiency of TTV, TTM, and MTTKRP is 30%, 60%, 40% for COO, and 30%, 60%, 28% for HiCOO; while the numbers are 30%, 69%, 110% (higher than 100% memory efficiency is due to the reasons we mentioned) for COO and 30%, 69%, 57% for HiCOO on DGX-1V (Figure 7). The average efficiency numbers on the four-socket Wingtip CPU are all lower than those on DGX-1P and DGX-1V GPUs and two-socket Bluesky CPU for TTV and TTM.

Observation 4: *HiCOO algorithms is faster than or similar to COO counterparts because of its better local locality and smaller memory footprint, except* MTTKRP *on GPUs where load imbalance and lower parallelism play more important roles.*

From the average efficiency and performance numbers shown in observations 1 and 3, HiCOO on average behaves better than COO for TEW, TS, and TTV and gets similar performance on TTM and MTTKRP on CPU platforms. On the two GPUs, due to their smaller last-level cache size, HiCOO does not benefit as much as on CPUs. From Figures 6 and 7, HiCOO obtains very similar performance on TEW, TS, TTV, and TTM because of their similar execution code for tensor value computation. While HiCOO-MTTKRP behaves worse than COO-MTTKRP because of their different parallel strategies. HiCOO parallelizes tensor blocks with severe load imbalance issue and lower parallelism compared to COO-MTTKRP algorithm. Thus, to take advantage of the HiCOO format, a careful tuning need to be done according to architecture features.

Observation 5: *Different datasets expose very different performance behavior, which shows the importance of synthetic datasets to performance benchmarking and analysis.*

Consider the performance trend of real and synthetic datasets. TEW and TS show obvious period trend from high to low or low to high on CPUs and GPUs respectively due to different cache sizes on the synthetic dataset, while it is hard to find trends in the real dataset. TTV and especially TTM show a matching trend with the Roofline predicted performance for both real and synthetic tensors. Since real tensors are from diverse real application scenarios, it is hard to do benchmarking and performance analysis solely based on them. Moreover, real tensors are limited due to data privacy and other publicity issues. Performance numbers are observed in a similar scale for large tensors from real and synthetic datasets. Thus, the generated synthetic tensors are good candidates to reveal the performance of tensor kernels. Extracting features from real tensors as a basis to create more complete synthetic tensors would be very helpful for sparse tensor research.

By running the benchmark suite to obtain kernel performance combining with the Roofline analysis, users can learn whether advanced performance optimization needs to be further explored to deal with the issues such as data locality, load imbalance, and lock efficiency. Besides, comparing among these sparse tensor kernels, choosing a good kernel for an application is possible by comparing their actual performance numbers using the same tensor from this application.

VI. Related Work

Work related to this benchmark suite includes various benchmarking collections and synthetic benchmark data generation. This benchmark suite is distinct from previous efforts in its focus on multi-dimensional sparse tensors combined with both real and synthetic inputs, and the first sparse tensor benchmarking effort on GPUs.

Benchmark suites measure machine attributes and exemplify computing patterns. Benchmarks that measure specific machine attributes include LINPACK [47], SPEC [48], STREAM [46], GeekBench [49], Multimaps [50], Bandwidth [51], and others. Most of them aim to measure memory bandwidth achieved under varying conditions and a few target architecture floating-point capabilities. Benchmark suites are often organized around the concept of application exemplars. These suites emulate common patterns and behaviors in application classes of interest. Several examples of these suites have been published: LAPACK/ScaLAPACK for dense linear algebra [52], Colella's Seven Motif's [53] for scientific computing, PARSEC [54] and SPLASH2 [55], Rodinia [56], Graph500 [57], SparseBench [58], GAP [59], SSCA#2 [60], and Tartan [61], to name a few.

Several approaches to synthetic graph generation have been proposed. Our work extends two of these, power law graphs from Firehose, and Kronecker graphs from Graph500 [57]. FireHose is a suite of stream processing benchmarks [40], one of a front-end generator of which is the biased power law generator. Existing synthetic tensor generators like SimTensor [62], Nway Toolbox [63], and the Tensor Toolbox [28] are specific to tensors with Tucker [64], CANDECOMP/PARAFAC decomposition [65], [66] structures or particular data distributions. This paper provides a starting point to generate sparse tensors that preserves the properties of real-world or multi-attributed graphs that can be realized as higher-order sparse tensors.

Many libraries support sparse tensor methods, such as Tensor Toolbox [28], Nway Toolbox [63], Tensorlab [29], TACO [31], SPLATT [23], and ParTI [32]. As a benchmark suite, we supply widely-adopted reference implementations and will make continuously effort to include state-of-the-art algorithms and data structures as well.

VII. Conclusion

This paper presents a benchmark suite targeting popular sparse tensor kernels. Operations on sparse tensors are common in a wide range of important applications. The operations are memory bound and often dominate application performance. This benchmark suite identifies important kernels and data representations and provides reference implementations to aid the community in effectively sharing and comparing performance and optimization results. Two methods for synthetic tensor generation are provided by preserving the properties of real-world graphs. A subset of possible synthetic tensors are used in this paper. The tool provides the ability to generate custom synthetic tensors in a reproducible manner.

Five observations are made based on performance analysis over Roofline models to gain insights of sparse tensor behavior across architectures. This benchmark suite is a continuous effort: additional operations, such as TTM-chain in Tucker decomposition, tensor contraction, a sparse tensor with a sparse vector/matrix products; more complete tensor methods, such as CANDECOMP/PARAFAC and Tucker decompositions; data representations, such as compressed sparse fiber (CSF) [23], balanced and mixed-mode CSF (BCSF, MMCSF) [24], [25]; more platforms, such as distributed systems, multiple GPUs, and other new architectures (e.g., FPGAs and Emu [67], [68]) will be considered adding to the suite in the future.

Acknowledgment

This research is partially funded by the US Department of Energy, Office for Advanced Scientific Computing (ASCR) under Award No. 66150: "CENATE: The Center for Advanced Technology Evaluation" and the Laboratory Directed Research and Development program at PNNL under contract No. ND8577. Pacific Northwest National Laboratory (PNNL) is a multiprogram national laboratory operated for DOE by Battelle Memorial Institute under Contract DE-AC05-76RL01830.

References

[1] T. Kolda and B. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009.

[2] A. Cichocki, "Era of big data processing: A new approach via tensor networks and tensor decompositions," *CoRR*, vol. abs/1403.2048, 2014.

[3] J. Li, "Scalable tensor decompositions in high performance computing environments," Ph.D. dissertation, Georgia Institute of Technology, Atlanta, GA, USA, July 2018.

[4] L. De Lathauwer, N. Vervliet, M. Boussé, and O. Debals, "Dealing with curse and blessing of dimensionality through tensor decompositions," 2017.

[5] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, "Speeding-up convolutional neural networks using fine-tuned CP-decomposition," *arXiv preprint arXiv:1412.6553*, 2014.

[6] A. Novikov, D. Podoprikhin, A. Osokin, and D. Vetrov, "Tensorizing neural networks," *CoRR*, vol. abs/1509.06569, 2015.

[7] J. Li, J. Choi, I. Perros, J. Sun, and R. Vuduc, "Model-driven sparse CP decomposition for higher-order tensors," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 1048–1057.

[8] O. Kaya and B. Uçar, "Parallel Candecomp/Parafac decomposition of sparse tensors using dimension trees," *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. C99–C130, 2018. [Online]. Available: https://doi.org/10.1137/16M1102744

[9] J. Li, C. Battaglino, I. Perros, J. Sun, and R. Vuduc, "An input-adaptive and in-place approach to dense tensor-times-matrix multiply," in *ACM/IEEE Supercomputing (SC '15)*. New York, NY, USA: ACM, 2015.

[10] E. Di Napoli, D. Fabregat-Traver, G. Quintana-Ortí, and P. Bientinesi, "Towards an efficient use of the BLAS library for multilinear tensor contractions," *Applied Mathematics and Computation*, vol. 235, pp. 454 – 468, 2014.

[11] D. Matthews, "High-performance tensor contraction without BLAS," *CoRR*, vol. abs/1607.00291, 2016. [Online]. Available: http://arxiv.org/abs/1607.00291

[12] 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. [Online]. Available: http://doi.acm.org/10.1145/3091966.3091968

[13] J. Li, Y. Ma, C. Yan, and R. Vuduc, "Optimizing sparse tensor times matrix on multi-core and many-core architectures," in *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA^3 '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 26–33. [Online]. Available: https://doi.org/10.1109/IA3.2016.10

[14] J. Li, G. Tan, M. Chen, and N. Sun, "SMAT: An input adaptive auto-tuner for sparse matrix-vector multiplication," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 117–126. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462181

[15] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on GPUs," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 99–108. [Online]. Available: http://doi.acm.org/10.1145/2751205.2751244

[16] J. Li, J. Sun, and R. Vuduc, "HiCOO: Hierarchical storage of sparse tensors," in *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, Dallas, TX, USA, November 2018.

[17] J. Li, Y. Ma, X. Wu, A. Li, and K. Barker, "Pasta: A parallel sparse tensor algorithm benchmark suite," *arXiv preprint arXiv:1902.03317*, 2019.

[18] S. Ji, W. Xu, M. Yang, and K. Yu, "3D convolutional neural networks for human action recognition," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 1, pp. 221–231, 2012.

[19] A. Anandkumar, R. Ge, D. Hsu, S. M. Kakade, and M. Telgarsky, "Tensor decompositions for learning latent variable models," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 2773–2832, Jan. 2014.

[20] R. Yu, S. Zheng, A. Anandkumar, and Y. Yue, "Long-term forecasting using tensor-train RNNs," 2018. [Online]. Available: https://openreview.net/forum?id=HJJ0w–0W

[21] A. Cichocki, N. Lee, I. V. Oseledets, A. Phan, Q. Zhao, and D. Mandic, "Low-rank tensor networks for dimensionality reduction and large-scale optimization problems: Perspectives and challenges part 1," *ArXiv e-prints*, Sep. 2016.

[22] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos, "Tensor decomposition for signal processing and machine learning," *IEEE Transactions on Signal Processing*, vol. 65, no. 13, pp. 3551–3582, July 2017.

[23] S. Smith, N. Ravindran, N. Sidiropoulos, and G. Karypis, "SPLATT: Efficient and parallel sparse tensor-matrix multiplication," in *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS, 2015.

[24] A. S.-R. R. V. P. S. Israt Nisa, Jiajia Li, "Load-balanced sparse MTTKRP on GPUs," (To be appeared), 2019.

[25] I. Nisa, J. Li, A. Sukumaran-Rajam, P. S. Rawat, S. Krishnamoorthy, and P. Sadayappan, "An efficient mixed-mode representation of sparse tensors," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: ACM, 2019, pp. 49:1–49:25. [Online]. Available: http://doi.acm.org/10.1145/3295500.3356216

[26] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi, "A unified optimization approach for sparse tensor operations on GPUs," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2017, pp. 47–57.

[27] M. Baskaran, B. Meister, N. Vasilache, and R. Lethin, "Efficient and scalable computations with sparse tensors," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, Sept 2012, pp. 1–6.

[28] B. W. Bader, T. G. Kolda *et al.*, "MATLAB Tensor Toolbox (Version 3.0-dev)," Available online, Oct. 2017. [Online]. Available: https://www.tensortoolbox.org

[29] N. Vervliet, O. Debals, L. Sorber, M. Van Barel, and L. De Lathauwer, "Tensorlab (Version 3.0)," Available from http://www.tensorlab.net, March 2016.

[30] S. Chou, F. Kjolstad, and S. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 123:1–123:30, Oct. 2018. [Online]. Available: http://doi.acm.org/10.1145/3276493

[31] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 77:1–77:29, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133901

[32] J. Li, Y. Ma, and R. Vuduc, "ParTI! : A parallel tensor infrastructure for multicore CPUs and GPUs (Version 1.0.0)," Oct 2018. [Online]. Available: https://github.com/hpcgarage/ParTI

[33] J. Li, B. Uçar, U. V. Çatalyürek, J. Sun, K. Barker, and R. Vuduc, "Efficient and effective sparse tensor reordering," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: ACM, 2019, pp. 227–237. [Online]. Available: http://doi.acm.org/10.1145/3330345.3330366

[34] Y. Ma, J. Li, X. Wu, C. Yan, J. Sun, and R. Vuduc, "Optimizing sparse tensor times matrix on GPUs," *Journal of Parallel and Distributed Computing*, 2018.

[35] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis, "FROSTT: The Formidable Repository of Open Sparse Tensors and Tools," 2017. [Online]. Available: http://frostt.io/

[36] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos, "HaTen2: Billion-scale tensor decompositions," in *IEEE International Conference on Data Engineering (ICDE)*, 2015.

[37] I. Perros, E. E. Papalexakis, F. Wang, R. Vuduc, E. Searles, M. Thompson, and J. Sun, "SPARTan: Scalable PARAFAC2 for large & sparse data," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17. New York, NY, USA: ACM, 2017, pp. 375–384.

[38] I. Jeon, E. E. Papalexakis, and C. F. U Kang, "HaTen2: Billion-scale tensor decompositions (Version 1.0)," Available from http://datalab.snu.ac.kr/haten2/, 2015.

[39] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *Journal of Machine Learning Research*, vol. 11, no. Feb, pp. 985–1042, 2010.

[40] K. Anderson and S. Plimpton, "Firehose streaming benchmarks," Sandia National Laboratory, Tech. Rep., 2015.

[41] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

[42] S. W. Williams, "Auto-tuning performance on multicore computers," Ph.D. dissertation, EECS Department, University of California, Berkeley, Dec 2008. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-164.html

[43] G. Ofenbeck, R. Steinmann, V. C. Cabezas, D. G. Spampinato, and M. Püschel, "Applying the roofline model," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 76 – 85.

[44] X. Zhang, G. Tan, S. Xue, J. Li, K. Zhou, and M. Chen, "Understanding the gpu microarchitecture to achieve bare-metal performance tuning," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '17. New York, NY, USA: ACM, 2017, pp. 31–43. [Online]. Available: http://doi.acm.org/10.1145/3018743.3018755

[45] Y. J. Lo, S. Williams, B. Van Straalen, T. J. Ligocki, M. J. Cordery, N. J. Wright, M. W. Hall, and L. Oliker, "Roofline Model Toolkit: A practical tool for architectural and program analysis," in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, S. A. Jarvis, S. A. Wright, and S. D. Hammond, Eds. Cham: Springer International Publishing, 2015, pp. 129–148.

[46] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," 1991-2007, a continually updated technical report. http://www.cs.virginia.edu/stream/. [Online]. Available: http://www.cs.virginia.edu/stream/

[47] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: past, present and future," *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003.

[48] K. M. Dixit, "The SPEC benchmarks," *Parallel computing*, vol. 17, no. 10-11, pp. 1195–1209, 1991.

[49] GeekBench, "Primate labs," 4. [Online]. Available: http://http://www.geekbench.com/

[50] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha, "A framework for performance modeling and prediction," in *Supercomputing 2002*. IEEE, 2002, pp. 21–21.

[51] Z. Smith, "Bandwidth: a memory bandwidth benchmark," 2008.

[52] L. S. Blackford, J. Choi, A. Cleary, A. Petitet, R. C. Whaley, J. Demmel, I. Dhillon, K. Stanley, J. Dongarra, S. Hammarling, G. Henry, and D. Walker, "Scalapack: A portable linear algebra library for distributed memory computers - design issues and performance," in *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '96. Washington, DC, USA: IEEE Computer Society, 1996. [Online]. Available: https://doi.org/10.1145/369028.369038

[53] P. Colella, "Defining software requirements for scientific computing," 01 2004.

[54] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.

[55] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," *ACM SIGARCH computer architecture news*, vol. 23, no. 2, pp. 24–36, 1995.

[56] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 44–54.

[57] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[58] J. Dongarra, V. Eijkhout, and H. van der Vorst, "Sparsebench: A sparse iterative benchmark," 2001.

[59] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[60] J. Kepner, D. Koester *et al.*, "Hpcs ssca# 2 graph analysis benchmark specifications v1. 0," 2005.

[61] A. Li, S. L. Song, J. Chen, X. Liu, N. Tallent, and K. Barker, "Tartan: Evaluating modern GPU interconnect via a multi-GPU benchmark suite," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2018, pp. 191–202.

[62] H. Fanaee-T and J. Gama, "Simtensor: A synthetic tensor data generator," *arXiv preprint arXiv:1612.03772*, 2016.

[63] C. A. Andersson and R. Bro, "The N-way Toolbox for MATLAB," *Chemometrics and Intelligent Laboratory Systems*, vol. 52, no. 1, pp. 1–4, Aug 2000.

[64] L. R. Tucker, "Some mathematical notes on three-mode factor analysis," *Psychometrika*, vol. 31, pp. 279–311, 1966.

[65] J. D. Carroll and J.-J. Chang, "Analysis of individual differences in multidimensional scaling via an n-way generalization of "eckart-young" decomposition," *Psychometrika*, vol. 35, no. 3, pp. 283–319, Sep 1970. [Online]. Available: https://doi.org/10.1007/BF02310791

[66] J. D. Carroll, S. Pruzansky, and J. B. Kruskal, "CANDELINC: A general approach to multidimensional analysis of many-way arrays with linear constraints on parameters," *Psychometrika*, vol. 45, pp. 3–24, 1980.

[67] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, R. Lethin, J. McMahon, C. Pawar, M. Perrigo, S. Rucker, J. Ruttenberg, M. Ruttenberg, and S. Stein, "Highly scalable near memory processing with migrating threads on the Emu system architecture," in *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA$\hat{3}$ '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 2–9. [Online]. Available: https://doi.org/10.1109/IA3.2016.7

[68] E. Hein, T. Conte, J. S. Young, S. Eswar, J. Li, P. Lavin, R. Vuduc, and J. Riedy, "An initial characterization of the Emu Chick," *2018 IEEE International Parallel and Distributed Processing Symposium Workshops*, May 2018.