Performance Challenges for Heterogeneous Distributed Tensor Decompositions

Thomas B. Rolinger, Tyler A. Simon, and Christopher D. Krieger
Laboratory for Physical Sciences
University of Maryland, College Park, MD
{tbrolin,tasimon,krieger}@lps.umd.edu

***Abstract*—Tensor decompositions, which are factorizations of multi-dimensional arrays, are becoming increasingly important in large-scale data analytics. A popular tensor decomposition algorithm is Canonical Decomposition/Parallel Factorization using alternating least squares fitting (CP-ALS). Tensors that model real-world applications are often very large and sparse, driving the need for high performance implementations of decomposition algorithms, such as CP-ALS, that can take advantage of many types of compute resources. In this work we present ReFacTo, a heterogeneous distributed tensor decomposition implementation based on DeFacTo, an existing distributed memory approach to CP-ALS. DFacTo reduces the critical routine of CP-ALS to a series of sparse matrix-vector multiplications (SpMVs). ReFacTo leverages GPUs within a cluster via MPI to perform these SpMVs and uses OpenMP threads to parallelize other routines. We evaluate the performance of ReFacTo when using NVIDIA's GPU-based cuSPARSE library and compare it to an alternative implementation that uses Intel's CPU-based Math Kernel Library (MKL) for the SpMV. Furthermore, we provide a discussion of the performance challenges of heterogeneous distributed tensor decompositions based on the results we observed. We find that on up to 32 nodes, the SpMV of ReFacTo when using MKL is up to 6.8x faster than ReFacTo when using cuSPARSE.**

***Keywords*-tensor decomposition, performance evaluation, GPU, parallel programming**

I. Introduction

Multi-dimensional arrays, called *tensors*, are becoming increasingly important in data analytics. Tensors can naturally represent the types of multidimensional data that appear in application domains such as social networks, signal processing, neuroscience, pharmaceutics, and finance. Within such domains, analysts are interested in uncovering previously unknown relationships between elements in the data.

Tensor decomposition has emerged as a useful tool for extracting patterns from these types of data. Tensor decomposition can be thought of as the higher-order analogue to matrix singular value decomposition (SVD). A popular algorithm for extending SVD to higher dimensions is Canonical Decomposition/Parallel Factorization using alternating least squares fitting (CP-ALS) [1].

Typically, tensors produced from real-world applications are very large and sparse, driving the need for high performance implementations of tensor decomposition algorithms like CP-ALS. Applying traditional high performance computing techniques, such as shared or distributed memory parallel programming models like OpenMP and MPI, to CP-ALS has been an active area of research [2]–[4]. Other efforts have focused on using graphics processing units (GPUs) [5].

This paper introduces ReFacTo, a CP-ALS tensor decomposition implementation that is an extension of a previous CP-ALS algorithm, DFacTo [2]. ReFacTo refactors and extends DFacTo to perform heterogeneous distributed tensor decomposition.

The primary contributions of our work are:

1) ReFacTo, a tool to perform tensor decomposition using an MPI based distributed memory approach. Within each node, ReFacTo leverages a GPU to perform the critical component of CP-ALS and uses OpenMP threads for other routines. To the best of our knowledge, this approach is the first within the field of tensor decomposition to utilize heterogeneous compute resources in this way.
2) A comparative performance study of ReFacTo when using GPUs for its critical computation and when using highly vectorized, multithreaded CPU code to perform the same calculations. We follow up with a discussion on the performance challenges that arise in our approach.

Section II presents related work on heterogeneous approaches to tensor decomposition. The DFacTo CP-ALS implementation, which we use as a starting point for the ReFacTo effort, is outlined in Section III. Section IV details ReFacTo's approach to distributed heterogeneous tensor decomposition. Section V describes the experiments we conducted and presents their results. In Section VI, we discuss the performance challenges that arise in our approach. Finally, we provide concluding remarks and avenues for future work in Section VII.

II. Related Work

Our motivation for this work comes from the need to improve the performance of tensor decompositions, particularly on large, realistic datasets [6]. This performance requirement drove us to consider decompositions on Linux clusters, which now often come with a GPU attached to each compute node.

Ballard et al. have demonstrated performance gains from using GPUs for computing tensor decompositions [7] and work from Zhoi et al. has looked at performing tensor decompositions for recommender systems using GPUs [5], [8]. Antikainen et al. have also developed a dynamic library for non-negative tensor factorization using MATLAB and GPUs [9].

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

However, these prior efforts have only looked at processing on a single GPU. There are several implementations of distributed tensor decompositions that leverage multiple compute nodes in a cluster [4], [10], [11], but they do not make use of GPUs. In order to leverage more of the computing resources of a cluster, we introduce a hybrid programming model that includes the Compute Unified Device Architecture (CUDA), MPI, and OpenMP for performing tensor decompositions at scale [12].

III. DFacTo

DFacTo[1] is a distributed CP-ALS implementation developed by researchers at Purdue University that operates on tensors with up to 3 dimensions (often called *modes*) [2]. DFacTo reformats, or *matricizes*, the original tensor along each mode and stores the transpose of these matrices, removing any rows that contain all zeros. These matrices are stored in Compressed Sparse Row (CSR) format and are denoted as $\mathbf{X}_n{}^T$, where n is the mode of the tensor that was matricized.

Algorithm 1 presents DFacTo's approach to computing the matricized tensor times Khatri-Rao product (MTTKRP) for the first mode of the tensor. The MTTKRP is the critical routine in CP-ALS and most of the computational and storage complexity stems from it. R is a scalar and denotes the rank of the decomposition. Assuming a tensor with dimensions I, J, and K, $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are dense matrices of size $I \times$R, $J \times$R and $K \times$R, respectively, and are referred to as *factor matrices*. $\mathbf{X}_2{}^T$ is a CSR matrix of size $IK \times J$ and represents the transpose of the mode-2 matricized tensor. $\mathbf{M}_2{}^r$ is an auxiliary CSR matrix of size $I \times K$. From $\mathbf{X}_2{}^T$, DFacTo computes the non-zero pattern of $\mathbf{M}_2{}^r$ and represents this via the CSR index arrays. This non-zero pattern is completely independent of $\mathbf{B}_{(:,r)}$ and therefore can be pre-allocated. By storing the vector result of $\mathbf{X}_2{}^T\mathbf{B}_{(:,r)}$ in the values array of $\mathbf{M}_2{}^r$, DFacTo effectively "reshapes" $\mathbf{X}_2{}^T\mathbf{B}_{(:,r)}$ into $\mathbf{M}_2{}^r$. $\mathbf{G}_2$ is a dense R$\times$R matrix that represents the inverse of the Gram matrix. It is computed as the inverse of $\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C}$, where "*" is the Hadamard product. Finally, $\mathbf{N}_2$ is a dense matrix of size $I \times$ R and stores the result of the MTTKRP. DFacTo's approach to the MTTKRP is unique in that it reduces the computation to a series of sparse matrix times dense vector multiplications (SpMVs), namely lines 4 and 5, which can be performed by optimized routines from a vendor or open source library.

Algorithm 1 DFacTo `MTTKRP` Algorithm

1: **Input:** R, $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, $\mathbf{X}_2{}^T$, $\mathbf{M}_2{}^r$, $\mathbf{G}_2$
2: **Output:** $\mathbf{N}_2$, $\mathbf{A}$
3: **for** r = 1,...,R **do**
4: $\quad\mathbf{M}_2{}^r.value \leftarrow \mathbf{X}_2{}^T\mathbf{B}_{(:,r)}$
5: $\quad\mathbf{N}_{2(:,r)} \leftarrow \mathbf{M}_2{}^r\mathbf{C}_{(:,r)}$
6: **end for**
7: $\mathbf{A} \leftarrow \mathbf{N}_2\mathbf{G}_2$

[1]DFacto code is available at http://web.ics.purdue.edu/~choi240/

DFacTo is designed for distributed execution over MPI and uses a coarse-grained approach to data distribution. Each MPI process is assigned a set of contiguous slices of the $\mathbf{X}_n{}^T$ and $\mathbf{M}_n{}^r$ matrices and is responsible for the corresponding rows in the factor matrices. This coarse-grained approach has the advantage of simplifying the MTTKRP, only requiring communication at the end of each iteration to exchange updated rows in the factor matrices. However, full copies of the factor matrices are held by each process since a given process may be assigned non-zeros that span all of the modes of the tensor.

DFacTo is implemented in C++ and uses the Eigen library [13] for the storage of dense and sparse matrices as well as for basic linear algebra subprograms (BLAS). Eigen uses parallelized and somewhat vectorized internal routines, but alternatively it can make direct use of Intel's Math Kernel Library[2] (MKL). MKL provides highly efficient, vectorized, and threaded BLAS routines, including SpMV, that have been carefully optimized to run on Intel CPUs.

IV. Approach

As noted in Section III, the MTTKRP is the critical portion of CP-ALS. Therefore, we focused our attention on the key routines performed within the MTTKRP, looking for operations that would benefit the most from acceleration. Across a range of data sets, we found that the SpMV routine in DFacTo, when using Eigen without MKL, consumed an average of 77% of the MTTKRP runtime when using 20 threads on a single node. This result indicates that accelerating the SpMV would provide a large performance gain. The existing DFacto code structure made using different SpMV implementations difficult because it relies entirely on the Eigen library for the storage of matrices and vectors and the operations on them. For example, an SpMV call is done through an overloaded C++ "*" operator. To resolve this, we developed ReFacTo, our own CP-ALS implementation that is based on a refactorization of DFacTo. ReFacTo eliminates the dependency on Eigen, allowing for the direct swapping of SpMV implementations without affecting other routines in the code. This allows us to execute the SpMV operation on CPUs or GPUs without significant changes to the rest of the algorithm.

ReFacTo makes use of the same data distribution scheme and communication patterns via MPI that DFacTo does. This allows ReFacTo to execute across multiple machines and leverage an available GPU on each machine to perform SpMV while also using OpenMP threads to parallelize other work. To the best of our knowledge, this is the first CP-ALS approach to utilize these compute resources together.

A. Use of Standard Routines and Data Structures

DFacTo uses Eigen's C++ classes to store sparse and dense data. This makes it difficult to create GPU-memory versions of structures like the row pointer, column index, and value arrays of CSR matrices as well as leverage Eigen's functions to perform operations on those GPU-memory structures.

[2]https://software.intel.com/en-us/intel-mkl

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

ReFacTo uses a standard implementation of the CSR format and stores the dense factor matrices in standard C row-major order. This allows for full control over the memory allocation of the internal structures and data.

Rather than using routines supplied by the Eigen library, we provide custom implementations for operations such as matrix 2-norm and max-norm, computing the symmetric inverse of the Gram matrix, and matrix multiplication. The implementations for these routines were adopted from SPLATT[3], an open source software toolbox for sparse tensor factorization and related kernels [3], [4], [14]. Many of these routines are parallelized via OpenMP. Since the focus of this work is the SpMV, we do not attempt to further optimize other operations.

B. Sparse Matrix Vector Multiplication on CPUs or GPUs

ReFacTo can call two different SpMV implementations, each of which uses different hardware. NVIDIA's cuSPARSE[4] library provides an SpMV routine that runs on NVIDIA GPUs. Intel's MKL provides a multithreaded, heavily vectorized SpMV for CPUs.

For CPU SpMV computation, we use MKL's `mkl_cspblas_?csrgemv` function. Since MKL runs directly on the CPU, no other modifications need to be made to the overall code to use its SpMV routine. The sparse matrices given to the routine are already in CSR format and do not need to be transposed.

When utilizing cuSPARSE for SpMV, the `cusparse<t>csrmv` function is called. cuSPARSE requires the input matrix and the input and output vectors to be allocated in GPU memory via `cudaMalloc` prior to the library call. Furthermore, the values for the input matrix and vector must be explicitly copied to the GPU via `cudaMemcpy` and the output values in the result vector must be copied from the GPU back to the host CPU prior to being used in other host routines.

Calls to `cudaMalloc` and `cudaMemcpy` are very expensive in terms of runtime, so we eliminate unnecessary calls. We note that the index arrays of the CSR matrices remain unchanged throughout the CP-ALS algorithm. Therefore, we can allocate device memory for these arrays and perform a one-time host-to-device (HtoD) copy to populate them. We also note that the values array of each CSR matrix is only accessed during calls to `cusparse<t>csrmv`, which occur on lines 4 and 5 in Algorithm 1. This is not true in DFacTo, which requires the $\mathbf{X}_2^{\mathrm{T}}$ and $\mathbf{M}_2^{\mathrm{r}}$ matrices when calculating the fit at the end of each iteration of CP-ALS. ReFacTo uses SPLATT's approach to calculating the fit and only requires the factor matrices and the output of the MTTKRP for the last mode of the tensor. Because of this, it is not necessary to perform any HtoD or DtoH copies for the CSR matrices before or after the SpMVs during the decomposition.

The input vectors to the SpMVs on lines 4 and 5 in Algorithm 1, as well as the output vector on line 5, are the only parameters that must be accessed on both the host and device during the decomposition. However, we know that the maximum length of the input/output vectors is equal to the length of the largest mode in the original tensor. Therefore, we can pre-allocate the device memory for a single input vector and output vector of maximum length and reuse these vectors during the MTTKRP, thus eliminating reoccurring calls to `cudaMalloc` within the MTTKRP. We only need to call `cudaMemcpy` prior to each SpMV to populate these vectors with columns from the factor matrices and after the second SpMV to retrieve the result. To this end, we do not require any `cudaMalloc` calls within the MTTKRP and only require three calls to `cudaMemcpy` per iteration of the for-loop in Algorithm 1. We also incorporate these memory allocation optimizations for the MKL version of the code, eliminating unnecessary calls to `malloc`.

It should be noted that within the MTTKRP, each SpMV must be performed sequentially since the output of the first SpMV is required as input to the second SpMV. This prevents significant computation/communication overlap between the SpMV calls.

V. Performance Evaluation

While much has been done to evaluate SpMV implementations on CPUs and GPUs [15], [16], this work is the first to consider the performance of such approaches in the context of distributed tensor decomposition. In this section, we describe our experimental setup and present the performance results of ReFacTo when utilizing MKL and CUDA. For comparison purposes, we also measure the performance of the original DFacTo code when using Eigen without MKL and with Eigen's vectorization disabled. In this configuration, the DFacTo code represents basic parallelized CPU code.

A. Experimental Setup

Performance of each run was measured on a 32 node cluster. Each node had 512 GB of DDR4 DRAM and consisted of two 10 core E5-2650v3 Xeon "Haswell" processors, each running at 2.3 GHz with 25 MB of shared last level cache and the AVX 2.0 extended vector instruction set. Available to each node was a NVIDIA Tesla K40m (Kepler) GPU with 12 GB of global memory, 48 KB of shared memory, a maximum memory bandwidth of 288 GB/sec and a single precision peak performance of 4.29 TFLOPs.

Both ReFacTo and DFacTo were built using the same compiler (icc version 17.0.4), the same version of OpenMP (Intel OpenMP, 4.0 standard), and the same version of MVAPICH for MPI (version 2.2b). MKL version 2017 update 3 was used in ReFacTo as well as CUDA 8.0. DFacTo used Eigen version 3.3 with vectorization disabled by defining the `EIGEN_DONT_VECTORIZE` preprocessor symbol. Single-precision floating point numbers and 32-bit integers were used in both ReFacTo and DFacTo. We refer to the ReFacTo results using MKL as simply *MKL*, the ReFacTo results using cuSPARSE as *CUDA* and the DFacTo results using Eigen as *Eigen*.

[3]SPLATT source code is available from https://github.com/ShadenSmith/splatt

[4]https://developer.nvidia.com/cusparse

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

TABLE I
PROPERTIES OF DATA SETS

Name	Dimensions	Non-Zeros	Density
YELP	41k x 11k x 75k	8M	1.97E-7
CELLAR TRACKER	34k x 363k x 163k	20M	9.81E-9
VAST	165k x 11k x 2	26M	6.91E-3
NELL-2	12k x 9k x 29k	77M	2.4E-5
NELL-1	3M x 2M x 25M	143M	9.05E-13

We evaluated a subset of the data sets from our prior work [6], namely YELP, CELLAR TRACKER, VAST 2015 MC 1 (which we denote as VAST for short), NELL-2 and NELL-1. We present a summary of these data sets in Table I. These data sets are non-synthetic and present a wide range of 3-dimensional tensors in terms of the number of non-zeros and sparsity. A density value of 1 indicates a completely dense tensor. The VAST, NELL-2, and NELL-1 data sets can be obtained from the Formidable Repository of Open Sparse Tensors and Tools (FROSTT) [17].

For each data set, we performed a rank-35 CPD with DFacTo and ReFacTo. Both tools use 20 OpenMP/MKL threads per node. The number of blocks and size of each block used on the GPU(s) for the SpMV is controlled entirely by cuSPARSE and is dependent on the input data. We focus our results on the total time spent performing SpMVs during the decomposition. However, we did measure the total decomposition runtime and observed that those results track with the SpMV runtime results. All results presented are the average of 5 runs.

B. Results

Figure 1 presents the total SpMV runtime of the three different approaches across the data sets when running on 1 node and utilizing 20 threads. Note that due to the vast difference in runtimes between NELL-1 and the other data sets, the vertical axis is logarithmic. It should also be noted that these results only represent the runtime of the SpMV routines (i.e. `mkl_cspblas_?csrgemv` and `cusparse<t>csrmv`) and exclude any memory allocation performed explicitly by ReFacTo/DFacTo as well as HtoD or DtoH communication. ReFacTo when using MKL outperforms the other implementations on all of the data sets with the exception of NELL-1 where the CUDA approach is 50 seconds faster. Eigen is 2.4x – 4.4x slower than MKL and 1.32x – 3.9x slower than CUDA.

Figure 2 shows the SpMV runtimes when running across 32 nodes and utilizing 20 threads per node. Again, the vertical axis is logarithmic. We observe that MKL runs 1.84x – 4.7x faster than Eigen and 2x – 6.8x faster than CUDA. In terms of scalability, Figure 3 shows the SpMV runtime speed-ups achieved by each approach when running on 32 nodes compared to 1 node. MKL and Eigen achieve an average speed-up of 24.8x and 32.8x, respectively, while CUDA only achieves an average speed-up of 12.3x.

We observe that the CUDA code is outperformed by the MKL and Eigen code when running on 32 nodes, suggesting

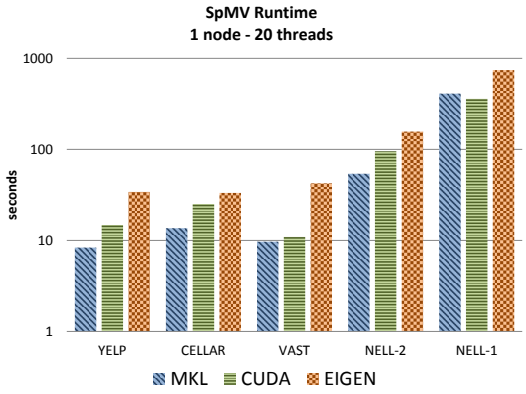

Fig. 1. The total SpMV runtime when running on 1 node and utilizing 20 threads. This represents the total amount of time spent performing SpMV during the decomposition. Note that the vertical axis is logarithmic. Shorter bars represent better performance.

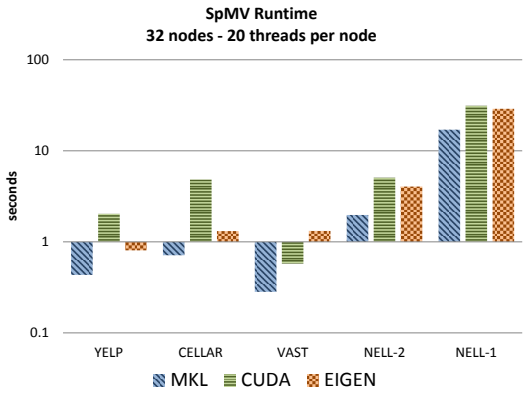

Fig. 2. The total SpMV runtime when running on 32 nodes and utilizing 20 threads per node. This represents the total amount of time spent performing SpMV during the decomposition on each node. Note that the vertical axis is logarithmic. Lower bars represent better performance.

that there are performance challenges to consider for such a heterogeneous distributed tensor decomposition approach. Such challenges are discussed in the next section.

VI. PERFORMANCE CHALLENGES

Based off of the results from the experiments in Section V, we have identified key performance challenges that arise in a heterogeneous approach to distributed tensor decomposition, as outlined below:

1) Maintaining GPU utilization and efficiency when scaling out to many nodes.
2) Caching and data reuse on the GPU.
3) Irregularity in the degree of sparsity of the matricized tensors arising from the original tensor.
4) Performance bottlenecks that are not improved by the introduction of GPUs

We note that as the number of nodes increases, the average achieved occupancy on the GPU for the CUDA code decreases, which affects the runtime performance and scalability of the approach. This is shown in Figure 4. The achieved

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

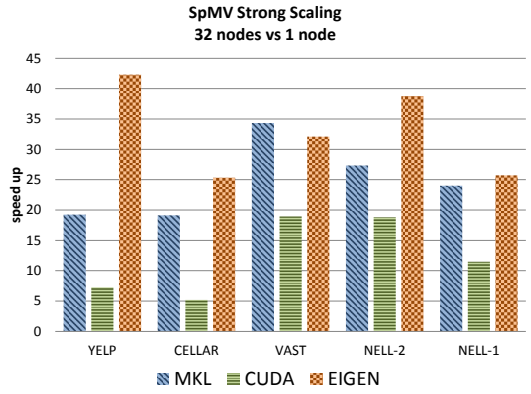

Fig. 3. The SpMV runtime speed-ups achieved when running on 32 nodes compared to 1 node. Larger bars represent better speedup.

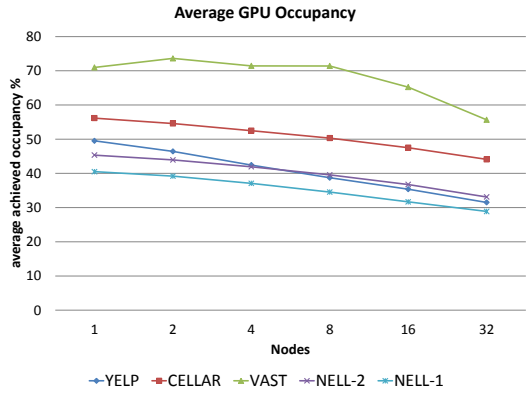

Fig. 4. The average achieved GPU occupancy for each data set when running on 1 to 32 nodes.

occupancy is the ratio of the average number of active warps per active cycle to the maximum number of warps supported on the GPU. Furthermore, as shown in Figure 5, the average floating-point operation (FLOP) efficiency on the GPU decreases as the number of nodes increases, loosely tracking the average achieved occupancy. The average FLOP efficiency is the ratio of achieved single-precision FLOPs to the peak performance of the GPU. On 1 node, we observe that ReFacTo using CUDA achieves expected occupancy rates as well as high FLOP efficiency on most data sets. This is reflected in the runtime results in Figure 1, which show that the CUDA approach is 1.32x – 3.9x faster than Eigen. However, as the number of nodes increases, each MPI rank is responsible for a smaller portion of the sparse matrices, which means that the SpMV routines are operating on a smaller amount of data. This has the effect of decreasing the average GPU occupancy rate and FLOP efficiency, as shown in Figures 4 and 5, and consequently negatively impacting the runtime performance and scalability. Indeed, Figure 2 shows that non-vectorized Eigen code is up to 3.7x faster than the CUDA code on 32 nodes and Figure 3 shows that the CUDA approach suffers greatly in terms of scalability.

The performance of cuSPARSE's SpMV has been shown to

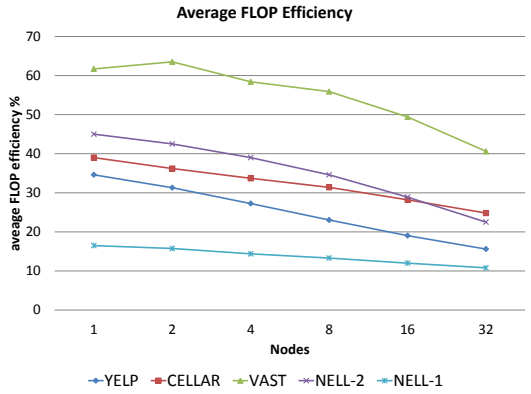

Fig. 5. The average single precision FLOP efficiency on the GPU for each data set when running on 1 to 32 nodes. This represents the percentage of peak FLOPs that were achieved.

be up to 4x faster than MKL on a wide range of matrices [18]. However, the results of the experiments in Section V-B show that MKL outperforms cuSPARSE in our application. To investigate this discrepancy, we converted the matricized tensors from each data set into Matrix Market format [19] and benchmarked the performance of a single SpMV with cuSPARSE and MKL on the matrices, recreating the experiments in [18] with our matrices and single-precision floating point numbers. What we found was that cuSPARSE was indeed about 4x faster than MKL when measuring the performance of a single SpMV. However, the CP-ALS algorithm that ReFacTo implements performs multiple SpMVs in quick succession, reusing the same CSR matrices. Because of this, the MKL version of ReFacTo is able to cache and reuse the CSR matrix data, dramatically increasing the runtime performance for subsequent SpMVs. On the other hand, the CUDA version of ReFacTo must launch a new kernel on the GPU for each SpMV it performs. Furthermore, we note that the largest difference in SpMV runtime performance between the MKL and cuSPARSE implementations occur on data sets where the non-zeros in the matricized tensors are grouped into columns. Figure 6 shows an example of this sparsity pattern for the $\mathbf{X}_3^{\mathrm{T}}$ matrix of the CELLAR-TRACKER data set. Note that the MKL SpMV was 6.8x faster than the cuSPARSE SpMV when running on 32 nodes on this data set. Due to these repeated columnar patterns of non-zeros in the sparse matrix, the same elements in the vector will be accessed row after row during the SpMV and can fit into the CPU cache, providing a significant performance gain for the MKL version. However, the performance for the cuSPARSE version is still hindered by the non-coalesced memory accesses to the vector.

While the sparsity of the original tensor has some bearing on the sparsity of the mode-n matricized tensors, some tensors may produce fairly dense matricized tensors as well as dense $\mathbf{M}_n^{\mathrm{r}}$ matrices, which represent the non-zero patterns in the matricized tensors. For example, the VAST tensor has a density of 0.0069 but its mode-3 matricized tensor is 50% dense and its $\mathbf{M}_1^{\mathrm{r}}$ matrix is completely dense. Thoroughly investigating

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

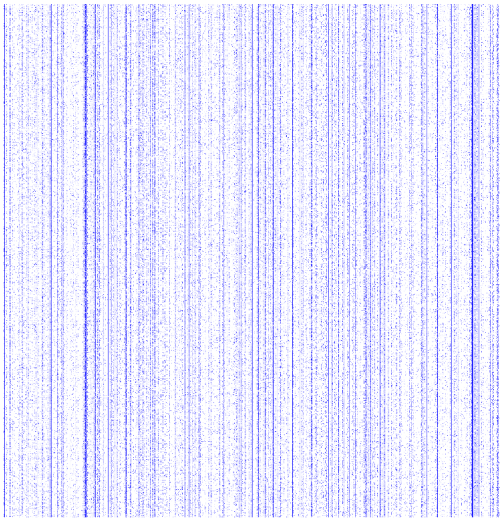

Fig. 6. The sparsity pattern for the $\mathbf{X}_3{}^{\mathrm{T}}$ matrix of the CELLAR-TRACKER data set. The matrix has dimensions 1M x 145K and has 20M non-zeros. Dark areas represent non-zeros.

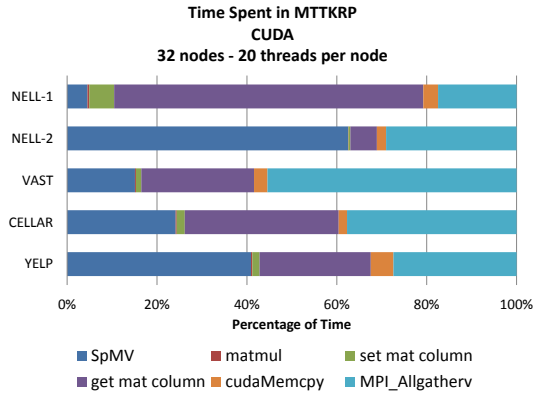

Fig. 7. The percentage of time that each operation within the MTTKRP contributes to the total MTTKRP runtime for ReFacTo when using CUDA on 32 nodes. `matmul` refers to matrix multiplication, `get mat column` and `set mat column` refer to reading/writing a column from/to a row-major matrix, respectively.

the effects of sparsity on the SpMV implementations is beyond the scope of this paper. However, we believe that this irregularity in sparsity affects the overall SpMV performance and warrants careful consideration when choosing which SpMV implementation to employ.

As the performance of the SpMV is improved with efficient parallel implementations, other bottlenecks will arise that are not easily improved by GPUs. In the case of DFacTo and ReFacTo's approach to CP-ALS, these bottlenecks are inefficient memory access patterns and MPI communication. Figure 7 shows the percentage of time that each operation within the MTTKRP contributes to the total MTTKRP runtime for ReFacTo when using CUDA on 32 nodes. DFacTo using Eigen and ReFacTo using MKL show similar trends as well. It can be seen that the SpMV is taking up less than 25% of the time on several data sets. However, the `get mat column` operation is now consuming up to 70% of the total time. This operation refers to reading a column from a dense factor matrix, which is stored in row-major order, and results in poor cache utilization for the MTTKRP [6]. By leveraging GPUs, the operation can be sped up by performing the read in parallel. However, this would require storing all of the dense factor matrices in GPU memory, which can easily consume more memory than is available on most GPUs. Since the factor matrices are not partitioned across MPI ranks, increasing the number of nodes does not alleviate this issue. The final growing cost, MPI communication, consumes an average of 33% of the total MTTKRP runtime. This cost is compounded by the need to move data to/from the GPU before and after the `MPI_Allgatherv` operation.

VII. Conclusions and Future Work

In this work we introduced ReFacTo, a heterogeneous distributed tensor decomposition implementation that can leverage multiple GPUs and nodes as well as OpenMP threads. ReFacTo is based on DFacTo's approach to CP-ALS and attacks the core operation of its MTTKRP, namely a series of SpMVs. We investigated the performance of ReFacTo when using cuSPARSE for SpMV as well as when using MKL. We found that while ReFacTo with cuSPARSE exhibits SpMV speed-ups on up to 32 nodes, its performance is hindered by several factors when compared to CPU-based implementations. We observed that Eigen's non-vectorized parallel implementation of SpMV on 32 nodes is 1.8x faster on average than ReFacTo when using cuSPARSE. On the other hand, ReFacTo when using MKL on up to 32 nodes produces SpMV runtimes that are as much as 4.7x and 6.8x faster than Eigen and cuSPARSE, respectively.

We have highlighted a few challenges that face heterogeneous distributed tensor decomposition based on our experimental results. These challenges consist of maintaining GPU utilization and efficiency as more nodes are used, caching and reusing data on the GPU, dealing with irregularity in the degree of sparsity of the matricized tensors, and overcoming other performance bottlenecks that are not easily alleviated with more nodes and/or GPUs.

Our future work includes exploring alternative GPU-based SpMV implementations and improving MPI communication costs. Steinberger et al. [20] has shown that naive SpMV implementations on the GPU can outperform state-of-the-art GPU-based SpMV implementations in some cases, so it would be worthwhile to investigate alternatives to cuSPARSE for SpMV approaches. Furthermore, we plan to port ReFacTo onto NVIDIA's DGX-1 system, which has multiple GPUs connected via NVLink, and run the entire CP-ALS algorithm on the GPUs. By doing so, the MPI communication can be replaced by GPU-to-GPU communication over NVLink, only returning to the host for synchronization purposes. We also intend to run subsequent experiments on tensors that are larger than those presented in this paper. We suspect that as the tensors become larger, the performance difference between the CPU and GPU versions will become smaller.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

REFERENCES

[1] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," vol. 51, no. 3, September 2009, pp. 455–500.

[2] J. H. Choi and S. Vishwanathan, "Dfacto: Distributed factorization of tensors," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 1296–1304. [Online]. Available: http://papers.nips.cc/paper/5395-dfacto-distributed-factorization-of-tensors.pdf

[3] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "Splatt: Efficient and parallel sparse tensor-matrix multiplication," *29th IEEE International Parallel & Distributed Processing Symposium*, 2015.

[4] S. Smith and G. Karypis, "A medium-grained algorithm for distributed sparse tensor factorization," *30th IEEE International Parallel & Distributed Processing Symposium*, 2016.

[5] B. Zou, C. Li, L. Tan, and H. Chen, "Gputensor: Efficient tensor factorization for context-aware recommendations," *Information Sciences*, vol. 299, pp. 159 – 177, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025514011414

[6] T. B. Rolinger, T. A. Simon, and C. D. Krieger, "Performance evaluation of parallel sparse tensor decomposition implementations," in *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*. IEEE Press, 2016, pp. 54–57.

[7] G. Ballard, T. G. Kolda, and T. Plantenga, "Efficiently computing tensor eigenvalues on a GPU," in *IPDPSW'11: Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. IEEE Computer Society, May 2011, pp. 1340–1348.

[8] B. Zou, M. Lan, C. Li, L. Tan, and H. Chen, *Context-Aware Recommendation Using GPU Based Parallel Tensor Decomposition*. Cham: Springer International Publishing, 2014, pp. 213–226. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-14717-8_17

[9] J. Antikainen, J. Havel, R. Josth, A. Herout, P. Zemcik, and M. Hauta-Kasari, "Nonnegative tensor factorization accelerated using gpgpu," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 7, pp. 1135–1141, Jul. 2011. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2010.194

[10] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos, "Gigatensor: Scaling tensor analysis up by 100 times - algorithms and discoveries," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '12. New York, NY, USA: ACM, 2012, pp. 316–324. [Online]. Available: http://doi.acm.org/10.1145/2339530.2339583

[11] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 77:1–77:11. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807624

[12] A. D. Robison and R. E. Johnson, "Three layer cake for shared-memory programming," in *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ser. ParaPLoP '10. New York, NY, USA: ACM, 2010, pp. 5:1–5:8. [Online]. Available: http://doi.acm.org/10.1145/1953611.1953616

[13] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[14] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, 2015.

[15] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 451–460, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1816021

[16] W. Yang, K. Li, Z. Mo, and K. Li, "Performance optimization using partitioned spmv on gpus and multicore cpus," *IEEE Trans. Computers*, vol. 64, pp. 2623–2636, 2015.

[17] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. (2017) FROSTT: The formidable repository of open sparse tensors and tools. [Online]. Available: http://frostt.io/

[18] "Cuda 7.0 performance report," http://on-demand.gputechconf.com/gtc/2015/webinar/gtc-express-cuda7-performance-overview.pdf, accessed: 2017-07-18.

[19] R. F. Boisvert, R. Pozo, K. Remington, R. F. Barrett, and J. J. Dongarra, "Matrix market: A web resource for test matrix collections," in *Proceedings of the IFIP TC2/WG2.5 Working Conference on Quality of Numerical Software: Assessment and Enhancement*. London, UK, UK: Chapman & Hall, Ltd., 1997, pp. 125–137. [Online]. Available: http://dl.acm.org/citation.cfm?id=265834.265854

[20] M. Steinberger, A. Derlery, R. Zayer, and H. Seidel, "How naive is naive spmv on the gpu?" in *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*, 2016, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/HPEC.2016.7761634

978-1-5386-3472-1/17/$31.00 ©2017 IEEE