Scalable Inference for Sparse Deep Neural Networks using Kokkos Kernels

J. Austin Ellis and Sivasankaran Rajamanickam
Center for Computing Research
Sandia National Laboratories
Albuquerque, USA
{johelli, srajama}@sandia.gov

***Abstract*—Over the last decade, hardware advances have led to the feasibility of training and inference for very large deep neural networks. Sparsified deep neural networks (DNNs) can greatly reduce memory costs and increase throughput of standard DNNs, if loss of accuracy can be controlled. The IEEE HPEC Sparse Deep Neural Network Graph Challenge serves as a testbed for algorithmic and implementation advances to maximize computational performance of sparse deep neural networks. We base our sparse network for DNNs, KK-SpDNN, on the sparse linear algebra kernels within the Kokkos Kernels library. Using the sparse matrix-matrix multiplication in Kokkos Kernels allows us to reuse a highly optimized kernel. We focus on reducing the single node and multi-node runtimes for 12 sparse networks. We test KK-SpDNN on Intel Skylake and Knights Landing architectures and see 120-500x improvement on single node performance over the serial reference implementation. We run in data-parallel mode with MPI to further speed up network inference, ultimately obtaining an edge processing rate of 1.16e+12 on 20 Skylake nodes. This translates to a 13x speed up on 20 nodes compared to our highly optimized multithreaded implementation on a single Skylake node.**

***Index Terms*—sparse neural networks, sparse linear algebra, performance portability**

I. Introduction

Deep neural networks (DNNs) are currently state-of-the-art in many domains including image classification [1], natural language processing [2], [3], and speech recognition tasks [4] DNNs can commonly have millions of network parameters and require billions of operations to evaluate one sample. Pruning of the network weights can reduce both storage requirements and flop count. There exists a large body of work on creating sparse networks from deep networks [5], [6] with the recent primary application driver being machine learning deployment for mobile devices. Recent focus has targeted sparse training to accelerate deep network training [7], [8].

The IEEE HPEC Sparse Deep Neural Network Graph Challenge [9] is a competition to encourage improvements in sparse DNN inference performance. Sparse inference itself has also been a research focus in the context of DNNs [10], [11]. This includes optimizing the sparse matrix-matrix kernels for DNNs or carefully maintaining data sparsity, for example. Other performance improvements are possible through exploiting parallelism. In sparse inference, there are two main ways to exploit parallelism for DNN training and inference: data- and model-parallelism [12]–[14]. Data-parallelism exploits the independence of samples for both training and inference. In the context of sparse inference, input data can either be sparse or dense as well. If the data is dense, rows are partitioned. If the data is sparse, then ensuring that nonzeros are reasonably distributed becomes essential. Model-parallelism may further improve training performance once data-parallelism stagnates, as accuracy with data-parallelism struggles to scale with large training batch sizes [13]. The model-parallel inference task uses distributed memory sparse matrix-matrix multiplication as its core kernel [15], which relies heavily on MPI. Data-parallelism for the inference task could sufficiently strong scale to high node counts, as long as load balance is maintained. With the ever growing popularity of mobile devices and the Internet of Things, sparse inference and training will remain an important task for researchers to optimize.

In this paper, we describe our approach to the Sparse Deep Neural Network Graph Challenge and introduce a performance portable Kokkos Kernels-based implementation of sparse deep neural networks (KK-SpDNN). Experiments include both single node and multi-node results on Intel Skylake and Knights Landing (KNL) platforms. We obtain up to 500x improvement over a serial reference implementation with a single Skylake node. The data-parallel mode obtains up to a 13x improvement on 20 Skylake nodes and a 7x improvement on 12 KNL nodes compared to our highy optimized multithreaded implementation on a single Skylake/KNL node, respectively. We also provide an analysis of performance per layer and multi-node load balancing observations. We conclude with potential pathways for future performance gains.

II. Background

The sparse deep neural network graph challenge consists of 12 sparse deep neural networks of varying sizes and depth, where size of the network is a fixed number of neurons per layer and depth is the number of layers. In this challenge, the number of neurons is either {1024, 4096, 16384, or 65536} and the number of layers is either {120, 480, or 1920}. Each layer in the network is a square matrix of fixed size ($N_{neurons} \times N_{neurons}$). The challenge networks are produced using the sparse DNN generator, RadiX-Net [16], which deterministically produces more diverse networks than other sparse network generators, such as X-Net [17].

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

For each layer, the network iteration is

$$Y_{\ell+1} = \phi(Y_\ell W_\ell + b_\ell), \tag{1}$$

where $Y_\ell \in \mathbb{R}^{(N_{vectors} \times N_{neurons})}$ is the image features at layer ℓ, $\phi(\cdot)$ is an activation function, $W_\ell \in \mathbb{R}^{(N_{neurons} \times N_{neurons})}$ is the layer ℓ network matrix, and $b_\ell \in \mathbb{R}^{(1 \times N_{neurons})}$ is a layer bias. After the layer's matrix-matrix multiplication, b_ℓ is only applied to the nonzeros in the feature matrix. Next, the network applies $\phi(\cdot)$, a modified ReLU activation function,

$$\phi(y_i) = \begin{cases} 0, & y_i \leq 0, \\ y_i, & 0 < y_i < 32, \\ 32, & y_i \geq 32. \end{cases} \tag{2}$$

The sparse network input, Y_0, is the MNIST corpus [18] that contains 60,000 handwriting images at various resolutions. The 16384 neuron network, for example, accepts images with resolution 128×128 pixels (or 16384 total). Both the network layers and images are provided in the sparse coordinate list format, {row, column, value}.

Table I contains the number of nonzeros in each network, the number of nonzeros in the MNIST images, and the layer bias values. The largest test case, even though sparse, requires over 16GB of storage.

TABLE I
THE NUMBER OF NONZEROS (EDGES) IN NETWORK AND IMAGE FOR EACH NEURON-LAYER PAIR. BIAS APPLIED AFTER EACH LAYER.

Neurons vs Layers	120	480	1920	Image Nonzeros	Bias
1024	3.93e6	1.57e7	6.29e7	6.37e6	-.30
4096	1.57e7	6.29e7	2.52e8	2.50e7	-.35
16384	6.29e7	2.52e8	1.01e9	9.89e7	-.40
65536	2.52e8	1.01e9	4.03e9	3.92e8	-.45

The goal of the sparse deep neural network graph challenge is to achieve the best edge rate for the inference problem. The network's *edge rate* is

$$R_{edge} = \frac{N_{vectors} N_{edges}}{t_{inference}}, \tag{3}$$

where $N_{vectors}$ is the size of the MNIST corpus (i.e. 60k images). It is important that approaches be transferrable to real-world sparse data and networks, so exploits of the RadiX network structure and MNIST data are forbidden.

III. APPROACH

Kokkos [19] is a C++ based programming model for writing performance portable applications for all major architectures and HPC platforms. It allows developers to abstract out both parallel execution and data management, specifically for complex heterogenous architectures. The Kokkos Kernels package [20] is a suite of sparse/dense linear algebra and graph kernels built on top of Kokkos. The sparse generalized matrix-matrix multiplication (SpGEMM) kernel [21], [22] within Kokkos Kernels is the backbone of KK-SpDNN.

Kokkos Kernels SpGEMM is designed to be a performance portable implementation that can run well on CPUs, KNLs and GPUs. In order to be performance portable, the method is designed to be a two-phase matrix-matrix multiplication. The first phase computes the number of non-zeroes in the result matrix so that the result matrix can be allocated. The second phase actually computes the matrix-matrix multiplication. This two phase approach is critical for portability so that one can allocate the result matrix in an architecture like GPU where dyamic allocations are inefficient. We utilize this portable two-phase approach even though our target architectures in this work are CPUs and KNLs. This allows reuse of optimized SpGEMM between machine learning, scientific computing and graph analysis use cases. This SpGEMM implementation allows efficient load balancing of the computation and minimizes data movement using a compression operation. The ability to use sparse hashmap based accumulators and dense accumulators allows the implementation to choose the best option for the given problem.

Algorithm 1 Sparse neural network inference

```
KK_SpDNN($N_{neurons}, N_{layers}, \tau_{trim}$)

  Read MNIST images $Y_0$ for $N_{neurons}$ as CRS matrix.
  Read network $W_\ell$ for $N_{neurons}$ and $N_{layers}$ as CRS matrices.
  Read truth categories $\bar{y}$ for $N_{neurons}$ and $N_{layers}$ as vector.
  Build bias vectors $b_\ell$.
  Begin challenge timer.
  for $\ell \in \{0, \cdots, N_{layers} - 1\}$ do
    $Z_\ell$ = KK_SPGEMM($Y_\ell, W_\ell$).
    $\tilde{Z}_\ell(i,:) = \phi(Z_\ell(i,:) + b_\ell)$ (Add bias to $Z_\ell$ nonzeros).
    if $N_{nzs}(Y_\ell) - N_{nzs}(\tilde{Z}_\ell) > \tau_{trim} \cdot N_{nzs}(Y_\ell)$ then
      Trim newly created zeros from $\tilde{Z}_\ell$.
    end if
    $Y_{\ell+1} = \tilde{Z}_\ell$.
  end for
  End challenge timer.
  Compare final rows of $Y$ with $\bar{y}$.
```

KK-SpGEMM accepts matrices in compressed row storage (CRS) format through the Kokkos Kernels `CRS_Matrix` class. To reduce reading time, we create binary files for all layers, inputs, and truth categories prior to calculations. In Algorithm 1, we present the KK-SpDNN algorithm. Here, the operator $N_{nzs}(\cdot)$ returns the number of nonzeros in the given sparse matrix and τ_{trim} controls the frequency of zero trimming.

Importantly, the bias and activation function in Algorithm 1 produce new zeros in Y_ℓ, and, if not managed, can lead to dense linear algebra. We have implemented a Kokkos-based trim kernel that removes all zeros and assembles a new CRS matrix. The trimming process is expensive when it is implemented outside the SpGEMM. As it requires reading the matrix again into the memory hierarchy. To avoid this expensive operation, we also implement a simple heuristic to determine whether trimming is actually necessary. If the difference in nonzeros between the previous and current iterations are above a relative threshold, then $\tilde{Z}_\ell$ is trimmed. A good default is $\tau_{trim} = .01$. As opposed to a fixed trim-frequency, this choice would be most appropriate when dealing with unknown sparse networks whose sparsity patterns may or may not change greatly layer to layer. KK-SpDNN's

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

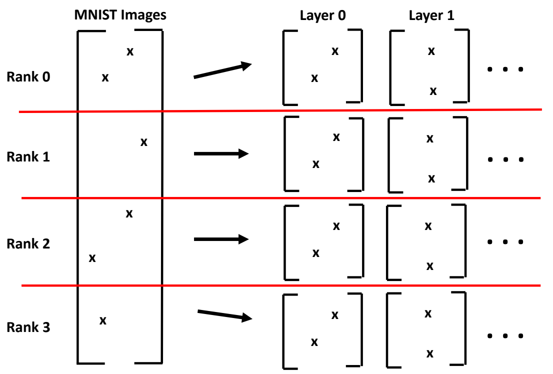

Fig. 1. KK-SpDNN data-parallel mode on 4 ranks. Network layers must be replicated on each rank.

TABLE II
NODE SPECIFICATIONS OF SANDIA NATIONAL LABORATORIES' ASC ADVANCED ARCHITECTURES TEST-BED.

	Skylake machine (Blake)	**KNL machine (Voltrino)**
Processor	Intel Xeon Platinum 8160 processors at 2.10GHz	Intel Xeon Phi 7250 processors at 1.40GHz
Cores	24 cores per socket	68 cores per socket
Sockets	two sockets	one socket
Memory	196GB	99 GB
Cache	1MB L2 Cache 34MB L3 Cache	1MB L2 Cache

optimized trim kernel along with the trim policy is less than 4% of total runtime, as opposed to naive implementation which can be 30-40% of total runtime. Further future optimizations could include fusing the SpGEMM, bias/activation, and trim kernels to minimize data access. But, given that the trim and activation/bias kernels are inexpensive after using our heuristic for reducing trimming frequency, the impact would be low except near the strong scaling limit. We implement the application of the bias and activation function using Kokkos, as well.

We also add data-parallel mode to KK-SpDNN that allows distributed-memory parallelism for increased edge rate. In the data-parallel mode, the rows of Y_0 are evenly distributed across different ranks before the challenge timer begins, see Figure 1, and the network itself must be replicated across all ranks.

Data-parallelization leads to much higher edge rates, but can also create load balancing issues which may prevent scalability (see Figures 5-6 and discussion in Section IV). The rows may be evenly distributed but nonzeros in the CRS format need not be. Also, even if the nonzeros of Y_0 were perfectly balanced, it does not ensure that they will remain balanced as Y_ℓ passes through the network. Throughout the calculation certain rows of Y_ℓ become relatively dense (see Figure 7 in Section IV). A possible cheat for improved load balance and increased edge rates would be to use the truth categories to partition rows in Y_0. We do not use this information. The RadiX networks converge the Y_ℓ quickly, so the majority of the layers are acting on the final solution. This would lead to near optimal static load balancing, though this is directly prohibited by the Graph Challenge rules. However, a dynamic load balancing scheme that balances different layers could be implemented with a balance policy, similar to the trimming policy. We plan to do this optimization in the future.

IV. NUMERICAL RESULTS

The numerical experiments were completed on the ASC Advanced Architectures test-bed at Sandia National Laboratories. The specification of the two test-beds are in Table II

KK-SpDNN obtains the best performance using 1 rank per node with 32 and 128 OpenMP threads for the Skylake and KNL nodes, respectively. On the Skylake machine, we use the `openmpi/2.1.5` and `intel/19.1.144` compilers. For the KNL machine, we use the `mpich/7.7.4` and `intel/18.0.5` compilers. We also set `OMP_PROC_BIND=spread` and `OMP_PLACES=threads` for all cases.

We compare the performance of KK-SpDNN, the Graph Challenge reference serial runtimes [9], and LAGraph [23]. The Graph Challenge reference provides both serial and data-parallel runtimes. Data-parallel calculations are performed on Intel KNL processors with 192GB of memory. LAGraph is based on SuiteSparse: GraphBLAS V3.0.0 [24], [25] [1]. We complete the LAGraph calculations on the same two testbeds as KK-SpDNN to ensure fair comparisons. Though, LAGraph is a single node code, so comparisons are not made against KK-SpDNN's multi-node data-parallel mode.

A. Single Node Results

In Table III, we compare performance on a single Skylake node between KK-SpDNN, the Graph Challenge serial reference implementation, and LAGraph. We report the best results having tested over various numbers of OpenMP threads. KK-SpDNN and LAGraph see a 300-500x improvement over the serial reference implementation. This is a further 10-15x improvement over the theoretical 32x improvement from 32 threads. Comparing LAGraph and KK-SpDNN, each has domains where one dominates the other. LAGraph performs well for the 65536 neuron cases, but KK-SpDNN dominates the 4096 cases. Increasing the problem size, in either neuron or layer count, increases the total work by 4. The KK-SpDNN scales roughly linearly in both dimensions, with exception of the 16384 neuron to 65536 neuron transition. This transition required the switch from `int` to `long` to index the nonzeros, and so performance was impacted. Also, LAGraph uses `float`, where as KK-SpDNN uses `double`, for scalar types. We plan to do further KK-SpDNN experiments with `float` scalar types in the future.

In Table IV, we have the same comparisons on a single KNL node. Note, missing entries for the 65536 neuron cases are due to excessive runtimes. For the KNLs, KK-SpDNN obtains speedups of 35-340x over the Graph Challenge serial reference implementation. Overall, Skylake nodes perform

[1] GraphBLAS V3.0.0 is not publicly available currently. We thank Timothy Davis for sharing a pre-release version.

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

TABLE III
BEST RESULTS WITH OPENMP ON 1 SKYLAKE NODE IN SECONDS WITH NTHREADS IN PARENTHESES. THE GRAPH CHALLENGE SERIAL REFERENCE RUNTIMES ARE REPORTED IN [9]

Neurons, Layers	**Graph Challenge Serial Reference**	**LAGraph V3.0.0**	**KK-SpDNN**
1024, 120	6.26e2 (1)	2.12e0 (20)	**2.00e0** (32)
1024, 480	2.44e3 (1)	**4.78e0** (20)	7.94e0 (32)
1024, 1920	9.76e3 (1)	**1.57e1** (20)	2.76e1 (32)
4096, 120	2.45e3 (1)	1.06e1 (40)	**5.58e0** (32)
4096, 480	1.02e4 (1)	3.43e1 (40)	**2.03e1** (32)
4096, 1920	4.02e4 (1)	1.29e2 (40)	**8.39e1** (32)
16384, 120	1.10e4 (1)	**2.97e1** (32)	**2.95e1** (32)
16384, 480	4.53e4 (1)	**1.08e2** (32)	**1.11e2** (32)
16384, 1920	1.79e5 (1)	**3.88e2** (32)	4.43e2 (32)
65536, 120	4.58e4 (1)	**1.11e2** (32)	3.85e2 (32)
65536, 480	2.02e5 (1)	**3.75e2** (32)	1.54e3 (32)
65536, 1920	—	**1.82e3** (32)	6.20e3 (32)

TABLE IV
BEST RESULTS WITH OPENMP ON 1 KNL NODE IN SECONDS WITH NTHREADS IN PARENTHESES. THE GRAPH CHALLENGE SERIAL REFERENCE RUNTIMES ARE REPORTED IN [9]

Neurons, Layers	**Graph Challenge Serial Reference**	**LAGraph V3.0.0**	**KK-SpDNN**
1024, 120	6.26e2 (1)	2.95e0 (68)	**2.20e0** (128)
1024, 480	2.44e3 (1)	**7.26e0** (68)	7.50e0 (128)
1024, 1920	9.76e3 (1)	**2.53e1** (68)	2.88e1 (128)
4096, 120	2.45e3 (1)	1.65e1 (68)	**1.08e1** (128)
4096, 480	1.02e4 (1)	5.32e1 (68)	**3.98e1** (128)
4096, 1920	4.02e4 (1)	2.01e2 (68)	**1.59e2** (128)
16384, 120	1.10e4 (1)	8.84e1 (68)	**8.07e1** (128)
16384, 480	4.53e4 (1)	**3.15e2** (68)	**3.12e2** (128)
16384, 1920	1.79e5 (1)	**1.24e3** (68)	**1.24e3** (128)
65536, 120	4.58e4 (1)	**6.44e2** (68)	1.31e3 (128)
65536, 480	2.02e5 (1)	—	**5.42e3** (128)
65536, 1920	—	—	—

equally well for smaller cases but is more performant at higher neuron counts. We see the same linear scaling as problem size increases in layer count, but the scaling for neuron count is sublinear for both LAGraph and KK-SpDNN. Comparing LAGraph and KK-SpDNN, KK-SpDNN is more dominant overall on the KNL nodes.

B. Multi-node Results

The KK-SpDNN's data-parallel mode greatly increases the inference edge rate by dividing work amongst independently running nodes. Parallel processing of feature vectors increases throughput in all cases. Figure 2 and Table V contain a strong scaling study on 24 Skylake nodes. The reference implementation reports a maximum edge processing rate of 2.0e11 in data-parallel mode on 600 KNL processors [9], indicated by the redline in Figure 2. We use this best result of the reference implementation to compare all our results. At 4 nodes, we have all test cases surpassing the reference rate. Edge rates continue to scale up to 20 nodes with a maximum edge rate of 1.16e12. After 20 nodes we see a decrease in performance due to load imbalances. This is partially due to the fact we are using test bed machines, and partially due to load balancing issues discussed in Section IV-C. In the Graph Challenge reference implementation's KNL scaling study [9], the highest edge rates occur for the 1024 neuron cases followed by the 4096 cases. For our experiments on Skylake nodes, KK-SpDNN performs best for the 4096 neuron cases then followed by the 16384 neuron cases.

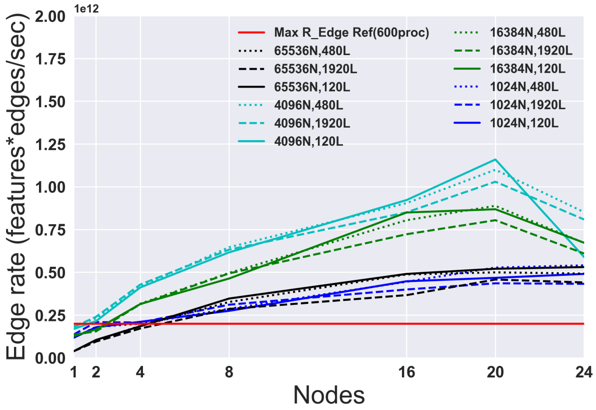

Fig. 2. Strong scaling study of KK data-parallel mode on Skylake nodes. The best rate achieved by the data parallel reference implementation among all the inputs (2.0e11) is shown as a single line. **We achieve 1.16e12 rate with 20 Skylake nodes.**

Figure 3 and Table VI contain a strong scaling study on 16 KNL nodes. The 1024 neuron and 480 layer case achieves the maximum KNL edge rate of 2.80e11 with 12 nodes. The KNL scaling results achieve lower peak rates than Skylake results. The 1024 neuron cases surpass the max reference rate at 4 KNL nodes, but scaling stagnates at 12 nodes. Stagnation occurs because the variance in runtimes can be up to $40-60\%$ around the mean at high node counts. For example, the mean runtime across 12 nodes of the 1024 neuron and 1920 layer case is 9.2 seconds but the maximum runtime is 13.5 seconds and minimum 6.58 seconds. Similarly, the mean runtime across 12 nodes of the 16384 neuron and 1920 layer case is 292.7 seconds, but the maximum is 462.6 seconds and minimum 174.4 seconds.

C. Load Imbalance

The individual kernel timings for each layer in the calculation are seen in Figure 4. The amount of work for each layer quickly stabilizes around a periodic pattern, where Figure 4 is representative of all neuron and layer counts. Networks with more layers maintain the same periodic pattern for all additional layers in the network. Even though there is a greater amount of work in the first few layers, the average runtime time of the periodic pattern largely determines the total runtime of the calculation. Once the layer runtimes has converged to the periodic pattern, the sparsity of the Y_ℓ matrix remains fixed. The trim heuristic is aptly suited for these deep RadiX networks, because the sparsity pattern of Y_ℓ converges quite quickly. If the sparsity pattern is unchanged, we may save 90% of trimming time by skipping the trim kernel in

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

TABLE V
SCALING STUDY OF KK-SPDNN WITH OPENMP ON SKYLAKE NODES IN SECONDS WITH NTHREADS = 32.

Neurons, Layers	2 Nodes	4 Nodes	8 Nodes	16 Nodes	20 Nodes	24 Nodes	Maximum Edge Rate
1024, 120	1.32e0 (32)	1.12e0 (32)	8.58e-1 (32)	5.27e-1 (32)	5.04e-1 (32)	**4.75e-1** (32)	4.91e11
1024, 480	5.22e0 (32)	4.72e0 (32)	3.29e0 (32)	2.12e0 (32)	**1.79e0** (32)	**1.77e0** (32)	5.41e11
1024, 1920	1.81e1 (32)	1.82e1 (32)	1.21e1 (32)	9.43e0 (32)	8.65e0 (32)	**8.00e0** (32)	4.72e11
4096, 120	4.38e0 (32)	2.29e0 (32)	1.53e0 (32)	1.02e0 (32)	**8.14e-1** (32)	1.12e0 (32)	*1.16e12*
4096, 480	1.64e1 (32)	9.04e0 (32)	5.84e0 (32)	4.17e0 (32)	**3.42e0** (32)	4.76e0 (32)	1.10e12
4096, 1920	6.29e1 (32)	3.52e1 (32)	2.39e1 (32)	1.77e1 (32)	**1.47e1** (32)	1.87e1 (32)	1.03e12
16384, 120	2.27e1 (32)	1.20e1 (32)	8.14e0 (32)	4.44e0 (32)	**4.34e0** (32)	5.56e0 (32)	8.69e11
16384, 480	9.75e1 (32)	4.77e1 (32)	3.04e1 (32)	1.88e1 (32)	**1.70e1** (32)	2.28e1 (32)	8.89e11
16384, 1920	3.89e2 (32)	1.92e2 (32)	1.22e2 (32)	8.36e1 (32)	**7.51e1** (32)	9.61e1 (32)	8.05e11
65536, 120	1.44e2 (32)	8.15e1 (32)	4.36e1 (32)	3.07e1 (32)	**2.89e1** (32)	**2.90e1** (32)	5.31e11
65536, 480	5.89e2 (32)	3.22e2 (32)	1.85e2 (32)	1.24e2 (32)	**1.21e2** (32)	**1.22e2** (32)	5.00e11
65536, 1920	2.54e3 (32)	1.40e3 (32)	8.43e2 (32)	6.59e2 (32)	**5.28e2** (32)	5.48e2 (32)	4.58e11

TABLE VI
SCALING STUDY OF KK-SPDNN WITH OPENMP ON KNL NODES IN SECONDS WITH NTHREADS = 128.

Neurons, Layers	2 Nodes	4 Nodes	8 Nodes	12 Nodes	16 Nodes	Maximum Edge Rate
1024, 120	1.50e0 (128)	1.06e0 (128)	9.49e-1 (128)	**8.54e-1** (128)	9.44e-1 (128)	2.76e11
1024, 480	5.32e0 (128)	3.93e0 (128)	3.73e0 (128)	**3.37e0** (128)	3.81e0 (128)	2.80e11
1024, 1920	2.10e1 (128)	1.54e1 (128)	1.47e1 (128)	**1.35e1** (128)	1.55e1 (128)	2.80e11
4096, 120	7.37e0 (128)	5.28e0 (128)	6.17e0 (128)	**4.45e0** (128)	4.56e0 (128)	2.12e11
4096, 480	2.75e1 (128)	2.06e1 (128)	2.48e1 (128)	**1.75e1** (128)	**1.81e1** (128)	2.16e11
4096, 1920	1.11e2 (128)	8.21e1 (128)	9.95e1 (128)	**7.00e1** (128)	7.26e1 (128)	2.16e11
16384, 120	7.00e1 (128)	4.02e1 (128)	3.56e1 (128)	**2.98e1** (128)	3.20e1 (128)	1.27e11
16384, 480	2.62e2 (128)	1.50e2 (128)	1.38e2 (128)	**1.18e2** (128)	1.30e2 (128)	1.28e11
16384, 1920	9.48e2 (128)	6.00e2 (128)	5.34e2 (128)	**4.63e2** (128)	5.15e2 (128)	1.31e11
65536, 120	7.38e2 (128)	3.93e2 (128)	2.45e2 (128)	1.94e2 (128)	**1.66e2** (128)	9.11e10
65536, 480	3.07e3 (128)	1.61e3 (128)	1.00e3 (128)	8.10e2 (128)	**7.06e2** (128)	8.58e10
65536, 1920	1.22e4 (128)	6.37e3 (128)	3.95e3 (128)	3.24e3 (128)	**2.87e3** (128)	8.43e10

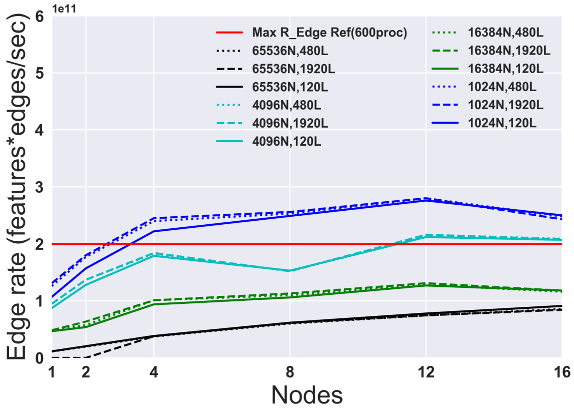

Fig. 3. Strong scaling study of KK data-parallel mode on KNL nodes. The best rate achieved by the data parallel reference implementation among all the inputs (2.0e11) is shown as a single line.

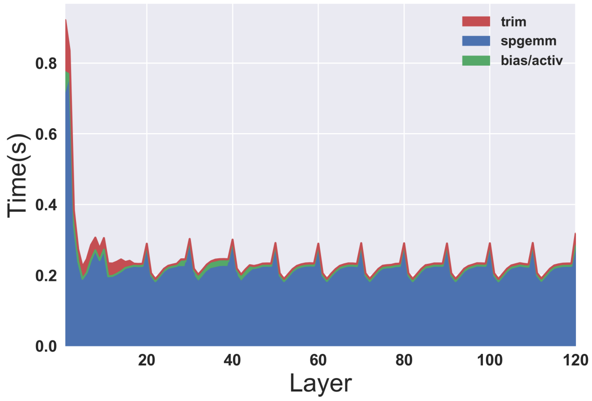

Fig. 4. Timings for each kernel per layer in KK-SpDNN for the 1 Skylake node, 16384 neuron, 120 layer case.

Algorithm 1. Consequently, the vast majority of total runtime is spent in the SpGEMM kernel.

Figures 5 and 6 contain the runtime max/mins and quartiles for all node counts in the 16384 neuron and 1920 layer calculation on both Skylake and KNL machines. The 24 Skylake and 16 KNL node cases, for example, experience a pronounced straggler effect. Parallel runtimes are limited by the slowest rank, so one poorly imbalanced rank can hurt parallel scaling. Also, the runtime variation between ranks in the same calculation is near 50 and 300 seconds for the Skylake and KNL experiments, respectively. This is over 50%

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

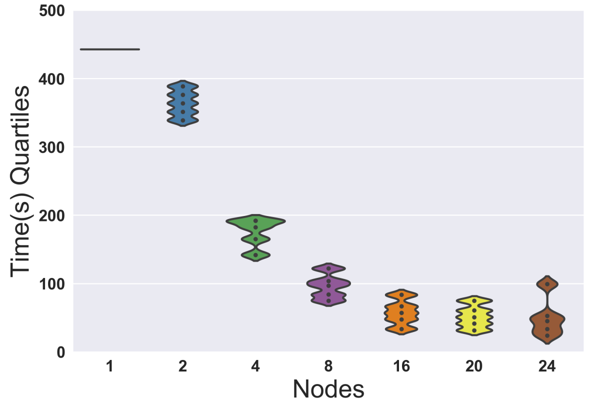

Fig. 5. Load imbalance for KK-SpDNN on Skylake nodes, 16384 neuron, 1920 layer case.

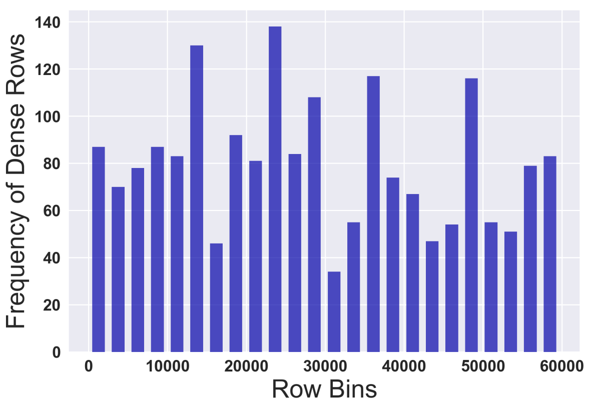

Fig. 7. Load imbalance of the truth categories on 24 nodes, 16384 neurons and 1920 layer case. The x-axis has the rows binned into 24 bins. The y-axis is the number of dense rows in each bin, or each node's load.

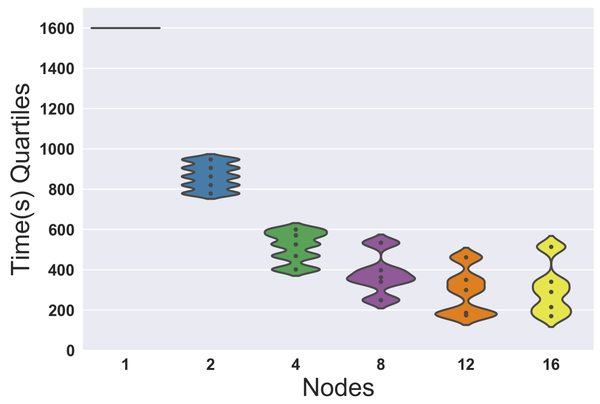

Fig. 6. Load imbalance for KK-SpDNN on KNL nodes, 16384 neuron, 1920 layer case.

of the total runtime at these node counts, so further scaling will likely see greatly decreased benefit. A histogram of the load imbalance from an example final solution is seen in Figure 7, where the maximum load is roughly 2x the average load. Each bar represents a node in the multi-node calculation, where the frequency is the number of dense rows that appear in the converged solution. Because, the solution converges so quickly for the RadiX networks, Figure 7 shows the load for the majority of the calculation.

The largest gains, other than optimizing SpGEMM for sparse inference, will most likely come from judicious dynamic load balancing. A static partitioning can not be optimally chosen without a priori knowledge of the solution's load imbalance. The simple approach of partitioning just Y_0 (i.e. balance nonzeros between ranks) is insufficient for these networks. The networks quickly change load within the first few layers. Furthermore, the nonzeros of Y_0 are already fairly balanced with less than 10% variation between ranks for the node counts tested. A dynamic load balancing algorithm would alleviate the load imbalance issue while also being applicable to real world networks. As long as the migration costs are low and migrations infrequent, dynamic load balancing may greatly reduce overall runtime and improve scalability.

V. Conclusions and Future Work

The KokkosKernels-based sparse deep neural network for both single- and multi-node calculations has greatly improved edge rate for all 12 test cases over the reference runtimes. The maximum edge rate is 1.16e+12, which is achieved for the 4096 neuron and 120 layer case on 20 Skylake nodes. We have proposed a heuristic to optimally trim the CRS matrix, Y_ℓ, of zeros created by the activation function. This optimization reduces trim kernel time from 30% to less than 4%. Greater than 95% of runtime is attributed to the SpGEMM kernel.

We have demonstrated the clear benefit of data-parallelism through increased edge rates, though scaling to higher node counts shows diminishing returns. Changes in load balance during the calculation results in worse performance at higher node counts, but we have proposed plausible pathways to further improve the KK-SpDNN implementation. We see dynamic load balancing as the clearest source of potential improvement. Other potential improvements may include inference-optimized SpGEMM GPU kernels. The largest cases would need to make use of CUDA streams due to GPU memory limitations. Moreover, reduced or mixed-precision arithmetic may accelerate the SpGEMM kernels, translating directly to reduced total runtime. The loss of representative bits from reduced precision is unlikely to affect accuracy for this challenge, because the final goal is identifying the correct sparsity pattern.

Acknowledgments

We thank Timothy Davis for giving us access to a pre-release version of LAGraph V3.0.0. We thank the ASC Advanced Architectures test-bed team at Sandia National Laboratories for supplying and supporting the systems used in this paper. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE CVPR*, 2016, pp. 770–778.

[2] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *arXiv:1810.04805v2*, 2019.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in NIPS 30: Annual Conference*, 2017, pp. 6000–6010.

[4] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "WaveNet: a generative model for raw audio," in *The 9th ISCA Speech Synthesis Workshop*, 2016, pp. 125–140.

[5] T. Gale, E. Elsen, and S. Hooker, "The state of sparsity in deep neural networks," *CoRR, abs/1902.09574*, 2019.

[6] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in NIPS*, 2015, pp. 1135–1143.

[7] J. Frankle and M. Carbin, "The Lottery Ticket Hypothesis: Finding sparse, trainable neural networks," in *ICLR*, 2019.

[8] T. Dettmers and L. Zettlemoyer, "Sparse networks from scratch: Faster training without losing performance," *arXiv:1907.04840*, 2019.

[9] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, L. Robinett, and S. Samsi, "Sparse Deep Neural Network Graph Challenge," https://graphchallenge.mit.edu/challenges, 2019.

[10] X. Liu, J. Pool, S. Han, and W. J. Dally, "Efficient sparse-winograd convolutional neural networks," *arXiv:1802.06367*, 2018.

[11] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," *Proceedings of the 44th Annual International Symposium on Computer Architecture - ISCA '17*, 2017. [Online]. Available: http://dx.doi.org/10.1145/3079856.3080254

[12] A. Krizhevsky, "One Weird Trick for Parallelizing Convolutional Neural Networks," *arXiv:1404.5997v2*, 2014.

[13] M. Wang, C.-c. Huang, and J. Li, "Unifying Data, Model and Hybrid Parallelism in Deep Learning via Tensor Tiling," *arXiv:1805.04170v1*, 2018.

[14] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks," *arXiv:1802.04924v2*, 2018.

[15] A. Buluç and J. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: implementation and experiments," *SIAM J. Sci. Comput*, vol. 34, no. 4, pp. C170–C191, 2012.

[16] R. Robinett and J. Kepner, "RadiX-Net: Structured sparse matrices for deep neural networks," in *IEEE IPDPS GrAPL Workshop 2*, 2019.

[17] Prabhu, G. Varma, and A. Namboodiri, "Deep Expander Networks: Efficient deep networks from graph theory," *CoRR, abs/1711.08757*, 2017.

[18] Y. LeCun and C. Cortes, "The MNIST database of handwritten digits," http://yann.lecun.com/exdb/mnist/, 1998.

[19] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[20] "Kokkos Kernels," https://github.com/kokkos/kokkos-kernels, 2017.

[21] M. Deveci, C. Trott, and S. Rajamanickam, "Performance-portable sparse matrix-matrix multiplication for many-core architectures," in *IEEE IPDPSW*, 2017, pp. 693–702.

[22] ——, "Multithreaded sparse matrix-matrix multiplication for many-core and gpu architectures," *Parallel Computing*, pp. 33–46, 2018.

[23] T. Davis, "LAGraph," https://github.com/GraphBLAS/LAGraph, 2019.

[24] ——, "Algorithm 9xx: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra," *To appear ACM Transactions on Mathematical Software*, 2019.

[25] ——, "Graph algorithms via SuiteSparse:GraphBLAS: triangle counting and k-truss," in *IEEE HPEC*, 2018.