Received: 7 February 2018 | Revised: 13 April 2018 | Accepted: 1 May 2018
DOI: 10.1002/cpe.4691

WILEY

SPECIAL ISSUE PAPER

Fast multidimensional reduction and broadcast operations on GPU for machine learning

Doğa Dikbayır | **Enis Berk Çoban** | **İlker Kesen** | **Deniz Yuret** | **Didem Unat**

Computer Science and Engineering, Koç University, Istanbul, Turkey

Correspondence
Doğa Dikbayır, Computer Science and Engineering, Koç University, Istanbul, Turkey.
Email: ddikbayir17@ku.edu.tr

Didem Unat, Computer Science and Engineering, Koç University, 34450 Istanbul, Turkey.
Email: dunat@ku.edu.tr

Summary

Reduction and broadcast operations are commonly used in machine learning algorithms for different purposes. They widely appear in the calculation of the gradient values of a loss function, which are one of the core structures of neural networks. Both operations are implemented naively in many libraries usually for scalar reduction or broadcast; however, to our knowledge, there are no optimized multidimensional implementations available. This fact limits the performance of machine learning models requiring these operations to be performed on tensors. In this work, we address the problem and propose two new strategies that extend the existing implementations to perform on tensors. We introduce formal definitions of both operations using tensor notations, investigate their mathematical properties, and exploit these properties to provide an efficient solution for each. We implement our parallel strategies and test them on a CUDA enabled Tesla K40 m GPU accelerator. Our performant implementations achieve up to 75% of the peak device memory bandwidth on different tensor sizes and dimensions. Significant speedups against the implementations available in the Knet Deep Learning framework are also achieved for both operations.

KEYWORDS

broadcast, CUDA, GPU, machine learning, multidimensional arrays, reduction, tensor

1 | INTRODUCTION

Nowadays, machine learning algorithms successfully address different types of problems in various fields including image classification,[1] image recognition better than humans,[2] sentiment analysis, caption generation,[3] medical image analysis[4] and many more. Since machine learning algorithms consist of complex data structures processed in an iterative fashion, any performance optimizations play a crucial role to reduce their execution time. The size of data structures used in machine learning architectures grows drastically with the increase in the amount of available data. This growth makes the time efficiency of basic tensor operations critical. Two such operations are broadcast and reduction which appear repetitively in core machine learning kernels. They are used in the computation of the gradient values of a loss function of a deep neural network, which indicates the accuracy of the predictions made by the network. More specifically, the broadcast operation is used for summing the bias parameters with the weight parameters of a deep learning model. The bias values are projected over the weight tensors. Then, the reduction operation is used in the backpropagation algorithm, which computes the gradient values of a loss function, based on these parameters. For example, cross-entropy loss function calculates the weighted average of the prediction values computed by the neural network by the ground-truth values. This calculation is performed by reducing the output tensor of the network. In addition to this fundamental usage of these operations, every calculation that involves tensors and a basic associative mathematical function can use broadcast or reduction operations in its core. For instance, the recent method and the network on neural caption with visual attention proposed by Xu et al utilizes both broadcast and reduction operations.[5] In their proposed long short-term memory model, the calculation of the hidden states involves the broadcast of the output state on the memory state, by multiplication; learning stochastic and deterministic attentions in the network requires the weighted sum of the predicted attention value tensors through reduction operation.

In this paper, we propose efficient multidimensional broadcast and reduction operations on GPUs. In the literature, either these operations are naively implemented or optimized only for single dimension. Our implementations can work for any number of dimensions and focus on data reuse

to reduce global memory accesses and kernel launch overheads on the GPU. In both operations, CUDA threads are assigned to process one or more tensor elements. However, accessing input tensor elements or storing resulting tensor elements may require non-consecutive indexing. In broadcast operation, the sizes of the broadcast and resulting tensors are different, and in reduction, non-consecutive elements may be reduced by contiguous threads. Therefore, in both cases, efficient indexing to access memory locations may be required. To overcome these challenges, we propose efficient methods to calculate the correct indices for retrieving or storing the tensor elements by using stride values of the tensor.

In our proposed parallel tensor reduction method, we exploit the associativity property of reduction operation and minimize the necessary synchronization points in the method. Instead of launching a reduction kernel for each dimension, we merge them to reduce the overhead of kernel launch and and write-back of the temporary data to the global memory. Since the order of the elements to be added together has no importance because of the associativity, we reduce the tensor in smaller independent partitions, and thus minimize the synchronization penalty between threads. Our arrangement of thread blocks and threads also eliminates the need for inter-block synchronization. This method allows the tensor to be reduced in a fully parallel fashion, increasing performance.

In broadcast, the elements of the broadcast tensor are projected more than once; as a result, CUDA threads access the same elements in the memory many times. This results in poor memory bandwidth usage if not optimized. We propose a data reuse technique to address this problem. Each thread loads an element from the broadcast tensor once and reuses the same element to project over the input tensor. The size of the broadcast dimension determines the amount of data reuse. If the dimension is large, the reuse is even higher; however, small broadcast dimension causes underutilization of GPU cores since fewer threads are assigned to the tensor elements. To balance between data reuse and core utilization, we introduce a metric called sliding factor. This factor determines the reuse amount of the data by a thread.

In short, this paper makes the following contributions.

- We first mathematically define the reduction and broadcast operations on tensors and their properties. We then exploit these properties to implement an efficient method for each.
- We develop a fully parallel multidimensional reduction method for tensors and implement our method with a robust work division strategy to avoid multiple kernel launches that result in extra global synchronization points and eliminate the temporary storage.
- Our proposed implementation of broadcast operation for tensors avoids physical replication of the broadcast tensor in the memory and save both space and memory bandwidth. We also introduce the sliding factor parameter to balance the amount of the data reuse to utilize the stream multiprocessors.
- For both operations, we efficiently calculate the correct memory index values of the elements in the input tensor to be paired with the virtually replicated elements in the output tensor.
- Last but not least, in our evaluation, we compare the performance of our proposed implementations against Knet Deep Learning framework,[6] developed in Julia, an efficient programming language with just-in-time compiler and built in GPU functionalities.[7] We test our proposed method on reductions and broadcasts involving different dimensions and tensor sizes and achieve up to 75% of the peak device memory bandwidth and significant speedups over the existing Knet implementations.

2 | TENSOR OPERATIONS

2.1 | Tensor notations

First, we introduce the necessary terminology to describe the broadcast and reduction operations on tensors. We inherit the terminology to describe tensors from the work of Liu et al[8] in order to formally present our methodology. The terms and notations to be used in the rest of this paper are defined as follows.

Tensor: Tensor is an array that consist of multiple dimensions. We use bold lowercase letters to address vectors and bold capital letters for matrices. Higher-order tensors are represented by calligraphic letters such as $\mathcal{T}$.

Order and modes: The dimensions of a tensor are called *modes*. The number of modes in a tensor is referred as its *order*. We use the triple pipe operator to notate the order of a tensor. For example, $|||\mathbf{b}||| = 1$ for vector $\mathbf{b}$ and $|||\mathbf{M}||| = 2$ for matrix $\mathbf{M}$.

Indexing: Tensor elements are accessed using coordinate values, similar to accessing multidimensional arrays in many popular programming languages. Each coordinate value represents the position of the element at the corresponding mode. Element with coordinate values (i,j,k) of a third-order tensor $\mathcal{T}$, is accessed as $\mathcal{T}(i,j,k)$. If all the elements in a mode are accessed, the colon notation is used. For example, $\mathcal{T}(:,j,k)$ refers to all the elements in the first mode of tensor $\mathcal{T}$.

Fibers and slices: Tensors can be segmented into different partitions over one or more modes. One-dimensional partitions of a tensor are called *fibers* and two-dimensional segments are defined as *slices*. Figure 1 demonstrates the *yz* slices and *z* fibers of the third-order tensor $\mathcal{T}$.

Decomposition: Tensors can be decomposed into smaller partitions of lower order over $m_1, m_2, \ldots, m_n$ modes. The *decomposition* operation results in a decomposition list $\mathbf{D}$, consisting of $\mathbf{d}_i$. Each $\mathbf{d}_i$ is an nth-order tensor containing the original tensor's elements over the $m_1, m_2, \ldots, m_n$ modes. We define these as *hyperslices*. The decomposition of a tensor is expressed using a superscript consisting of the modes of decomposition on the tensor's identifier. For example, $\mathcal{T}^{y,z}$ would result in all of $\mathcal{T}$'s *yz* slices, as it is shown in Figure 1.

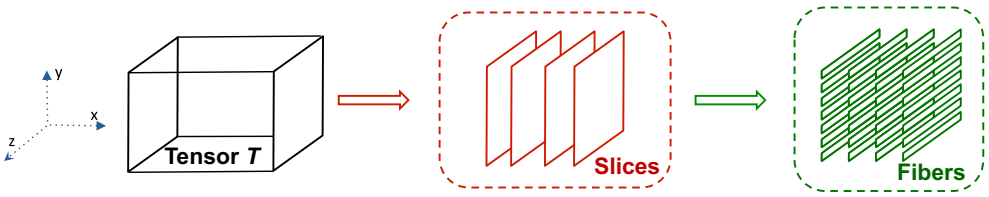

FIGURE 1 Example slices and fibers of tensor $\mathcal{T}$

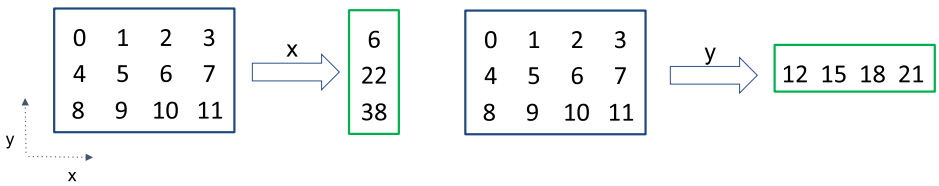

FIGURE 2 Reducing a matrix over its first mode (left) and over its second mode (right) by addition function

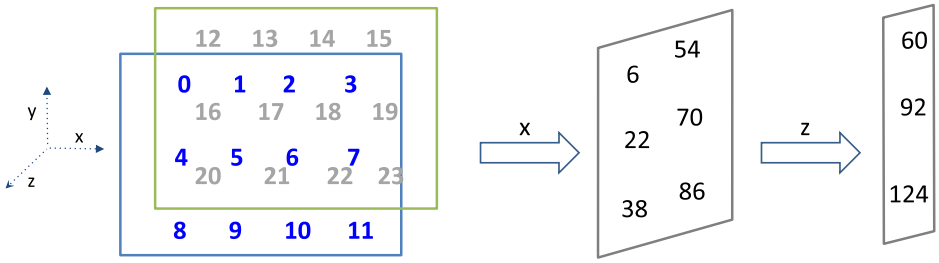

FIGURE 3 Reducing a 3rd-order tensor over its x and z modes

2.2 | Reduction operation

Reduction operation is the process of reducing one or more modes of a tensor by an associative function such as addition or maximum. The operation takes a tensor and the modes to reduce as input and produces a lower-order tensor as an output. The output tensor of the operation contains the reduced dimensions as its elements. Figure 2 demonstrates 1st-order reductions performed on matrix **M**, over its modes *x* and *y* separately by addition. If the fiber decomposition $\mathbf{M}^x$, is taken and f_i represents the *i*th fiber in $\mathbf{M}^x$, the *i*th element of the output vector is equal to $\sum_k f_i(k)$.

In this paper, we use the symbol $\biguplus$ to define the reduction operation. This should, however, not be interpreted as the disjoint union operation usually used in set theory. The reduction operation is formally defined as follows:

$$\biguplus_{M} \mathcal{A} = \mathcal{B}, \quad \text{where} \quad |||\mathcal{A}||| \geq n \quad , |M| = n \quad \text{and} \quad |||\mathcal{B}||| = |||\mathcal{A}||| - n. \tag{1}$$

The modes of reduction are contained in the *mode set M*, given under the reduction operator in Equation (1). If a mode set is not given, the input tensor is reduced over all of its modes to a scalar value. As it is seen in the equation, the difference between the input tensor's order and number of reduction modes, or the cardinality of the mode set, is equal to the order of the output tensor.

This operation is very common in scientific computing, and some programming models such as MPI and OpenMP provide primitives to perform the operation.[9,10] In these primitives, the input is usually reduced to a scalar value. However, in machine learning, the output of the operation can be multidimensional or the application may require a reduction over multiple modes of a tensor, making the reduction multidimensional. Neuman et al adressed this problem in his bachelor's thesis.[11] However, his implementation and evaluation is specific to Online Analytical Processing (OLAP). Moreover, many architectural and software improvements have been introduced to GPU devices since then, enabling more efficient solutions to be proposed. Therefore, in this paper, we focus more on reductions over multiple modes and reductions that result in a multidimensional output. Figure 3 illustrates both of these cases in a single reduction example. The figure shows sequentially how a 2nd-order reduction over modes *x* and *z*, $\biguplus_{\{x,z\}}$ is performed on a 3rd-order tensor.

2.3 | Broadcast operation

In broadcast operation, one input tensor is projected over the other by using an associative function (eg, addition and multiplication). In the formal definition of the broadcast operation, we use the symbol $\coprod$ to notate broadcast as in Equation (2):

$$\mathcal{B} \coprod_{M} \mathcal{A} = \mathcal{C}. \tag{2}$$

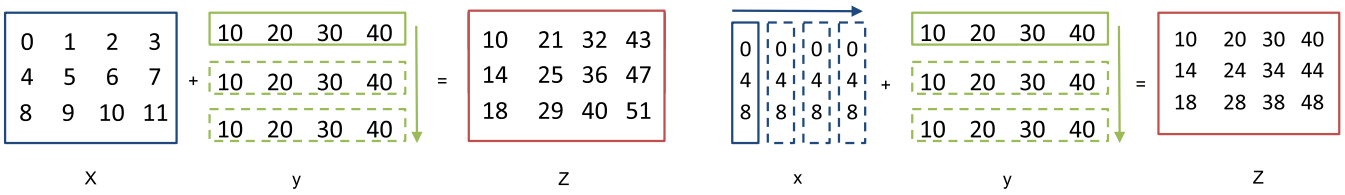

FIGURE 4 Broadcasting vector ***y*** on matrix ***X*** by addition operation (left). Broadcasting a row over a column vector by addition operation (right)

Different than the reduction operation, broadcast operation takes two tensors as arguments, ie, the $\mathcal{B}$ tensor is broadcasted on the $\mathcal{A}$ tensor over modes in the mode set M and the output tensor $\mathcal{C}$ is produced. We refer to $\mathcal{B}$ as the broadcast tensor and $\mathcal{A}$ as the input tensor. Output tensor always has the same or larger order than both of the input tensors. In machine learning, usually, tensor broadcast is utilized rather than scalar broadcast. Figure 4 demonstrates a broadcast example with a vector projected over a matrix, by addition. In this example, the vector ***y*** is replicated to match the order of matrix ***X***, and the elements are added to obtain matrix ***Z***. Both of the tensors, also multiple modes from each tensor, can be broadcast. Figure 4 on the right shows the addition of a row and a column vector. It demonstrates the case where both of the vectors need to be scaled in their respective missing modes.

3 | IMPLEMENTATION

3.1 | Reduction implementation

An nth order reduction operation on a tensor can be decomposed into n different r_ith order reductions where $\sum_{i=0} r_i = n$. For example, an nth order reduction can be written as n different first-order reductions:

$$\biguplus_{M} \mathcal{A} = \biguplus_{\{m_1\}} \biguplus_{\{m_2\}} \dots \biguplus_{\{m_n\}} \mathcal{A} \quad \text{where} \quad M = \{m_1, m_2, \dots, m_n\}. \tag{3}$$

In addition to the property shown in Equation (3), elements in a tensor are processed by an associative function, where the order of processing is not important. Therefore, if we want to reduce a higher-order tensor over n different modes, the processing order of the elements in these n modes does not affect the result of the operation. Then, the reduction operation in Equation (3) could also be expressed as follows, where \ represents set difference:

$$\biguplus_{M} \mathcal{A} = \biguplus_{M\setminus\{m_i\}} \biguplus_{\{m_i\}} \mathcal{A}, \quad \text{where} \quad 1 \le i \le n. \tag{4}$$

Figure 5 demonstrates $\biguplus_{\{x,y\}} A$, performed as $\biguplus_{\{y\}} \biguplus_{\{x\}} A$ and $\biguplus_{\{x\}} \biguplus_{\{y\}} A$, respectively. As it can be seen from the figure, the order of the reduction sequence does not affect the result.

The associativity property of reduction gives the freedom to perform the reduction in any order. However, each mode reduction in Equation (4) can start upon the completion of the previous mode reduction and synchronization between successive reductions is necessary. Unfortunately, CUDA does not support a global synchronization method that provides inter-thread block synchronization in a single kernel. This results in separate kernel launches for each mode reduction. This repetitive kernel launch approach causes poor utilization of memory bandwidth because all the threads are globally synchronized at the end of each kernel execution. In addition, kernel launch overheads are added to the overall execution time. To overcome these performance issues, we exploit the reduction operation's associativity property and combine the elements over each reduction mode in nth-order tensors. To obtain these hyperslices, we use the tensor decomposition described in Section 2. We first obtain the decomposition $D = \mathcal{A}^{r_1, r_2, \dots, r_n}$, consisting of hyperslices d_i. Then, each element b_i in the output tensor $\mathcal{B}$ is

$$b_i = \biguplus d_i. \tag{5}$$

As Equation (5) suggests, each hyperslice is reduced to a scalar value, and thus does not have any dependency on the reduction modes. A synchronization is not needed anymore in each mode reduction because all of the elements in a hyperslice are simply added together. Moreover, hyperslices d_i are independent from each other, since the values of elements in $\mathcal{B}$ do not depend on each other. As a result, global synchronization is not needed either. This approach provides two benefits, ie, temporary values are not written back to memory and overhead of multiple kernel launches is omitted because we can now perform the reduction with a single kernel launch. In the next sections, we describe how single kernel reduction operation is implemented in detail.

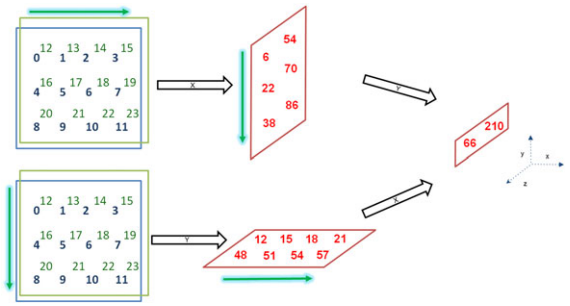

FIGURE 5 A tensor is reduced over its first and second modes in two alternative orderings

3.1.1 | Kernel configuration

One of the challenges in obtaining high performance on GPUs is to utilize CUDA resources properly. The number of threads per CUDA block and the total number of available blocks in a GPU are limited. The work division of the thread blocks must be carefully organized in order to achieve high performance. After giving a detailed formal explanation of our method in the previous section, we now explain our kernel configuration strategy on a 2nd-order reduction that results in a multidimensional output for the sake of simplicity. In a 2nd-order reduction operation, the decomposition of the tensor over the reduction modes would result in slices. There are several ways to assign these slices to CUDA thread blocks. These can be categorized based on the number of slices per block.

- # of slices per block > 1: requires within a block synchronization since a block can compute the next slice once it finishes the current slice.
- # of slices per block $= 1$: incurs no synchronization problem.
- # of slices per block < 1: leads to global synchronization problem since a slice is computed by more than one thread block.

The optimal way to assign thread blocks is one block per slice as it does not require any synchronization between thread blocks. However, two problems may arise with the one-slice-per-block assignment. First, when reduction slices are very small in size but large in total count, assigning multiple slices to a single CUDA thread block might be required. In this case, the number of maximum thread blocks that CUDA supports is fewer than the number of independent reduction slices in the decomposition. We overcome this issue with *grid stride loops*, where a grid stride is the number of thread blocks in the execution. In our grid stride loop, each thread block is assigned a single slice and the blocks are reassigned to the remaining slices once they finish reducing a slice. The next slice that a thread block computes is a grid stride away from the current slice.

The second problem is related to the slice size being too big. A CUDA block can contain 1024 threads at maximum. If the size of the reduction slice is too big, a single block may be insufficient to reduce a slice. Assigning multiple thread blocks to a single slice would require synchronization between thread blocks, thus multiple kernel launches. To avoid this, we increase the number of elements to be computed by a single thread in a block. For example, if we have 1024 threads in a CUDA block and assign four elements to each thread in the block, we can reduce slices containing up to 4096 elements, without assigning an additional block to the same slice. The efficiency will naturally decrease in case of corner cases where we might have to assign too many elements to each thread, but the parallelism will not be affected.

Lastly, if the tensor is skinny and tall meaning that the slices are big but there are only few of them, our implementation would only create a small number of thread blocks, which may not be sufficient to occupy all the streaming multiprocessors. We do not have a good solution for this yet. Either one can divide the slice among multiple thread blocks, which introduce synchronization problem, or launch parallel kernels on the device. We leave this for future work.

3.1.2 | Virtual coordinate calculation

In this section, we explain how we assign the CUDA threads to the correct tensor elements. As described in Section 2, in our method, a reduction over multiple modes is performed on the hyperslices of the input tensor's decomposition. The decomposition results in a virtual re-arrangement of the original tensor in a different multidimensional space that has the reduction modes as its dimensions. We do not rearrange the elements in the memory physically; therefore, we cannot use the original tensor's coordinate values to access them. Instead, we need to transform the linear indices to virtual coordinates.

If there are n modes to reduce the original tensor over, then the reduction space is $(n + 1)$-dimensional. We first derive an $n + 1$ dimensional coordinate system for the reduction space. Then, the first n coordinates determine the target element's position in the nth order hyperslice, while the last coordinate indicates which hyperslice the target element belongs to. We can easily access an element once its reduction space coordinates are known. However, in the actual parallel implementation, we are only given the one-dimensional global index of each consecutive thread ID in a CUDA kernel. Using this global index value, we need to calculate the coordinate values in the $(n + 1)$-dimensional reduction space in order to assign threads to the correct tensor elements. Algorithm 1 demonstrates the steps required to derive the coordinate values of the target element. The aim of the algorithm is to determine to which positions a linear index corresponds in a multidimensional space. The algorithm takes three inputs,

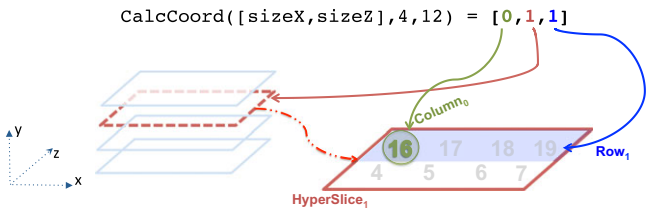

FIGURE 6 Virtual coordinates of thread 12 visualized on the tensor used in the example coordinate calculation

ie, *ModeSizes*, *HyperSliceCount*, and *ThreadID*, and produces a single output *Coordinates*. *ModeSizes* is the list containing the size of each reduction mode; *HyperSliceCount* is the number of hyperslices in the decomposition, which is also equivalent to the size of the last dimension in the reduction space; and *ThreadID*, which is the one-dimensional global index of the CUDA thread. The virtual coordinate of the target elements is then stored in *Coordinates*.

For the sake of simplicity we explain the algorithm using an example. Let's suppose that we are reducing the tensor $\mathcal{T}$ from Figure 3. As it is shown in the figure, we are reducing a 3rd-order tensor over *x* and *z* modes. The decomposition will result in a reduction space containing three *xz* slices, each containing two fibers of size four. Thus,

$$ModeSizes = [4, 2] \quad and \quad HyperSliceCount = 3.$$

Now, let us find the virtual coordinate values for the thread with *ThreadID* equal to 12. First, we calculate the linear offset of the thread in a fiber as it is shown in line 1 in the algorithm, ie, $12\%4 = 0$. Then, we initialize *PrevModesArea* as *ModeSizes*[0], which is 4 in this case. In line 4, two operations are combined. For the first iteration, the division operator finds the number of fibers covered by *ThreadID*, ie, $12/4 = 3$. Then, the modulus operator calculates the offset of the linear index in the second mode, ie, $3\%2 = 1$. In the last line of the loop, the area of the previous modes is updated with the second mode's size. This procedure is repeated for all the remaining reduction modes, and all the coordinates within the hyperslice are calculated. In this example, there are only two reduction modes so the hyperslice coordinates are [0, 1]. Finally, we calculate to which hyperslice the thread must be assigned by finding how many hyperslices the linear index covers, ie, $12/8 = 1$. For our example, the thread having 12 as its global index value is assigned to the element with virtual coordinates [0, 1, 1], which is illustrated in Figure 6. This means that the thread will process the first element in the second row of the second hyperslice in the decomposition. After the process described above, we calculate the memory index of the element using the virtual coordinate values and the memory stride values, which is further explained in Section 3.2.1 as it is also used in broadcast implementation.

Algorithm 1 CalcCoord: Calculate Virtual Coordinates

Input : *ModeSizes*: *Sizes of reduction modes*
HyperSliceCount: *Number of hyperslices in decomposition*
ThreadID: *One dimensional global index of the thread*
Output: *Coordinates*: *Virtual coordinates*

```
1 Coordinates[0] = ThreadID % ModeSizes[0]
2 PrevModesArea = ModeSizes[0]
3 for i = 1 .. ModeSizes.length − 1 do
4 |   Coordinates[i] = (ThreadID/PrevModeArea)%ModeSizes[i]
5 |   PrevModesArea = PrevModesArea * ModeSizes[i]
6 end
7 Coordinates[ModeSizes.length] = ThreadID/PrevModesArea
```

3.1.3 | Parallel reduction algorithm

In order to achieve good performance, it is necessary to optimize the reduction performed by each thread block. Harris proposed a detailed strategy for efficient parallel reduction to a scalar value.[12] In his method, each thread block processes a partition of the input tensor, and the partial sums are then processed by a separate kernel or by CPU to obtain the scalar result. To calculate the thread block sums, they leveraged the on-chip memory, called *shared memory*, which is accessible by all the CUDA threads in the same thread block. Each thread in a block sums the elements assigned to it and then writes the result to a shared memory slot. Then, after the threads in the block are synchronized, the sums in the block's shared memory are added together to obtain the block sum. This approach lets the program to avoid accessing the device memory, which is a costly operation. However, in this method, block synchronization is needed before combining the partial sum in shared memory. This implementation can be further improved by utilizing the properties of structures called *warps*. Warps contain a certain number of threads that are all scheduled together at the same time, and thus threads in the same warp do not need to be synchronized. In addition to this, NVIDIA introduced a new instruction called *shuffle* with the Kepler architecture that allows a thread to read data from the local register of another thread in the same warp.[13] This means that threads in the

same warp can communicate with each other without any shared memory access or synchronization calls. This also implies that the synchronization calls and shared memory accesses are reduced by a factor of warp size, which is 32 for Kepler.

We adopt the block reduction method proposed by Luitijens[14] that exploits the warp properties and leverages the shuffle instruction. In Luitijens' block reduction, each warp in the block performs a partial reduction using the shuffle instruction. Then, the first thread in each warp writes the partial sum into a shared memory slot and waits for the other warps to perform their partial sums. Finally, all the partial sums are summed together by the first warp with a single warp reduction.

Algorithm 2 Parallel Reduction Algorithm

Input : $\mathcal{A}$:Input tensor
MemStrides: Memory stride values
ModeSizes: The size of each reduction mode
HyperSliceCount: The number of hyperslices in decomposition
NoEls: Number of elements processed by each thread
Output: $\mathcal{B}$: Output tensor

```
Coordinates = CalcCoord(ModeSizes, HyperSliceCount, ThreadID)   ▷ calls Algorithm 1 to compute virtual coordinates
MemIndex = Coordinates .* MemStrides                            ▷ calculates physical memory location using strides and coordinates
NumSlices = CalcSliceCount(BlockID)                             ▷ calculates number of slices to be computed by this thread block for grid stide loop
for Slice = 0 .. NumSlices -1 do
    for Iter = 0 .. NoEls - 1 do
        Sum = Sum + A[MemIndex + Iter*Skip]                     ▷ Thread sums its assigned tensor elements, uses skip to find next element location
    end
    Sum = BlockReduce(Sum)                                      ▷ Block sum using warps and shuffle instruction
    if first thread then
        B[ Slice*GridSize + BlockID] = Sum
    end
end
```

Now that we have an efficient block reduction method, we can construct the overall reduction kernel by incorporating the strategies from the previous sections along with the block reduction method explained in this section. Algorithm 2 demonstrates our proposed parallel reduction method's pseudo code. Note that many details are omitted to clarify the code. We begin by calculating the virtual coordinate values and, then, take their dot product with the memory stride values to find the memory index of the element. Then, the thread sums the elements assigned to it and passes it to the *BlockReduce* function, which implements the block-level reduction with warps and shuffle instructions as explained earlier in this section. After the block sum is calculated, we write the result to the output tensor in global memory. If the thread block is assigned to more than one slice to compute, it advances to the next slice using the grid stride.

3.2 | Broadcast implementation

In order to perform the broadcast operation, the broadcast tensor needs to be replicated in the memory to match the input tensor's broadcast mode. For example, if a vector with N elements is going to be broadcast over a matrix of size $N * M$, a total of $(N - 1) * M$ elements must be stored in memory. This requirement is costly in terms of storage space. In addition, the broadcast operation has very low arithmetic intensity as it performs only a single arithmetic operation for every two memory accesses, making it a memory-bound operation. Because of these reasons, accessing the same tensor element multiple times wastes memory bandwidth as well as memory capacity. We choose not to replicate the data in memory to save memory capacity and bandwidth; then, the challenge is to find the corresponding indices of a tensor because the corresponding elements do not have the same index value. We propose a representation for index values of elements to facilitate the calculation of the corresponding indices, which is discussed in the next section.

3.2.1 | Index calculation

One can calculate the global index of an element from its coordinates in each mode if the tensor's data is contiguously allocated in memory. For example, the matrix X in Figure 7 has an element at coordinate (0,1) with value of 4, which has a global index of 4 assuming first address starts at 0. However, elements in the y vector other than the first row do not physically exist in memory as we chose not to duplicate the data. During broadcast, accesses to those indices have to be translated into physical memory locations (global indices) even though the virtual coordinates are the same.

The global index value of an element in a tensor can be calculated using its coordinates and the tensor's stride values.[15] A stride value is the number of jumps required in memory to access the next element in the tensor's corresponding mode.[16] Therefore, if i, j, and k represent the coordinates of an element in a 3rd-order tensor $\mathcal{A}$, the stride value of the third mode is equivalent to the number of jumps in memory needed to get from $\mathcal{A}(i,j,k)$ to $\mathcal{A}(i,j,k+1)$. The coordinate of the element for each mode is stored in the virtual coordinate list and the stride values for the

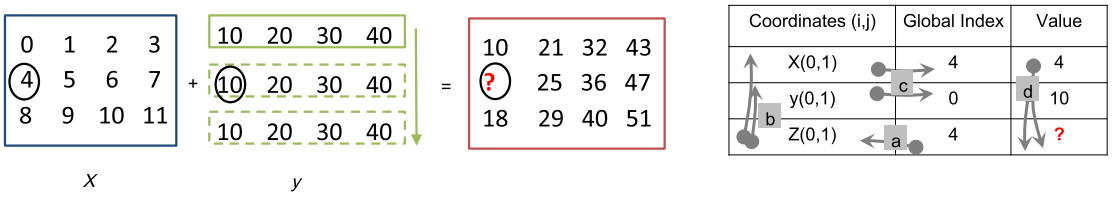

FIGURE 7 Coordinates and indices in tensors. Necessary index calculation steps for broadcast operation are shown

corresponding modes are also known. Then, the global memory index is as follows:

$$\text{Index} = \text{stridevalues}[\,] . * \text{coordinates}[\,].$$

When the dot product is taken as above, each stride value is multiplied with the coordinate value of the element on the corresponding mode. Similar to reduction operation, this representation of indices makes the broadcast operation working on the non-contiguous tensors easier. We set the stride value of the broadcasted modes to 0. As we increment the coordinates in that mode, ie, when we multiply with stride values, the broadcasted modes do not have an effect on index calculation.

In CUDA, we assign one CUDA thread to at least one element to compute in the output tensor, which is always kept continuously in device memory. However, the broadcast tensor is not accessed contiguously in memory. An increment on a coordinate value would mean jumps in memory addresses since the broadcast tensor elements in the desired coordinate may not exist as in the case of Figure 7. From output tensor's global index, we can calculate the coordinates of the output tensor's elements which shares the same coordinates of the input tensor's and broadcast tensor's elements. The corresponding memory locations of broadcast elements are then calculated from the stride and coordinate values for each input using the formula above. Figure 7 explains this procedure with an example. A thread is assigned to compute the tensor element in *Z* at the global index of 4. In step (a), we convert this global index into coordinate values. Step (b) shows that the coordinate value of output tensor is shared between the broadcast tensor and input tensor. In step (c), from coordinates, we compute the global indices of vector *y* and matrix *X*. Then, in the last step, we retrieve the values at those global indices and apply the broadcast function to obtain the value in the output tensor.

3.2.2 | Sliding factor optimization

The algorithm described in the previous section supports tensor broadcast with any number of modes. In this part, we present a more specialized broadcast implementation that introduces a sliding factor for reuse, but it is restricted to the matrix-vector broadcast, which is one of most common cases in machine learning. As mentioned earlier, broadcast operation is memory bandwidth bound and any optimization to improve its memory accesses would increase its performance. Threads at different thread blocks would have to load the vector elements again because there is no sharing between them. To enable data reuse, we let a single thread to apply the same broadcasted value to multiple rows or columns of the input matrix.

A thread loads an element from the broadcast vector once in device memory into its register and slides down this value through from beginning of matrix to the end of it. If the size of vector is *N*, then a thread uses the loaded element for *M* times over a matrix with dimensions of *NxM* if a one-dimensional thread block is used. For multidimensional thread blocks, *M* would be divided by the size of the thread block in that dimension. We refer to this optimization as *thread sliding*, and the amount of times a thread slides as *sliding factor*.

For the broadcast operation, we explore the usage of shared memory for the thread sliding optimization. In the shared memory implementation, all threads in the first row of a 2D thread block load one element from vector *y* to shared memory. Threads in the same column of the same thread block can use the values loaded into the shared memory by the threads in the first row after performing a block-level synchronization. Then, similar to sliding factor implementation, thread blocks start from the top of the matrix and slide over the matrix until the end of it, adding numbers from the vector.

Figures 8A and 8B show the thread sliding optimization using registers for two thread blocks with 2×4 threads in each for a matrix-vector broadcast. Each thread is assigned to one element from the input vector but responsible for calculating 4 elements of resulting matrix. A 2×4 thread block shown in the figure slides over the matrix. Since there is no sharing between threads, there is no need for block-level synchronization. Figure 8C shows the shared memory version of the thread sliding. Four threads in the first row of each thread block would load data from the input vector to shared memory, and the other 12 threads in the same block would use the loaded values after they synchronize at block level.

4 | EVALUATION

In this section, we discuss the experiments we have performed to measure the performance of our implementations against Knet and how the performance is affected by different parameters. In our experiments, we measure the effective bandwidth rate achieved by our implementations because both reduction and broadcast operations are known to be memory bandwidth limited and have very low arithmetic intensity.[12] We calculate the effective bandwidth by dividing required data movement by the elapsed time. Here, the required data movement is the minimum amount

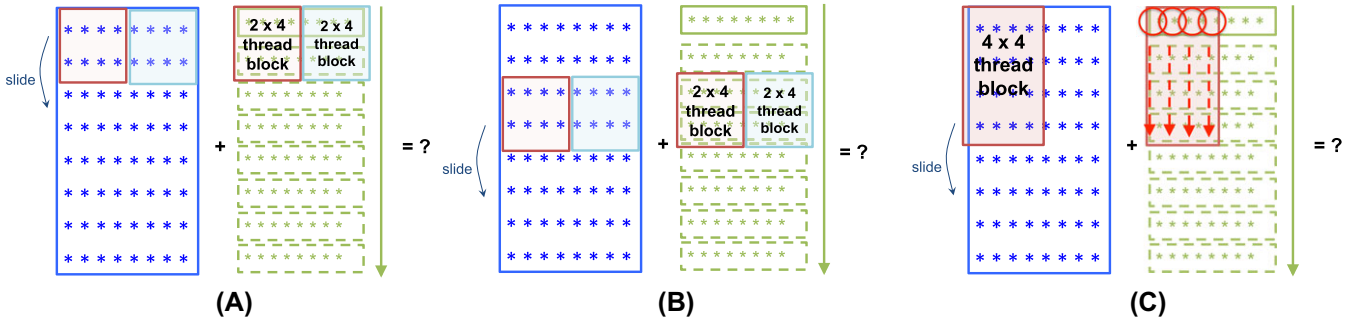

FIGURE 8 (A) and (B) show thread sliding optimization for 2×4 thread blocks with a sliding factor of 4. In (C), shared memory version of thread sliding is shown for 4×4 thread blocks with a sliding factor of 2

of read and write operations required to perform broadcast or reduction. Since the required data movement is the same for both Knet and our implementations, essentially, the effective bandwidth is proportional to the inverse of elapsed time.

We conduct our measurements with CUDA 9.0 on a Tesla K40 m GPU accelerator. The device has 12 GB of GDDR5 on-board memory, supports PCI Express 3.0, and its peak memory bandwidth rate is 288 Gb/s.[17] The NVIDIA bandwidth test in the CUDA toolkit measures 215 GB/s on device-to-device pinned memory. The CPU on the host is a 2x Intel Xeon E5 2695 v2 with 256 GB of memory. All the experiments are done with double precision floating point arithmetic.

4.1 | Reduction performance

For reduction, we divided our experimental setup into two parts. First, we observe how the structure of the input tensor affects the performance, and in the second part, we test our method with different reduction scenarios and compare the performance to Knet's reduction method. In the first part, we change the width and length of the input tensor to observe in which cases our method achieves the best performance. We perform a second-order reduction on a 4th-order tensor, so the decomposition results in slices. We first test our algorithm on a tall and skinny 4th-order tensor, which has high number of slices in its decomposition but only few elements in each slice. Then, we increase the number of elements in each slice while keeping the total number of elements in the tensor fixed to shorten and fatten the tensor.

Figure 9 shows the effective bandwidth rate changing with the number of elements in each slice in the decomposition. The number of elements in the input tensor is fixed around ~64 million. The number of elements in a slice starts at 8×16 and goes up to 512×512. The slice count decreases in the same range since the total number of elements in the tensor is fixed. We observe that, for skinny and tall tensors the performance is low, only achieving about 20% of the peak device bandwidth. This is mainly because each thread block has few number of threads and processes few number of elements since each block is assigned to a single slice, which results in poor usage of stream multiprocessors. However, we observe almost no performance degradation with the increase in slice size (decrease in slice count) since there are enough slices for all thread blocks. As expected, this experiment suggests that our method works best when the slices in the decomposition are fat enough to keep the stream multiprocessors occupied.

In the second part of our experiments, we measure how the order of the reduction and choice of reduction modes affect the performance. To measure these effects, we select the first modes and last modes in a tensor; the former represents the continuous memory accesses in the fast changing dimensions and the latter represents the largest strided memory accesses in the slowest changing dimensions. As baseline, we use Knet's reduction implementation. We keep the minimum size of a hyperslice at 4096 elements to ensure that the stream multiprocessors are not underutilized. We increase the total number of elements in the input tensor from 2*M* up to 128*M*. We measure reduction performance on tensors $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, where

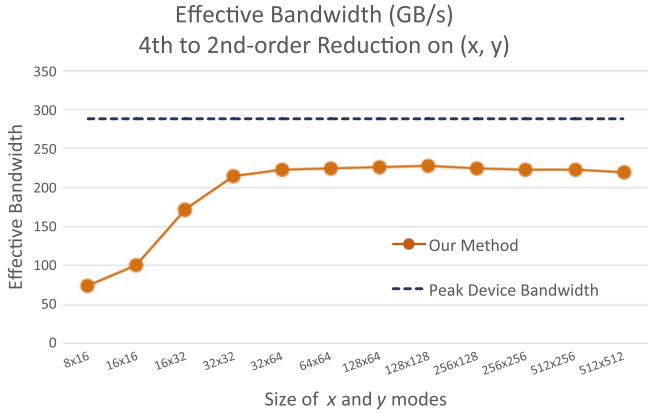

FIGURE 9 Performance of our method with increasing # elements in each slice in the decomposition

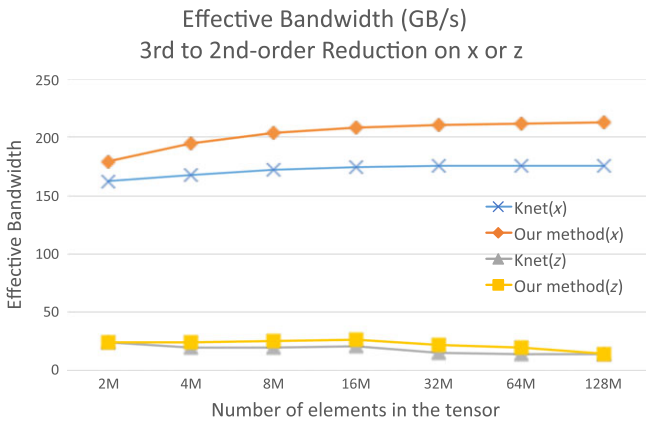

FIGURE 10 Comparing our method against Knet with different tensor sizes, reducing third-order tensor over its *x* or *z* modes

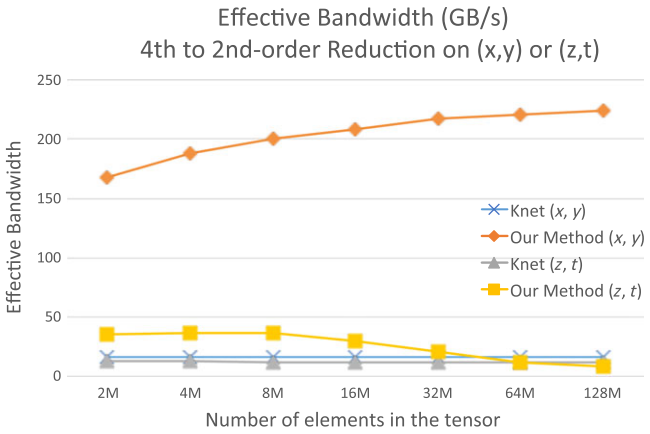

FIGURE 11 Reducing fourth-order tensor over its (x, y) or (z, t) modes

we reduce 1, 2, and then 3 modes, respectively. $\mathcal{A}$ is a 3rd-order tensor and we perform $\biguplus_{\{x\}} \mathcal{A}$ and $\biguplus_{\{z\}} \mathcal{A}$. Then, we perform $\biguplus_{\{x,y\}} \mathcal{B}$ and $\biguplus_{\{y,z\}} \mathcal{B}$ on the 4th-order tensor $\mathcal{B}$, and finally, we perform $\biguplus_{\{x,y,z\}} \mathcal{C}$ and $\biguplus_{\{z,t,k\}} \mathcal{C}$ on the 5th-order tensor $\mathcal{C}$.

Figure 10 shows the effective bandwidth of reductions on tensor $\mathcal{A}$. We provide very similar performance to Knet when reduction is performed on a single mode because there is only a single mode to reduce and not much additional benefit of our approach that combines multiple modes into a single kernel.

The performance drastically decreases for both our method and Knet when the tensor is reduced over its *z* mode. This is expected because elements over the *z* mode of the tensor are far from each other in the memory thus require strided accesses by the threads.

Figures 11 and 12 show the reduction on tensor $\mathcal{B}$ with 2 mode reduction and tensor $\mathcal{C}$ with 3 mode reduction, respectively. First of all, similar to single mode reduction, we clearly observe that strided memory accesses affect the performance of all the implementations negatively. However, different from Figure 10, we achieve a significant speedup over Knet when the tensors are reduced over their first modes. This indicates that, while an iterative implementation is critically affected by the number of modes in the reduction, our approach enables to perform multidimensional reduction operations without losing performance. Our implementation is mainly bounded by strided accesses to the memory and not affected by the number of reduction modes. Even though it is not shown in the figure for clarification, reduction modes in between the fastest and slowest changing dimensions result in an effective bandwidth somewhere in highest and lowest bandwidth rates.

4.2 | Broadcast performance

In this section, we compare the performance of the broadcast algorithm for three different versions, ie, *sliding factor*, *sliding factor with shared memory*, and *generalized version*. As opposed to two other versions, the generalized version supports any number of modes. As discussed in Section 3.2.2, we proposed two optimizations over generalized broadcast algorithm, specific to vector broadcast. First is the sliding factor implementation using only registers, and then, in the second, we added shared memory support to sliding factor algorithm to decrease memory accesses even further. We compare the performance of the two optimizations specific to the vector broadcast with our generalized kernel and also use the Knet's broadcast as baseline. It is important to note that Knet only supports vector broadcast.

In our performance tests, we first broadcast a vector over a matrix, that is $\mathbf{b} \coprod_{x} \mathbf{A}$. The length of the vector determines how many elements will be broadcasted. The length of second mode of the matrix correlates with how many times we can use a broadcast element from the vector.

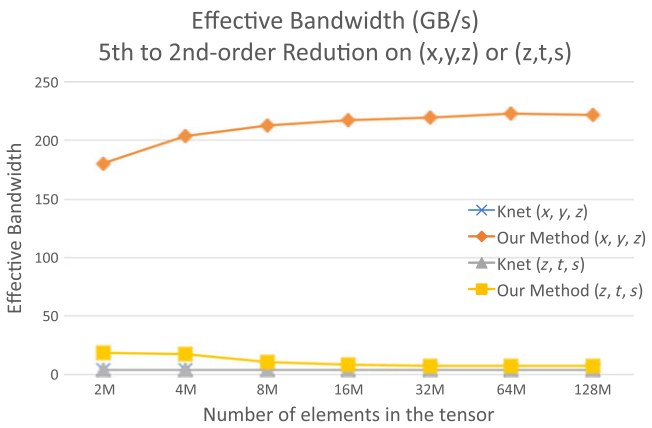

FIGURE 12 Reducing fifth-order tensor over its (x, y, z) or (z, t, s) modes

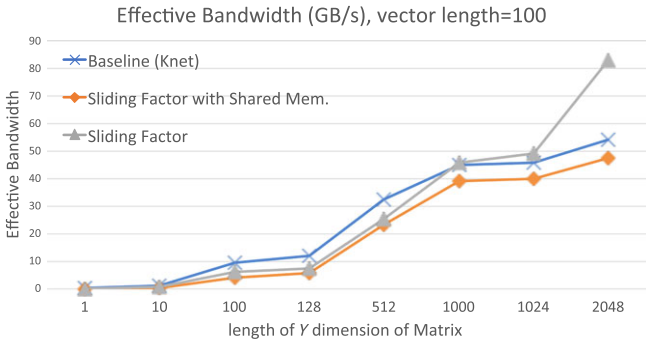

FIGURE 13 Broadcast performance of Knet, sliding factor, and sliding factor with shared memory optimizations. Vector size is 100

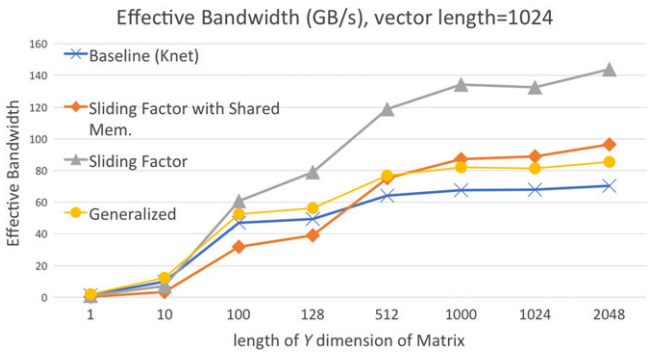

FIGURE 14 Broadcast performance of Knet, generalized kernel, sliding factor, sliding factor with shared memory optimizations. Vector size is 1024

In Figure 13, we compare two optimized versions of the algorithm and existing implementation from Knet for vector broadcast involving a small vector in size. In Figure 14, we increase the vector size and compare our optimization's performance with Knet's implementation and the generalized implementation. *Y*-axis shows the effective bandwidth (GB/s) achieved in the broadcast operation, so higher is better. *X*-axis shows the length of second dimension in the matrix. In both figures, for small number of elements in vector, Knet is slightly better because there is not enough data reuse for optimization, and the overhead of optimization slows down the program. In fact, the sliding factor with shared memory version is always slower than the baseline for small vectors. The sliding factor implementation with no shared memory performs better than the baseline when the size of the second dimension reaches about 1000. At that point, data reuse can be 4 to 6 times as the thread block count; as a result, the algorithm keeps all stream multiprocessors (SMs) busy on the GPU. Otherwise, the number of thread blocks might be insufficient to occupy all the SMs, and performance of our algorithm suffers due to idle cores. The sliding factor implementation with no shared memory is also better than the version that uses shared memory because there is no sharing between threads; thus, there is no need for block-level synchronization.

In Figure 14 for a large vector, the algorithm with shared memory performs better than baseline algorithm but still performs worse than the sliding factor optimization without shared memory. This is mainly because data sharing between threads in a thread block is not beneficial enough to cover the cost of synchronization between threads. Sliding factor algorithm that uses registers to store the loaded elements require no synchronization even though it performs more loads compared to the shared memory version. This version performs much better than the baseline, reaching over

140 GB/s and getting closer to the sustained memory bandwidth of the device memory while the Knet version only achieves about the half of that bandwidth. We observe from Figure 14 that our generalized kernel is better than Knet even though it does not perform optimizations specific to vector broadcast. As expected, it performs worse than the optimized versions for vector broadcast.

5 | CONCLUSION

In this paper, we have focused on two commonly used deep learning operations, namely, broadcast and reduction, and optimized their implementations on GPUs. We present tensor notations and operations in order to build a clear terminology and mathematically define reduction and broadcast operations on tensors. We investigate the properties and structure of these operations in order to propose efficient solutions. For reduction, we exploit the associativity property to design a fully parallel strategy that avoids multiple kernel launches that result in temporary storage and overhead. For broadcast operation, we make use of data reuse techniques in order to eliminate unnecessary intermediate device memory accesses and storage and introduce the sliding factor to avoid underutilization of GPU cores. We test our implementations on Tesla K40 m GPU accelerator and compare their performance to existing implementations in Knet. We achieve high performance with our implementation measuring up to 75% of the theoretical device memory bandwidth and up to 56x speedups over Knet. Nevertheless, we also observe that reductions performed on the last modes of a tensor requiring strided accesses in memory, decrease the performance drastically for all of the cases. For broadcast, we extend Knet's capability of performing the operation only for vectors to tensors containing any number of modes. We also optimize the widely used vector broadcast and achieve up to 2x speedup over Knet's existing implementation. Future work will focus on improving the performance of other common machine learning operations and optimize them on multiple GPUs.

ORCID


Doğa Dikbayır http://orcid.org/0000-0003-4673-9612

REFERENCES

1. Krizhevsky A, Sutskever I, Hinton GE. ImageNet classification with deep convolutional neural networks. In: Proceedings of Neural Information Processing Systems 2012 (NIPS 2012); 2012; Lake Tahoe, NV.
2. He K, Zhang X, Ren S, Sun J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. 2015. CoRR abs/1502.01852.
3. Karpathy A, Fei-Fei L. Deep visual-semantic alignments for generating image descriptions. In: Proceedings of the 2015 IEEE Conference on Computer Vision and Pattern Recognition; 2015; Boston, MA.
4. Zhou SK, Greenspan H, Shen D. *Deep Learning for Medical Image Analysis*. 2nd ed. London, UK: Academic Press; 2017.
5. Xu K, Ba J, Kiros R, et al. Show, attend and tell: Neural image caption generation with visual attention. 2015. CoRR abs/1502.03044.
6. Yuret D. Knet: beginning deep learning with 100 lines of Julia. Paper presented at: Machine Learning Systems Workshop at NIPS 2016; 2016; Long Beach, CA.
7. Bezanzon J, Karpinski S, Shah V, Edelman A. Julia: A fast dynamic language for technical computing; 2012. arXiv preprint arXiv:1209.5145.
8. Liu B, Wen C, Sarwate AD, Dehnavi MM. A unified optimization approach for sparse tensor operations on GPUs. Paper presented at: 2017 IEEE International Conference on Cluster Computing (CLUSTER 2017); 2017; Honolulu, HI.
9. Makpaisit P, Ichikawa K, Uthayopas P, Date S, Takahashi K, Khureltulga D. MPI_Reduce algorithm for OpenFlow-enabled network. Paper presented at: 2015 15th International Symposium on Communications and Information Technologies (ISCIT); 2015; Nara, Japan.
10. Chapman B, Jost G, van der Pas R. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. London, UK: The MIT Press; 2007.
11. Neumann AB. Parallel reduction of multidimensional arrays for supporting online analytical processing (OLAP) on a graphics processing unit (GPU); 2008.
12. Harris M. Optimizing CUDA. Paper presented at: 2007 Supercomputing Conference; 2007; Reno, NV.
13. Warp shuffle functions. http://docs.nvidia.com/cuda/cuda-c-programming-guide/#warp-shuffle-functions. Published 2017. Accessed February 4, 2018.
14. Luitijens J. Faster parallel reductions on Kepler. https://devblogs.nvidia.com/faster-parallel-reductions-kepler. Published 2014. Accessed February 4, 2018.
15. Wilson G, Oram A. *Beautiful Code: Leading Programmers Explain How They Think*. Sebastopol, CA: O'Reilly Media; 2007.
16. Wikipedia. Stride of an array. Published 2017. Accessed February, 12 2017.
17. NVIDIA. NVIDIA K40 m active board specifications. https://www.nvidia.com/content/PDF/kepler/Tesla-K40-Active-Board-Spec-BD-06949-001_v03.pdf. Accessed January 28, 2018.

How to cite this article: Dikbayır D, Çoban EB, Kesen İ, Yuret D, Unat D. Fast multidimensional reduction and broadcast operations on GPU for machine learning. *Concurrency Computat Pract Exper*. 2018;e4691. https://doi.org/10.1002/cpe.4691