Multidimensional Broadcast Operation on the GPU

Enis Berk Çoban
Computer Science and Engineering
Koç University
Istanbul, Turkey
ecoban16@ku.edu.tr

Deniz Yüret
Computer Science and Engineering
Koç University
Istanbul, Turkey
dyuret@ku.edu.tr

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

***Abstract*—Broadcast is a common operation in machine learning and widely used in calculating bias or subtracting maximum for normalization in convolutional neural networks. Broadcast operation is required when two tensors possibly with different number of dimensions, hence with different number of elements, are input to an element-wise function. Tensors are scaled in process so that the two tensors match in size and dimension. In this research, we introduce a new broadcast functionality for matrices to be used on CUDA enabled GPU devices. We further extend this operation to multidimensional arrays and measure its performance against the implementation available in the Knet deep learning framework. Our final implementation provides up to 2x improvement over the Knet broadcast implementation, which only supports vector broadcast. Our implementation can handle broadcast operations with any number of dimensions.**

***Index Terms*—GPU, CUDA, machine learning, broadcast, multidimensional arrays**

I. Introduction

Machine learning algorithms can effectively learn to solve complex problems with the use of enormous quantities of data. Fortunately nowadays, more and more data is available in the forms of text data from the web (e.g. Wikipedia), or video from video publishing sites (e.g. Youtube). The Big Data era has allowed machine learning algorithms to succeed in various important computational tasks. These tasks involve image classification [4] and image recognition better than humans [2], sentiment analysis, caption generation [3] and many more. However, due to the complexity and iterative nature of machine learning algorithms, performance optimisations play an important role for their successful usage. With the incorporation of big data, the weights and parameter matrices necessary to construct an accurate machine learning algorithm hence grow larger in size as well, which makes basic matrix operations such as broadcast extremely critical to the time efficiency of those algorithms. Therefore, in this paper we focus on optimizing the broadcast operation in order to speed up such algorithms.

In machine learning, broadcast is used in summing the bias and weight matrices, in loss function calculations. For example, images can be processed in batches to run on a GPU for higher efficiency. This process may require weights and bias to be used more than once for a batch, thus those values are broadcast. As a result, optimizing the broadcast operation is essential for speeding up the machine learning algorithms.

In this paper, we propose a solution for multi dimensional broadcast operation on GPUs. We map CUDA threads to resulting tensor elements. For element-wise operations each thread needs to know which elements to use from input tensors to compute the output tensor. However, since broadcast tensors have fewer elements than the resulting tensor, correct indices should be calculated. Our method uses the stride values of tensors and coordinates of resulting element to calculate the corresponding elements' indices from input tensors. Because the elements of the broadcast tensors are used more than once, different threads access the same elements over and over again from GPU memory. This causes poor utilization of memory bandwidth. For this reason we propose data reuse methodology. Each thread loads an element from the broadcast tensor and uses this element to calculate more than one element in the resulting tensor. The highest data reuse is achieved when one thread is assigned to each element of the broadcast dimension. However, when the size of broadcast dimension is low, it causes underutilization of GPU cores. To balance the trade-off between data reuse and GPU core usage, we introduce a sliding factor. Sliding factor indicates how many times a thread will use the same data.

We implement the broadcast operation and compare its performance against the Knet deep learning framework [7], which only supports vector broadcasting. Knet is written in Julia [1], which is a high level programming language that generates CUDA code in the background. Our implementation achieves up to 2x speedup over Knet, and also supports multidimensional capability for broadcast operations. Our implementation can be used by any other machine learning frameworks that require broadcast support for GPU programming.

II. Broadcast Operation

Broadcast operation is an element-wise operation requiring two input tensors, where one tensor is broadcasted over the other by applying a binary function (e.g. addition, multiplication). For a broadcast operation, the number of elements in the same dimension of two tensors should be the same or one of these tensors should be a vector in other words have a single dimension.

In relevance to machine learning we focus on broadcasting tensors rather than scalar values. Figure 1 is a broadcast example of a matrix X and a vector Y, performing an addition.

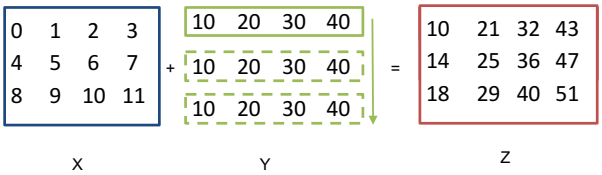

Fig. 1: Broadcasting a vector on a matrix by addition operation

To match the dimension of the matrix X for the addition operation, the vector Y is scaled by replicating the same elements over the missing dimension and then elements of the scaled vector is added to the elements of the matrix to obtain matrix Z.

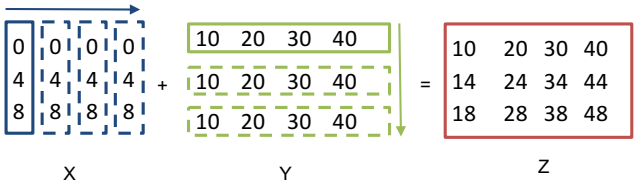

Fig. 2: Broadcasting a row over a column vector by addition operation

Both of the tensors, also multiple dimensions from each tensor can be broadcast. Figure 2 shows the addition of a row and column vectors. It demonstrates the case where both of the vectors need to be scaled in their respective missing dimension. This example also shows that resulting tensor always have the same or bigger dimension than both of the input tensors.

III. Implementation

A. Scale or not to Scale

In the previous section, we describe the broadcast operation by scaling tensors by replicating their elements in the missing dimension in memory. If a vector with N elements is scaled to the size of $N * M$ matrix, we would need extra memory space for $(N-1) * M$ elements, which is not the best way to implement a broadcast operation. Since device memory is limited on GPUs, replicating elements is not a good option. Moreover broadcast operation is mostly memory bound due to low computational density; for two memory accesses, only one arithmetic operation is performed at best. For these reasons, accessing the same element multiple times wastes memory bandwidth as well as memory capacity. If one chooses not to replicate the data in memory, then there is a challenge of finding the corresponding indices of a tensor because if the tensors are scaled in memory, they would have the same size and their corresponding elements would have the same index value. To compute each element of the resulting tensor, we need to calculate corresponding elements' indices. Because indices will not remain the same, we propose a representation for index values of elements to facilitate the calculation of the corresponding indices.

B. Index Calculation

Multidimensional arrays can be stored in memory continuously, similar to a one dimensional array. One can calculate the global index of an element from its coordinates in each dimension. For example, X array in Figure 3, has an element at coordinate (0,1) with value of 8, which has a global index of 4. However, elements in the Y vector other than first row do not exists in memory. For this reason, during broadcast, access to those indices should be translated into original elements' indexes because the vector is not physically scaled. In Figure 3, the circled element from the broadcasted vector Y has the same coordinates with the corresponding element from X matrix, however its index is different, which is 0. As a result, index values of elements in broadcast tensor do not represent the correct place of that element in memory.

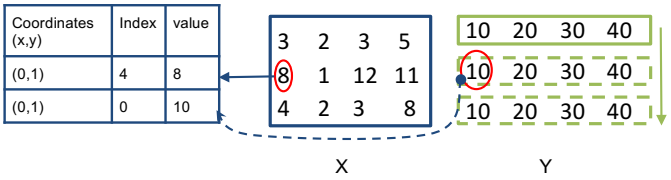

Fig. 3: Coordinates and indices in tensors

An element's index can be calculated from its coordinate values and tensor's stride values [6]. A stride value of a dimension is the number of jumps in the memory required to access the next element in the same dimension [5]. In other words, let x, y, z be integers representing coordinate values of an element in a 3D tensor, stride value of the third dimension, is the number of jumps required to get from element (x, y, z) to element $(x, y, z+1)$.

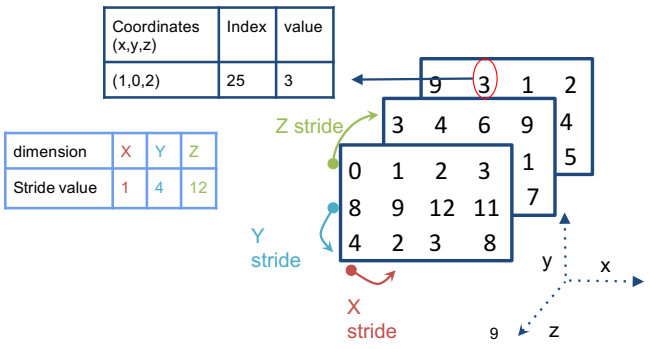

Fig. 4: Coordinate and stride values of 3D tensor

To calculate index of an element we take the dot product of the stride values of the matrix with the coordinates of the element. As an example, the index of the circled element in Figure 4 is:

$$Index = stridevalues[\,] . * coordinates[\,]$$

$$Index = [1, 4, 12] . * [1, 0, 2]$$

We store the stride values in memory for index calculation. To iterate from one element to the next, we increase the

coordinate values. This representation of indices makes the broadcast operation working on the non-contiguous tensors easier. The stride values can be changed according to one's needs. In case of broadcast, we set the stride value of the broadcasted dimensions to 0. Therefore, when we change the coordinates in that dimension, i.e. when we multiply with stride values, the broadcasted dimensions will not have an effect on index calculation.

C. CUDA Implementation

In a serial implementation of broadcast operation, we would iterate over resulting tensor elements with an iterator. Iterator would increase the coordinate values one by one and set to zero when it reaches to an upper limit. When an index calculation is required, then the coordinate and stride values would be used. This is possible because one iterator is responsible for calculating all the elements. In CUDA, we cannot do this because broadcasted tensor is not kept contiguously in memory and an increment on a coordinate value would require jumps in memory. we assign threads to the resulting tensor, which is always kept continuously in memory. From resulting tensor's index (global index), we calculate the coordinates of the resulting tensor, which has the same coordinates for input elements. The corresponding memory indexes of broadcast elements are then calculated from the stride and coordinate values for each input. Only after the index values are known, arithmetic operation can be applied.

D. Sliding Factor Optimization

Algorithm we described previously supports tensors with all different number of dimensions. This wide support prevents us to optimize our algorithm easily. For this reason, we create an optimized broadcast implementation that limits the operation inputs to a vector and a matrix, which is still commonly used in machine learning.

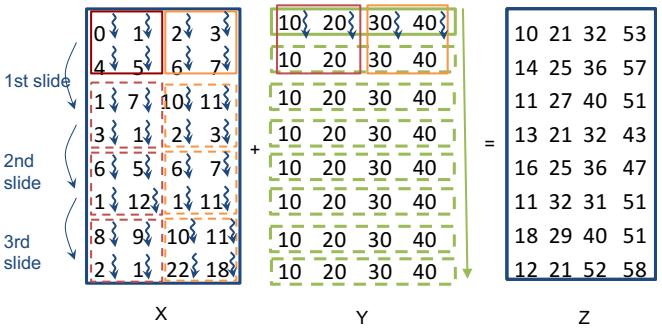

Fig. 5: Sliding thread blocks over matrix for addition

As mentioned earlier, broadcast operation is memory bandwidth bound. To enable data reuse for improving the memory bandwidth usage, we assign a single thread to apply the same broadcasted value to multiple rows or columns of the matrix. A thread loads element once from the vector in GPU memory and slides down this value through from beginning of matrix to end of it. In this scenario, an thread loads an element from a vector with size of N and uses this element for M times over a matrix with dimensions of NxM if a single 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 implementation as *thread sliding* and how many times a thread slides as *sliding factor*. Figure 5 shows two thread blocks with 2x2 threads in each. Each thread is assigned to one element from the vector Y but responsible for calculating 4 elements of resulting matrix Z. A 2x2 thread block shown in the figure slides over the matrix and new locations after each slide is drawn with dashed lines.

E. Shared Memory Optimization

GPUs come with software managed memory, called shared memory, which can act as a cache managed by the programmer. For the broadcast operation we explore the usage of shared memory to improve the performance further. In the shared memory implementation In each thread block, all threads in the first row load one element from vector Y to shared memory. Threads in the same thread block but in other rows can use the values loaded into the shared memory by the threads in the first row. Then similar to sliding factor implementation, thread blocks start from the top of the matrix and slides over the matrix until end of it, adding numbers from vector. If we refer back to the example in Figure 5, two threads in the first row of each thread block would load data from vector Y to shared memory. In the sliding factor implementation every thread in a block would load data from global memory.

IV. Results

We explained three algorithms, first one is generalized broadcast algorithm, which supports tensors with any number of dimensions. Secondly we proposed two performance improvements over generalized algorithm specific to vector broadcast. We used idea of sliding factor to increase data reuse. Next, we added shared memory support to sliding factor algorithm to decrease memory accesses even further. In this section, we compare the performance of those implementations and use Knet's broadcast as baseline.

We measured the performance on an Nvidia Tesla K40m GPU. The bandwidth test provided by Nvidia shows 179GB effective bandwidth for the device memory. In our performance tests, we broadcast a vector to a matrix. Length of the vector determines how many elements will be broadcast. Length of second dimension of the matrix correlates with how many times we can use a broadcast element from the vector.

In Figure 6 and 7, we compare two optimized versions of algorithm and native implementation from Knet. Y-axis shows the effective bandwidth (Gb/s) achieved in the broadcast operation, so higher is better. X-axis shows the length of Y dimension in the matrix. In both figures, we can see that for the small number of elements, baseline is slightly better because there is not enough data reuse for optimization and the overhead of optimization slows down the program. Sliding factor with shared memory is always slower than the baseline.

The sliding factor implementation performs better than the baseline when the size of the Y dimension reaches 1000. At that point data reuse can be 4 to 6 times with large number of thread blocks so the algorithm uses all stream multiprocessors (SM) efficiently on the GPU. Otherwise, the number of thread blocks might not be sufficient to occupy all the SMs and performance of our algorithm suffers due to inactive SMs.

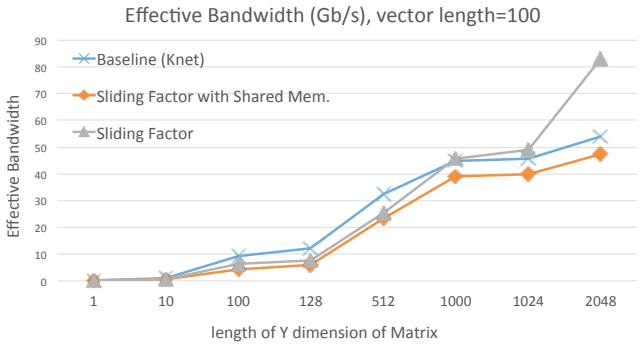

Fig. 6: Effective bandwidth achieved by Knet, sliding factor optimization, and sliding factor using shared memory. Higher is better.

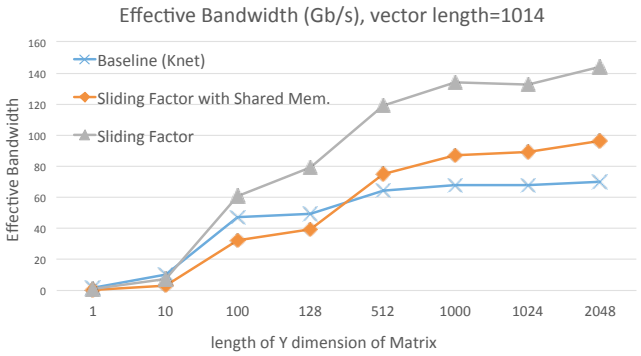

Fig. 7: Effective bandwidth achieved by Knet, sliding factor optimization, and sliding factor using shared memory. Vector size is 1024.

In Figure 7, 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 performs better than the baseline, reaching over 140 GB/s and getting closer to the sustained memory bandwidth of the device memory.

Lastly we compare Knet with our generalized multidimensional broadcast algorithm according to their effective bandwidth performance in Figure 8. X-axis shows the number of elements in a matrix in a single dimension. We could not compare Knet's performance on broadcasts other than vector broadcast because because Knet does not support it. Our algorithm is more generalized for this reason it is slower

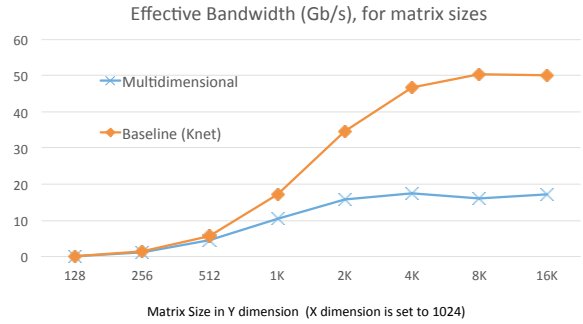

Fig. 8: Effective bandwidth comparison between generalized multidimensional broadcast and Knet for broadcasting a vector to a matrix with different sizes. Vector size is 1024.

than Knet, however, for vector broadcast one should use our implementation that performs the sliding factor optimization, which is a lot faster.

V. Conclusion

In this paper, we proposed an algorithm for multidimensional broadcast operation that works on CUDA enabled GPU devices. We optimized our algorithm for broadcasting vectors to over multidimensional arrays. We incorporated our algorithm into a deep learning framework called Knet, which was previously only capable of supporting vector broadcasts. The optimized vector broadcast achieves up to 2x speedup over the baseline and reaches 140GB/s on K40 GPUs. We also support multidimensional broadcast operation, which is general and can be applied to tensors with any dimensions. Future work will focus on improving the performance of other common machine learning operations such as reduction, permutation.

References

[1] J. Bezanzon, S. Karpinski, V. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. In *Lang.NEXT*, Apr. 2012.

[2] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[3] A. Karpathy and L. Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3128–3137, 2015.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[5] Wikipedia. Stride of an array — Wikipedia, the free encyclopedia, 2017. [Online; accessed 12-February-2017].

[6] G. Wilson and A. Oram. *Beautiful Code: Leading Programmers Explain How They Think*. Theory in Practice (O'Reilly). O'Reilly Media, 2007.

[7] D. Yuret. Knet: beginning deep learning with 100 lines of julia. In *Machine Learning Systems Workshop at NIPS 2016*, 2016.