SLIDING WINDOW SUM ALGORITHMS FOR DEEP NEURAL NETWORKS

Roman Snytsar

AI & Research, Microsoft, Redmond, USA

ABSTRACT

Sliding window sums are widely used for string indexing, hashing and time series analysis. We have developed a family of the generic vectorized sliding sum algorithms that provide speedup of O(P/w) for window size w and number of processors P. For a sum with a commutative operator the speedup is improved to O(P/log(w)). Even more important, our algorithms exhibit efficient memory access patterns. In this paper we study the application of sliding sum algorithms to the training and inference of Deep Neural Networks. We demonstrate how both pooling and convolution primitives could be expressed as sliding sums and evaluated by the compute kernels with a shared structure. We show that the sliding sum convolution kernels are more efficient than the commonly used GEMM kernels on CPUs and could even outperform their GPU counterparts.

KEYWORDS

Convolution, Machine Learning, Parallel, Vector

1. INTRODUCTION

A Deep Neural Network (DNN) is one of the most significant tools in the arsenal of the machine learning (ML) researcher [9]. DNNs are constructed from multiple layers that transform data sequentially via operations such as pooling, convolution, and activation. In most successful DNNs the vast majority of computational resources is consumed performing convolution.

A common approach to implementing convolutional layers is to expand the input into a column matrix (*im2col*) and then call a highly tuned General Matrix Multiplication (GEMM) procedure from the existing linear algebra library such as BLIS [13] or MKL [15]. Since the hardware optimized GEMM implementations exist for every standard CPU, graphics processing unit (GPU), or digital signal processor (DSP), the *im2col* approach has been highly successful in DNN frameworks such as Caffe [8], ONNX [2] and Torch [5]. The popularity of the *im2col* tactics also influence the design of custom hardware marketed as ML accelerators that are in fact GEMM accelerators.

The major disadvantages of the *im2col* conversion are the greatly increased memory footprint of the input matrix and the reduced data locality. For a convolution with a filter size *k*, the column matrix is *k* times larger than the original input. A lot of effort has been put into remediating this problem. A mainstream approach is converting the input to a low precision floating point or even integer representation [6]. The quantization techniques reduce the memory footprint and latency by an order of magnitude and even influence the design of the GEMM accelerators. It is important to note though that the quantization is not entangled with GEMM and could be equally successfully applied to the original convolution problem. Yet another research trend is applying the GEMM routines to the smaller intermediate data structures [1] [14] or even to the original input data [16].

Dhinaharan Nagamalai (Eds): EMVL, EDUT, SECURA, AIIoT, CSSE -2023
pp. 71-78, 2023. IJCI – 2023 DOI:10.5121/ijci.2023.120507

We propose an approach that replaces GEMM with a new kind of computation kernel that operates on the unmodified input.

2. METHODS

2.1. Prefix Sum

At the foundation of our method is the concept of a prefix sum, and the accompanying reduce and scan algorithms. A prefix sum is a transformation that takes an operator $\oplus$, and a sequence of elements

$$x_0, x_1, \ldots, x_k, \ldots$$

and returns the sequence

$$y_i = \sum_{j=0}^{i} x_j = x_0 \oplus x_1 \oplus \ldots \oplus x_i \tag{1}$$

or in recurrent form

$$y_{i+1} = y_i \oplus x_{i+1} \tag{2}$$

Despite the data carry dependency, the Nth element of the prefix sum with an associative operator could be computed in *O(log(N))* parallel steps using the reduce algorithm. An even stronger statement is that all N elements of the prefix sum could be computed in the same *O(log(N))* parallel steps using the scan algorithm, as shown by [3].

2.2. Sliding Window Sum

The sliding window sum (sliding sum) takes a window size *w*, an operator $\oplus$, and a sequence of elements, and returns the sequence

$$y_i = \sum_{j=i}^{i+w-1} x_j = x_i \oplus x_{i+1} \oplus \ldots \oplus x_{i+w-1} \tag{3}$$

where each sum is defined in terms of the operator $\oplus$ and contains exactly *w* addends. The asymptotic complexity of a naive sliding sum algorithm is *O(wN)* where *N* is the length of the source sequence.

Every sum defined by Equation 3 is a prefix sum with operator $\oplus$ and input sequence $x_i \ldots \oplus x_{i+w-1}$. Many useful operators are associative, so the prefix scan algorithm is applicable here, reducing complexity of every sum in Equation 3 to *O(log(w))* and, trivially, the overall sliding sum complexity to *O(Nlog(w))* parallel steps.

While working on the bioinformatics applications, we have used sliding window sums to represent the minimizer seeds and have developed a family of algorithms to achieve better parallel speedups [11]. Now we will apply the same approach to the DNN operators.

2.3. Pooling

The average pooling operator is trivially the sliding window sum with the associative operator +. By analogy, the max pooling operator is a sliding window sum with the associative operator *max*. Implementing the sliding pooling operator could be a warm-up before concentrating on the convolution.

2.4. Dot Product

As a corollary we will show that a dot product is a prefix sum. The dot product of the two vectors of length M is defined as:

$$c = \sum_{i=0}^{M-1} a_i \, b_i \tag{4}$$

First, we replace vectors *a* and *b* with vectors α and β so that

$$\alpha_i = \begin{cases} 1 & if\ a_i = 0 \\ a_i & otherwise \end{cases}, \beta_i = \begin{cases} 0 & if\ a_i = 0 \\ b_i & otherwise \end{cases} \tag{5}$$

It holds that

$$\sum_{i=0}^{M-1} \alpha_i \beta_i = \sum_{i=0}^{M-1} a_i b_i \tag{6}$$

Next, we define a sequence of (M + 1) pairs,

$$\gamma_i = \begin{pmatrix} u_i \\ v_i \end{pmatrix}, \quad where \quad u_i = \begin{cases} 1 & i = 0 \\ \alpha_{i-1} & 0 < i < M \\ \alpha_i & \\ \alpha_{M-1} & i = M \end{cases}, \quad and\ v_i = \begin{cases} \beta_i & i < M \\ 0 & i = M \end{cases} \tag{7}$$

and the operator $\oplus$ such that

$$\gamma_i \oplus \gamma_j = \begin{pmatrix} u_i \\ v_i \end{pmatrix} \oplus \begin{pmatrix} u_j \\ v_j \end{pmatrix} = \begin{pmatrix} u_i \cdot u_j \\ u_j \cdot v_i + v_j \end{pmatrix} \tag{8}$$

Operator $\oplus$ is associative [3], and the sequence

$$\delta_i = \begin{cases} \gamma_0 & i = 0 \\ \delta_{i-1} \oplus \gamma_i & 0 < i \leq M \end{cases} \tag{9}$$

is a prefix sum. The bottom element of the last sum δ_M is the original dot product, and δ_M could be evaluated using the reduce algorithm in *log(M)* parallel steps of fused multiply-add (FMA) operations. The total work is still *M*. The important result here is representing the dot product as a prefix sum in preparations for the sliding window implementation of the convolution.

2.5. Convolution

Convolution is a sliding window sum (dot product) with the associative operator defined by equation 8. Consequently, our family of sliding window algorithms is applicable to the convolution operator.

3. ALGORITHMS

Our first algorithm is a vector-friendly way of calculating the sliding sum assuming the input sequence elements become available one by one and are processed using the vector instructions of width $P > w$:

Algorithm 1 Scalar Input

procedure SCALARINPUT($x_0 \dots x_{n-1}$)

$$Y \leftarrow \Big(\underbrace{\sum_{j=0}^{w-2} x_j, \sum_{j=1}^{w-2} x_j, \dots, x_{w-3} \oplus x_{w-2}, x_{w-2}}_{w-1}, 0, \dots, 0\Big)$$

for $i = w - 1$ to N **do**

$$X \leftarrow \Big(\underbrace{x_i, x_i, \dots, x_i}_{w}, 0, \dots, 0\Big)$$

$$Y \leftarrow Y \oplus X$$

$$y_{i-w+1} \leftarrow Y[0]$$

$$Y \leftarrow Y \lll 1$$

end for

end procedure

Vector Y is initialized to the suffix sums with the number of elements decreasing from $w-1$ to 0. Then in a loop every incoming element x_k is broadcast to the first w elements of vector X. After vector addition the zeroth element of Y contains the next sliding sum. Next, the vector Y is shifted left by one element, as denoted by operator $\ll$, and the state is ready for the next iteration.
Asymptotic complexity of the scalar input algorithm is $O(N)$ with no additional requirements on the operator $\oplus$.

This result could be improved if we assume that the input sequence arrives packed in vectors of width $P > w$.

Algorithm 2 Vector Input

procedure VECTORINPUT($x_0 \dots x_{N-1}$)

$$Y \leftarrow \Big(\underbrace{\sum_{j=0}^{w-2} x_j, \sum_{j=1}^{w-2} x_j, \dots, x_{w-3} \oplus x_{w-2}, x_{w-2}}_{w-1}, 0, \dots, 0\Big)$$

for $i = w - 1$ to N step P **do**

$$X \leftarrow \Big(x_i, x_{i+1}, \dots, x_{i+p-1}\Big)$$

$$X1 \leftarrow \Big(\underbrace{X[0], X[0] \oplus X[1], \dots, \sum_{j=0}^{w-2} X[j]}_{w-1}, \sum_{j=0}^{w-1} X[j], \dots, \sum_{j=p-w}^{p-1} X[j]\Big)$$

$$Y1 \leftarrow \Big(0, \dots, 0, \underbrace{\sum_{j=p-w}^{p-1} X[j], \sum_{j=p-w}^{p-2} X[j], \dots, X[p-w]}_{w-1}\Big)$$

$$Y \leftarrow Y \oplus X1$$

$$y_{i-w+1} \dots y_{i-w+p} \leftarrow Y[0] \dots Y[p-1]$$

$$Y \leftarrow Y1 \lll (P - w)$$

end for

end procedure

At every iteration, *P* input elements are placed into vector *X*. *X1* is filled with the prefix sums of up to w addends, and *Y1* is filled with the suffix sums constructed from the elements of *X*. Then the vector sum of *Y* and *X1* yields the next *P* output elements. Finally, the suffix sums from *Y1* are shifted into proper positions in vector *Y*, and it is ready for the next iteration.

Algorithm 3 Ping Pong

procedure PINGPONG($x_0 \dots x_{N-1}$)
  for $i = 0$ to N step 2P-w+1 **do**
    $Y \leftarrow \left(x_i, x_{i+1}, \dots, x_{i+p-1}\right)$
    $X \leftarrow \left(x_{i+P}, x_{i+P+1}, \dots, x_{i+2P-1}\right)$
    $Y1 \leftarrow \Big(\underbrace{\sum_{j=0}^{w-1} Y[j], \sum_{j=1}^{w} Y[j], \dots, \sum_{j=P-w}^{P-1} Y[j]}_{P-w+1}, \sum_{j=P-w}^{P-1} Y[j], \dots, Y_{P-1}\Big)$
    $y_i \dots y_{i+P-w} \leftarrow Y[0] \dots Y[P-w]$
    $Y1 \leftarrow Y1 \lll (P-w)$
    $X1 \leftarrow \Big(\underbrace{X[0], X[0] \oplus X[1], \dots, \sum_{j=0}^{w-2} X[j]}_{w-1}, \sum_{j=0}^{w-1} X[j], \dots, \sum_{j=p-w}^{p-1} X[j]\Big)$
    $Y1 \leftarrow Y1 \oplus X1$
    $y_{i+P-w+1} \dots y_{i+2P-w} \leftarrow Y[0] \dots Y[P-1]$
  end for
end procedure

The asymptotic complexity thus is $O(N \cdot w/P)$ with the parallel speedup $O(P/w)$ for any operator $\oplus$. If $\oplus$ is associative, the prefix/suffix sums could be computed in parallel using the algorithm in [3], and the complexity is reduced to $O(N \cdot log(w)/P)$ with the speedup improving to $O(P/log(w))$.

For example, since min is an associative operator, the sliding window minimum can be computed using the faster version of the vector input algorithm.

Algorithm 4 Vector Slide

procedure SLIDE($Y1$, $Y2$, $offset$)
  $Y \leftarrow CONCAT(Y1, Y2)$
  return $\underbrace{\left(Y[offset], Y[offset+1], \dots, Y[offset+P-1]\right)}_{P}$
end procedure
procedure VECTORSLIDE($x_0 \dots x_{N-1}$)
  $Y \leftarrow (x_0, x_1, \dots, x_{P-1})$
  for $i = P$ to N step P **do**
    $Y1 \leftarrow \left(x_i, x_{i+1}, \dots, x_{i+p-1}\right)$
    $X \leftarrow Y1$
    for $k = 1$ to $w-1$ **do**
      $X \leftarrow X \oplus \text{Slide}(Y, Y1, P-k)$
    end for
    $y_i \dots y_{i+p-1} \leftarrow X[0] \dots X[p-1]$
    $Y \leftarrow Y1$
  end for
end procedure

One might notice that the suffix sum computation utilizes only $w - 1$ out of p elements of vector *Y1*. Eliminating this inefficiency leads to the Ping Pong algorithm where both suffix and prefix sums yield some elements of the output sequence.

The Ping Pong algorithm does not offer any asymptotic improvements but is 30-50% faster in practice. However, it accesses memory in strides unaligned to P, and the two memory loads per iteration present a challenge while implementing boundary conditions like padding, mirroring, or periodicity.

A simpler algorithm could be formulated in terms of the Slide operation that extracts a vector from the concatenation of the two inputs. It directly maps to the **EXT** instruction from the ARM SVE instruction set [12], and is easily implemented via the appropriately named **vslideup/down** instructions of the RISC-V vector extension [10], or the **vperm*2ps** Intel AVX512 instructions [7]. Similar to the previous algorithms, if $\oplus$ is associative, the inner loop could be replaced by the parallel reduction algorithm for maximum parallel speedup.

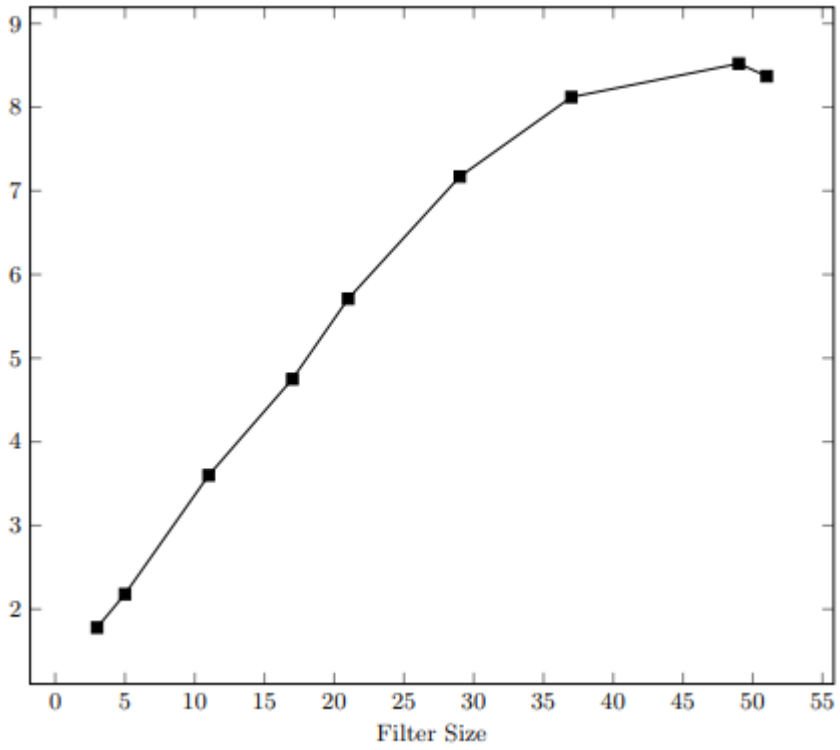

Figure 1. Speedup of the 1-D Convolution.

4. EXPERIMENTS

We have implemented the sliding window convolution as an alternate execution path in the ONNX framework [2]. Figure 1 shows the achieved speedup when compared to the baseline *MlasConv* procedure applied to the large 1-D input and the filters of various sizes. The speedup is approximately proportional to the logarithm of the kernel size.

Additionally, we have recreated the scenario for the very large dilated kernel described in [4]. We have achieved up to 6.8x speedup on the small data set and around 4x speedup across the board approaching the GPU performance numbers.

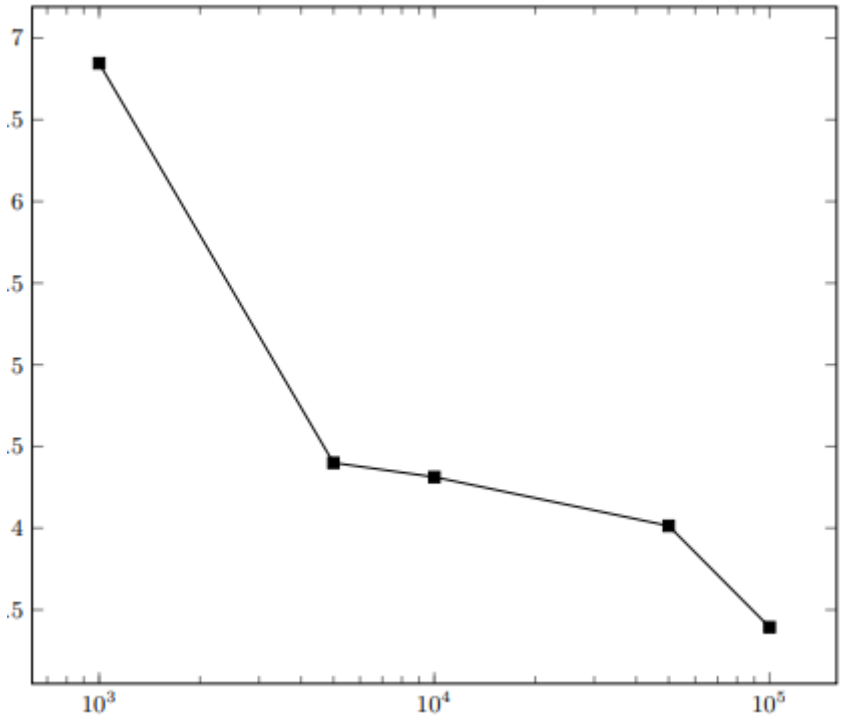

Figure 2. Dilated Convolution Speedup

5. CONCLUSION

We have demonstrated the excellent performance of Sliding Sum algorithms using commodity hardware without the need for specialized accelerators. Despite the promising results, there is plenty of work ahead.

The most obvious next step is extending the sliding convolution approach to more than one dimension covering the majority of DNN applications. The most common filter sizes in DNN applications are 3 and 5 in every dimension. With a filter this small the current sliding convolution algorithms demonstrate very modest speedup since the number of the arithmetic instructions per memory load is low. The situation improves in the multiple dimensions but still could require custom compute kernels for small filter sizes.

Lastly, since the accelerators for matrix multiplication are already present in the current generation of hardware, it would be wise re-using them. Thus, it is important to re-formulate our algorithms in terms of the small matrix multiplication completing the circle.

REFERENCES

[1] Anderson, A., Vasudevan, A., Keane, C., Gregg, D.: Low-memory gemm-based convolution algorithms for deep neural networks. arXiv preprint arXiv:1709.03395 (2017)
[2] Bai, J., Lu, F., Zhang, K.: Onnx: Open neural network exchange (May 2023), ONNX: Open Neural Network Exchange
[3] Blelloch, G.E.: Prefix sums and their applications. In: Synthesis of Parallel Algorithms. Morgan Kaufmann (1993)
[4] Chaudhary, N., Misra, S., Kalamkar, D., Heinecke, A., Georganas, E., Ziv, B., Adelman, M., Kaul, B.: Efficient and generic 1d dilated convolution layer for deep learning. arXiv preprint arXiv:2104.08002 (2021)

[5] Collobert, R., Bengio, S., Mari´ethoz, J.: Torch: a modular machine learning software library. Tech. rep., Idiap (2002)
[6] Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M.W., Keutzer, K.: A survey of quantization methods for efficient neural network inference. arXiv preprint arXiv:2103.13630 (2021)
[7] Intel 64 and ia-32 architectures software developer manuals (May 2023), https://www.intel.com/content/www/us/en/developer/articles/technical/intelsdm.html
[8] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. In: Proceedings of the 22nd ACM international conference on Multimedia. pp. 675–678 (2014)
[9] Li, Z., Liu, F., Yang, W., Peng, S., Zhou, J.: A survey of convolutional neural networks: analysis, applications, and prospects. IEEE transactions on neural networks and learning systems (2021)
[10] Riscv-v-spec (May 2023), https://github.com/riscv/riscv-v-spec
[11] Snytsar, R., Turakhia, Y.: Parallel approach to sliding window sums. In: Algorithms and Architectures for Parallel Processing: 19th International Conference, ICA3PP 2019, Melbourne, VIC, Australia, December 9–11, 2019, Proceedings, Part II. pp. 19–26. Springer (2020)
[12] Stephens, N., Biles, S., Boettcher, M., Eapen, J., Eyole, M., Gabrielli, G., Horsnell, M., Magklis, G., Martinez, A., Premillieu, N., et al.: The arm scalable vector extension. IEEE micro 37(2), 26–39 (2017)
[13] Van Zee, F.G., Van De Geijn, R.A.: Blis: A framework for rapidly instantiating blas functionality. ACM Transactions on Mathematical Software (TOMS) 41(3), 1–33 (2015)
[14] Vasudevan, A., Anderson, A., Gregg, D.: Parallel multi channel convolution using general matrix multiplication. In: 2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP). pp. 19–24. IEEE (2017)
[15] Wang, E., Zhang, Q., Shen, B., Zhang, G., Lu, X., Wu, Q., Wang, Y., Wang, E., Zhang, Q., Shen, B., et al.: Intel math kernel library. High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures pp. 167–188 (2014)
[16] Wang, H., Ma, C.: An optimization of im2col, an important method of cnns, based on continuous address access. In: 2021 IEEE International Conference on Consumer Electronics and Computer Engineering (ICCECE). pp. 314–320. IEEE (2021)

AUTHORS

Roman Snytsar is a Principal member of Microsoft Research, working on software optimization and energy efficiency problems.