Implementing block-sparse matrix multiplication kernels using Triton

Priya Mishra [1] **Trevor Gale** [1] **Matei Zaharia** [1] **Cliff Young** [2] **Deepak Narayanan** [3]

Abstract

MegaBlocks is the state-of-the-art system for efficient training of MoE models based on block-sparse matrix multiplication kernels. The library is currently restricted to a specific block size in the sparse matrices, data type, and GPU architecture. This is due to the CUDA kernels used for the block-sparse matrix products in the MoE layers. These kernels have been hand-tuned and manually optimized to obtain the highest performance for a specific choice of parameters.

In this work, we evaluate re-writing these kernels in Triton, a Python-embedded domain specific language (DSL) for high-performance kernels for GPUs. We show that it is possible to achieve same levels of performance as the hand-tuned CUDA kernels, while maintaining portability across GPU architectures and easily supporting different block sizes and data types without any code changes. We identify the challenges and advantages of using Triton in implementing these block-sparse matrix multiplication kernels.

1. Introduction

MegaBlocks is the state-of-the-art system for efficient training of MoE models on GPUs (Gale et al., 2022). It is based on a reformulation of MoE computations into block-sparse operations that accommodates imbalanced assignment of tokens to experts without requiring dropping tokens or adding padding. Despite its efficiency, MegaBlocks can only be used with a block size of 128 for sparse matrices, fp16 datatype, and Ampere GPU architecture.

This limitation is a direct result of the block-sparse matrix multiplication kernels used by MegaBlocks. These block-sparse kernels are implemented in CUDA and are challenging to generalize across possible block sizes, data types, and GPU architectures. The choice of these parameters affects design decisions of the kernels, for instance, the shared memory allocation and layout. The kernels have been manually optimized for specific values of these parameters. To maintain the same level of performance for a different set of parameters requires non-trivial implementation changes to the kernels, which makes experimenting across a range of settings challenging.

In this work, we evaluate re-writing these kernels in Triton (Tillet et al., 2019). Triton is a python embedded domain specific language (DSL) for implementing high-performing GPU compute kernels. After some evaluation, we find that our Triton kernels achieve the same level of performance as the highly optimized existing CUDA kernels. The kernels are compact, with $10\times$ fewer lines of code than the CUDA kernels, and are portable across GPU architectures. They support any data type or block size out-of-the-box without requiring any code change while maintaining high performance. Using Triton greatly simplifies the block-sparse kernels, however, we encountered significant road blocks in realizing this. First, Triton is under active development and provides limited documentation which made it challenging to implement and debug performance issues. Second, triggering optimizations like software pipelining required careful experimentation and reformulating the kernel.

After some investigation, we were able to replace the existing CUDA kernels with these Triton kernels. This allows us to generalize MegaBlocks for training MoEs with any configuration while maintaining the training efficiency and model quality [1].

2. Implementation

To describe the matrix products, we use the notation introduced in (Tillet et al., 2019). Each operation is described using three letters. The letters denote the output, left hand input and right hand input respectively. Hence, SDD denotes the operation $sparse = dense \times dense$. The letters following this denote whether the input matrices are transposed. We use N if the matrix is not transposed, and T if the matrix is transposed. For example, TN denotes that the left hand input matrix is transposed and the right hand matrix is

[1]Stanford University [2]Google Brain [3]Microsoft Research. Correspondence to: Priya Mishra <priyamis@cs.stanford.edu>.

Work presented at the ES-FoMo Workshop at 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

[1]Complete code at https://github.com/stanford-futuredata/stk

not transposed.

We use the hybrid blocked CSR-COO sparse matrix encoding introduced in MegaBlocks (Gale et al., 2022). This format is based on blocked compressed sparse row (BCSR) primitive making it easy to iterate over the nonzero blocks when one of the input matrices is sparse (DSD or DDS). When the output matrix is sparse (SDD), we need to identify both the row and column of each nonzero block which requires looping through the row offsets since BCSR only stores the column indices of the nonzero blocks. The hybrid blocked CSR-COO encoding additionally stores the row indices for each nonzero block for trivial access in SDD kernels. Since the MoE computations require sparse matrix transposition during forward and backward passes, this encoding also maintains metadata for the transposed matrices without explicitly transposing the nonzero blocks. This allows efficient iteration over the transposed sparse matrix through a layer of indirection.

The implementation of our SDD matrix product launches a 1D grid of thread blocks that run in parallel. The number of thread blocks launched is equal to the number of nonzero blocks in the sparse matrix output. Each thread block uses the blocked-COO metadata to identify the row and column of the nonzero block, and hence the sub-matrices of the dense input operands needed to compute the particular nonzero block. Pseudo-code for SDD kernel is shown in Figure 1. The kernel provides a high level abstraction of the matrix multiplication computation simplifying the code. The lower level details such as shared memory allocation or synchronization are handled by the Triton compiler. Note that the kernel is not specific to any block size or data type, allowing ease of experimentation for a range of configurations.

For DSD and DDS, we launch a 2D grid of thread blocks in the size of number of blocks in each dimension. In contrast to SDD, these operations involve indirect memory access of the dense input operand based on the layout of the nonzero blocks in the sparse input operand. We use the sparse matrix metadata to only load the sub-matrices of the dense input that will contribute to a nonzero block in the sparse matrix.

```
def _sdd_kernel (A, B, C, M, N, K,
                 BLOCK_M, BLOCK_N, BLOCK_K,
                 row_indices, column_indices):

    # Identify the row & column of non-zero block
    pid = tl.program_id()
    pid_m = tl.load(row_indices + pid)
    pid_n = tl.load(column_indices + pid)

    # Pointers to the input matrices
    A = A + pid_m * BLOCK_M
    B = B + pid_n * BLOCK_N

    # Matrix multiplication
    acc = tl.zeros(BLOCK_M, BLOCK_N)
    for k in range (0, K // BLOCK_K):
        a = tl.load(A)
        b = tl.load(B)
        acc += tl.dot(a, b)
        A += BLOCK_K
        B += BLOCK_K

    # Store to sparse output matrix
    C = C + pid * BLOCK_M * BLOCK_N
    tl.store (C, acc)
```

Figure 1. Pseudo-code of SDD kernel in Triton.

3. Evaluation

We started with implementing the kernels in Triton 2.0, the latest stable release. However, the throughput of the Triton implementation was 50% worse than the CUDA implementation. This major gap in performance was a result of missing software pipelining when accessing the input matrices through a layer of indirection from the sparse matrix metadata. Support for software pipelining with indirect memory accesses was added during our investigation [2]. Hence, we use a nightly build of Triton for all experiments. Since Triton is being actively developed, it has limited documentation and can have unexpected behavior, for instance, the pipelining issue described above. It is also unclear to which cases the Triton compiler optimizations apply, and we experimented with multiple ways of structuring the main loops in our kernel to trigger software pipelining. We relied on the PTX source code to understand the Triton code compilation and debug performance issues. Adding debugging support (Brahmakshatriya & Amarasinghe, 2023) in Triton can help in identifying these bottlenecks more easily.

In the following sections, we evaluate the Triton implementation (MegaBlocks-Triton) against the CUDA implementation (MegaBlocks-CUDA) of the MegaBlocks kernels.

We note that Blocksparse is an existing library built using Triton for block-sparse matrix multiplication kernels [3] (Tillet et al., 2019). However, it assumes that the location of nonzero blocks in the matrices remains constant between kernel invocations. These kernels rely on pre-processing the sparse matrix layout to compute lookup tables which adds a significant overhead. This makes Blocksparse unusable for MoE computations since subsequent training iterations and different MoE layers have different sparsity layout and we would incur the cost of the pre-processing at every step repeatedly. Hence, we did not use Blocksparse in our work.

3.1. Throughput

The performance of blocksparse matrix multiplication kernels is crucial to the training speedups obtained by

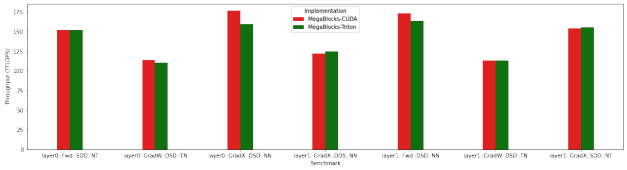
[2] https://github.com/openai/triton/pull/1291

[3] https://github.com/openai/triton/blob/main/python/triton/ops/blocksparse/matmul.py

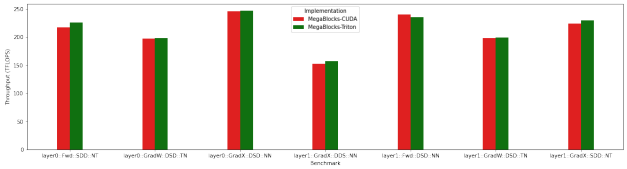
MegaBlocks. We noted in section 2 that kernels using previous versions of Triton had significant performance limitations, necessitating the present CUDA-based kernels used within MegaBlocks.

In this section, we evaluate the throughput obtained with these implementations for various matrix operations and different matrix sizes (sequence length/output matrix side length). The notation used for the matrix operations is described in section 2.

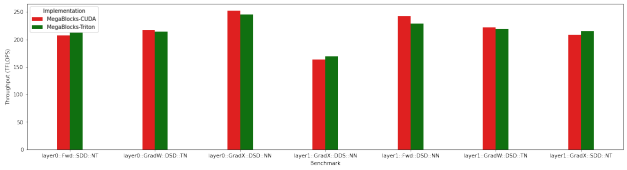
Figure 2 compares the throughput (in TFLOPS) using different implementations across a range of matrix product benchmarks. We observed that MegaBlocks-Triton achieves same levels of throughput as the MegaBlocks-CUDA implementation. The throughput of MegaBlocks-Triton kernels ranges between $0.96\times$ to $1.1\times$ compared to the throughput of MegaBlocks-CUDA kernels and the implementations have the same throughput on average . This makes it feasible to replace the CUDA-based kernels with MegaBlocks-Triton.



(a) Sequence length 8192.



(b) Sequence length 32768.



(c) Sequence length 65536.

Figure 2. Throughput (in TFLOPS) for different matrix product benchmarks using MegaBlocks-Triton and MegaBlocks-CUDA implementations of block-sparse matrix multiplication kernels.

3.2. Source Lines-of-Code

We use source lines-of-code (SLOC) (Wheeler) as a measure to compare the complexity of different implementations. As summarized in Table 1, MegaBlocks-Triton has 10x fewer lines of code than MegaBlocks-CUDA.

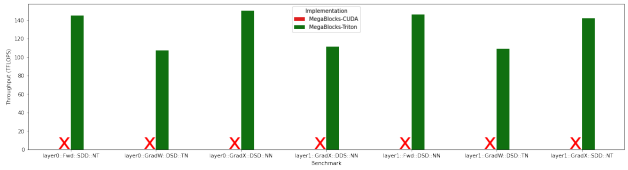
Table 1. Source Lines-of-Code in different implementations

IMPLEMENTATION	SLOC
MEGABLOCKS-TRITON	298
MEGABLOCKS-CUDA	3139

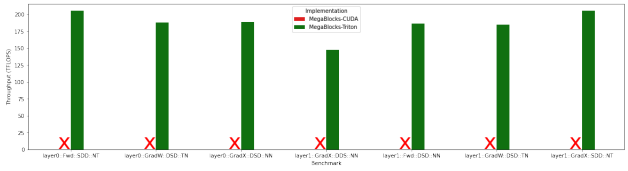
Triton allows us to write the kernels at a high-level abstraction and hides the lower-level optimization details such as shared memory allocation, synchronization, and memory coalescing. This makes the MegaBlocks-Triton kernels compact, portable across different hardware architectures and easily extensible to any block size and data type without requiring any code modification.

3.3. Generality

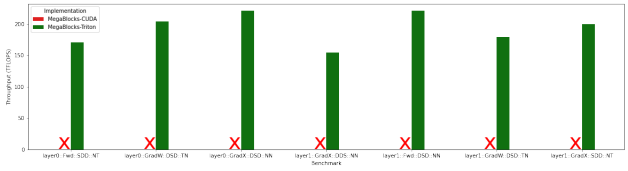
Figures 3 and 4 describe the throughput of MegaBlocks-Triton kernels for bfloat16 datatype and block size 64 respectively. These parameters are not supported by MegaBlocks-CUDA. The average throughput obtained when using bfloat16 across different sequence lengths and matrix operations is 170 TFLOPS, and when using a block size 64 is 130 TFLOPS. MegaBlocks-Triton supports a range of configurations out-of-the-box while maintaining high levels of performance.



(a) Sequence length 8192.

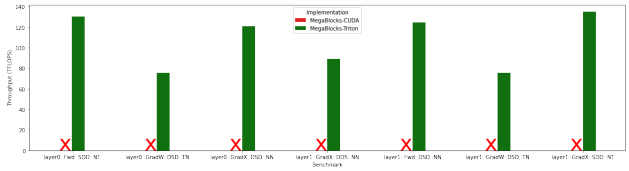


(b) Sequence length 32768.

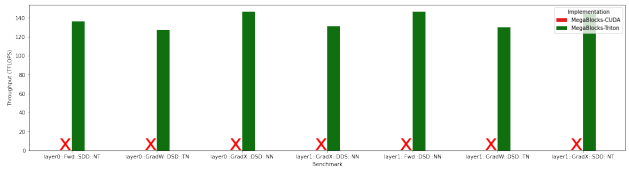


(c) Sequence length 65536.

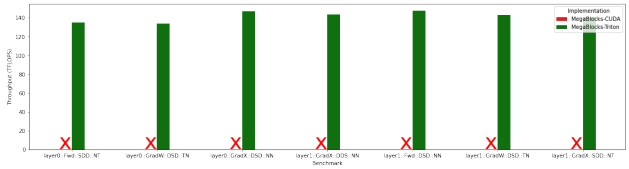
Figure 3. Throughput (in TFLOPS) for different matrix product benchmarks with block size 128 and bfloat16 datatype.



(a) Sequence length 8192.



(b) Sequence length 32768.



(c) Sequence length 65536.

Figure 4. Throughput (in TFLOPS) for different matrix product benchmarks with block size 64 and float16 datatype.

4. Conclusion

In this work, we evaluated re-writing the block-sparse matrix multiplication kernels used for MoE computations in MegaBlocks in Triton. Using Triton greatly simplified our kernels, however, it is under active development and has unexpected behavior in some cases. It is challenging to debug since it there is limited documentation, and the optimizations applied by the Triton compiler in different kernel formulations can vary significantly. We relied on the PTX source code to debug performance issues and reformulated the kernels multiple times to trigger certain optimizations such as software pipelining.

We show that our Triton kernels (MegaBlocks-Triton) can obtain same performance as the existing CUDA kernels (MegaBlocks-CUDA) making them a feasible replacement. Replacing the existing CUDA kernels with MegaBlocks-Triton removes any restrictions on block size, data type, and architecture. This extends MegaBlocks into a general system for training MoE models across a range of configurations while realizing the same training speedups and model quality.

References

Brahmakshatriya, A. and Amarasinghe, S. D2x: An extensible contextual debugger for modern dsls. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2023, pp. 162–172, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701016. doi: 10.1145/3579990.3580014. URL https://doi.org/10.1145/3579990.3580014.

Gale, T., Narayanan, D., Young, C., and Zaharia, M. Megablocks: Efficient sparse training with mixture-of-experts, 2022.

Tillet, P., Kung, H. T., and Cox, D. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, pp. 10–19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367196. doi: 10.1145/3315508.3329973. URL https://doi.org/10.1145/3315508.3329973.

Wheeler, D. A. Sloccount. https://dwheeler.com/sloccount/.