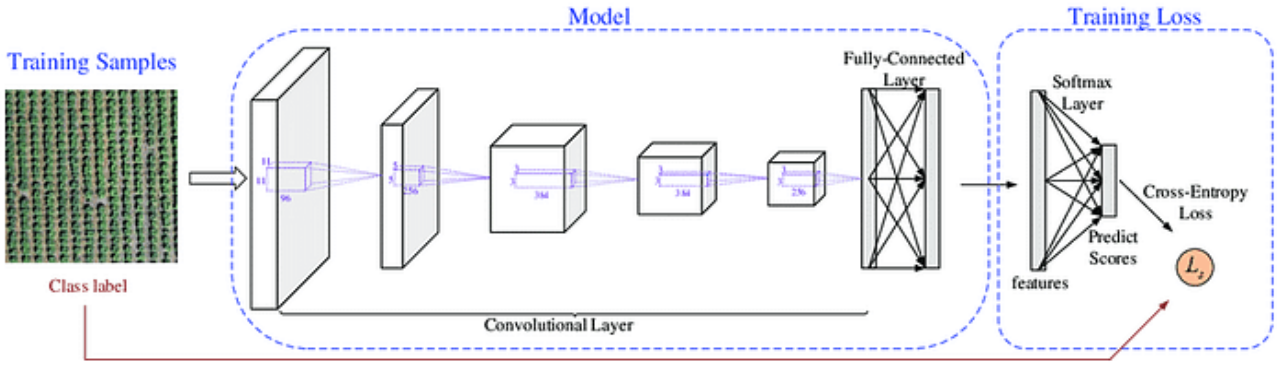
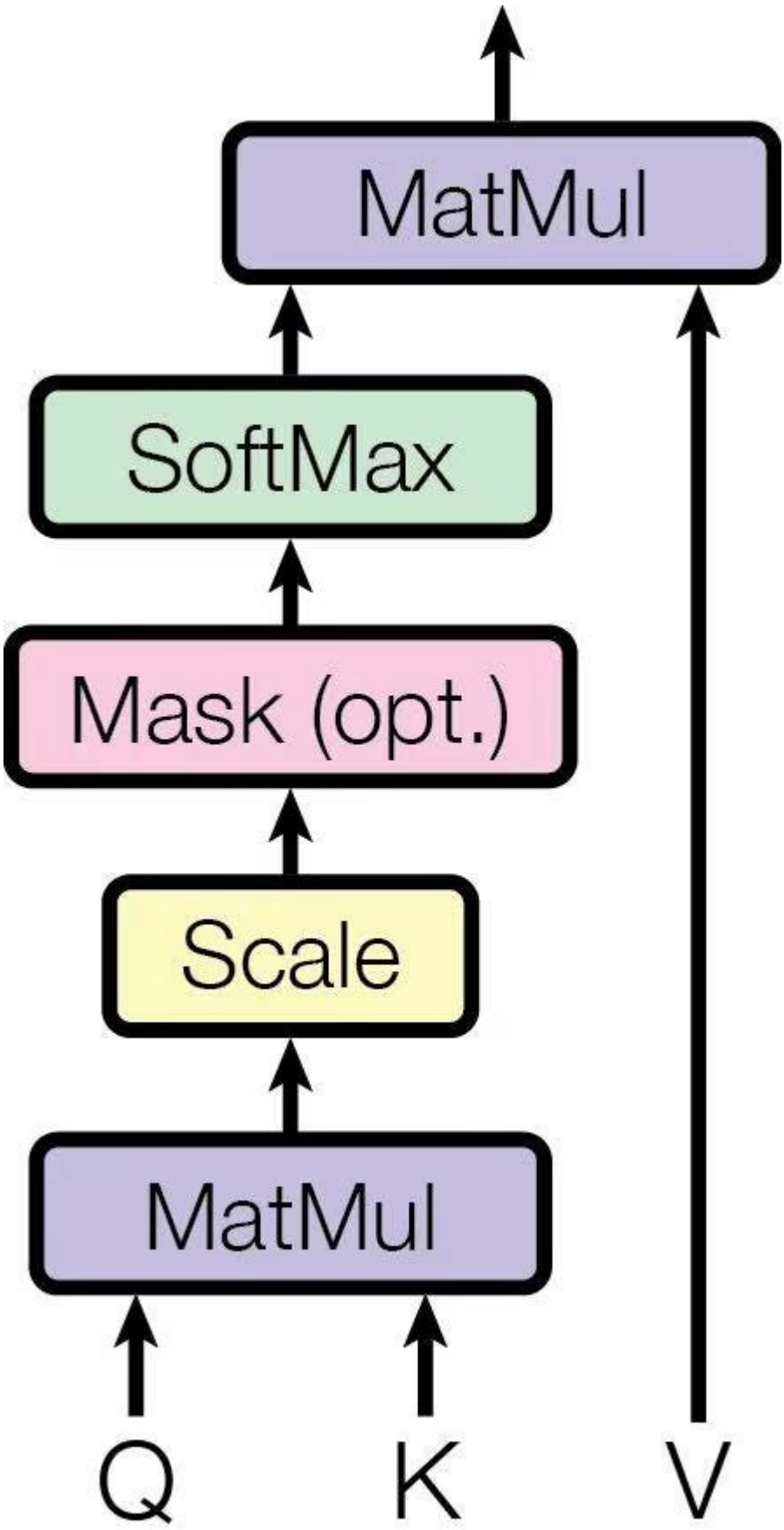
How to Implement an Efficient Softmax CUDA Kernel— OneFlow Performance Optimization

Written by Ran Guo; Translated by Kaiyan Wang

The Softmax operation is one of the most common operations in deep learning models. In deep learning classification tasks, the network's final classifier is often a combination of Softmax and CrossEntropy:



Although the mathematical derivation can be reduced when Softmax and CrossEntropy are used together, there are many scenarios where Softmax Op is used separately. For example, Softmax is used separately to solve the probability distribution of attention in the attention layer of each layer of BERT's Encoder; Softmax is also used separately in the multi-head part of GPT-2's attention, etc.



All ops computed in deep learning frameworks are translated into CUDA kernel functions on the GPU, and

Softmax operations are no exception. Softmax is a widely used op in most networks, and the efficiency of its CUDA kernel implementation can affect the final training speed of many networks. So how can an efficient Softmax CUDA Kernel be implemented?

In this article, we will introduce techniques for optimizing the Softmax CUDA Kernel in **OneFlow** and experimentally compare it with the Softmax operation in cuDNN. The results show that OneFlow's deeply optimized Softmax can utilize the memory bandwidth close to the theoretical upper limit, much higher than the cuDNN implementation.

GPU Basics and CUDA Performance OptimiZation Principles:

An introduction to GPU basics and the principles and objectives of CUDA performance optimization can be found in the previous article:

https://zhuanlan.zhihu.com/p/271740706

The hardware architecture and execution principles of the GPU are briefly introduced:

- Kernel: the CUDA Kernel function, is the basic computational task description unit of the GPU. Each Kernel is executed in parallel by very many threads on the GPU according to the configuration parameters. GPU computation is efficient because it can be executed by

thousands of cores (threads) at the same time, making it far more efficient than the CPU.
- GPU threads are logically divided into Thread, Block and Grid levels, and hardware is divided into CORE and WARP levels.
- GPU memory is divided into Global memory, Shared memory, Local memory three levels.
- The GPU provides two main resources: computing resources and display memory bandwidth resources. If we can make full use of these two resources, and the demand on them cannot be reduced any further, then performance is optimized to the limit and execution time is minimized. In most cases, the GPU computational resources are fully utilized in deep learning training, and the optimization goal of a GPU CUDA Kernel is to make the best possible use of the memory bandwidth resources.

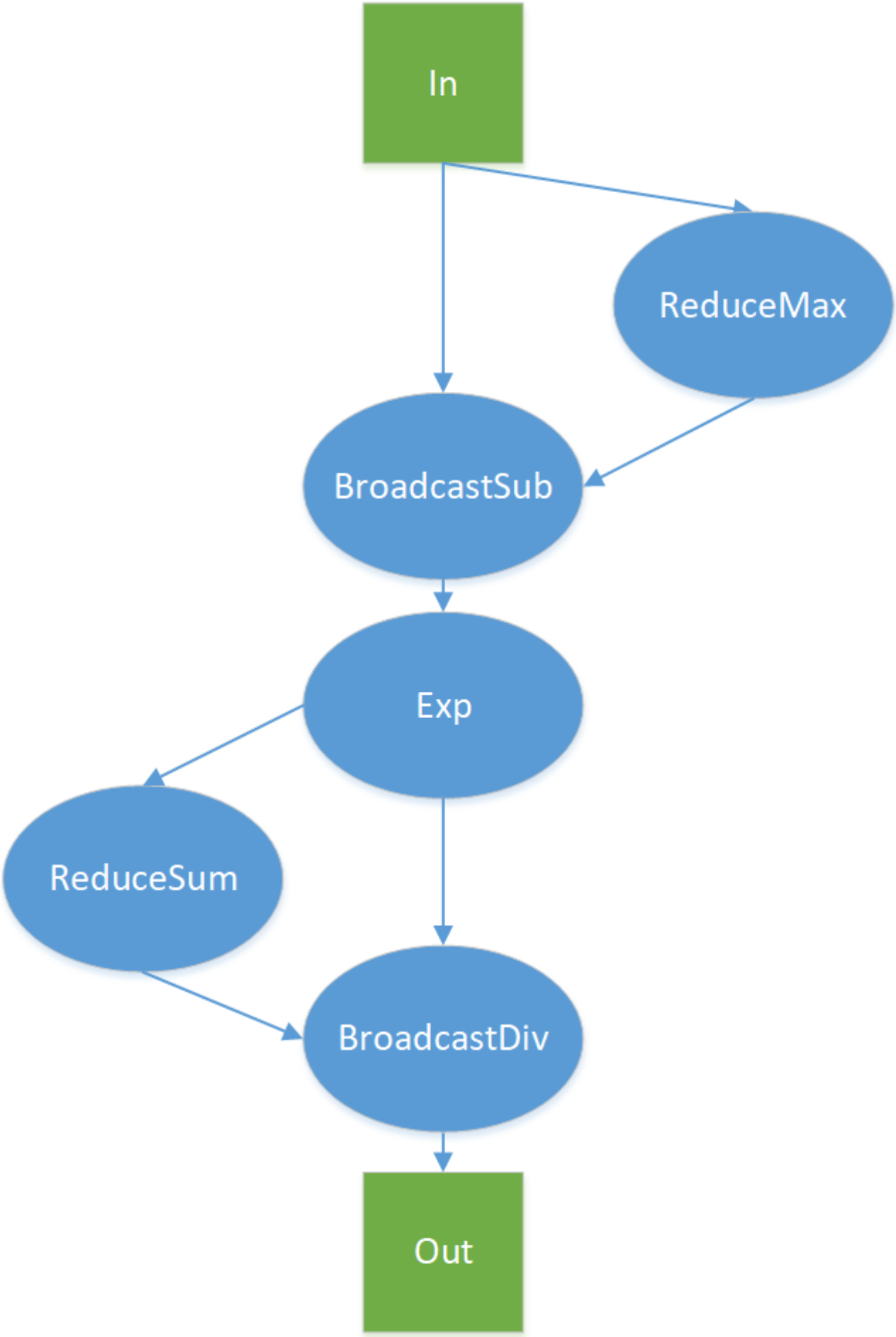
How to Assess Whether a CUDA Kernel is Making Full Use of the Video Memory Bandwidth Resources?

For memory bandwidth resources, “fully utilized” means that the Kernel’s effective memory read/write bandwidth is at the maximum of the device’s memory bandwidth, which can be obtained by executing bandwidthTest in cuda. The effective memory bandwidth of the Kernel is evaluated by the amount of data read and written to the Kernel and the execution time of the Kernel: *Current effective memory bandwidth of the*

Kernel = amount of data read and written / execution time

Naive Softmax Implementation:

Before presenting the optimization techniques, let’s see what the maximum theoretical bandwidth of an non-optimized Softmax Kernel is. As shown in the figure below, a simplest implementation of a Softmax computation calls each of several basic CUDA Kernel functions to complete the overall computation:


In
ReduceMax
BroadcastSub
Exp
ReduceSum
BroadcastDiv
Out

Assuming that the input data size is `D` and `shape = (num_rows, num_cols)`, i.e. `D = num_rows * num_cols`. The most Navie operation will access Global memory multiple times, where:

- ReduceMax = `D + num_rows` (read for D, write for num_rows)
- BroadcaseSub = `2 * D + num_rows` (read for D + num_rows, write for D)
- Exp = `2 * D` (read and write are both D)
- ReduceSum = `D + num_rows` (read as D, write as num_rows)
- BroadcastDive = `2 * D + num_rows` (read as D + num_rows, write as D)

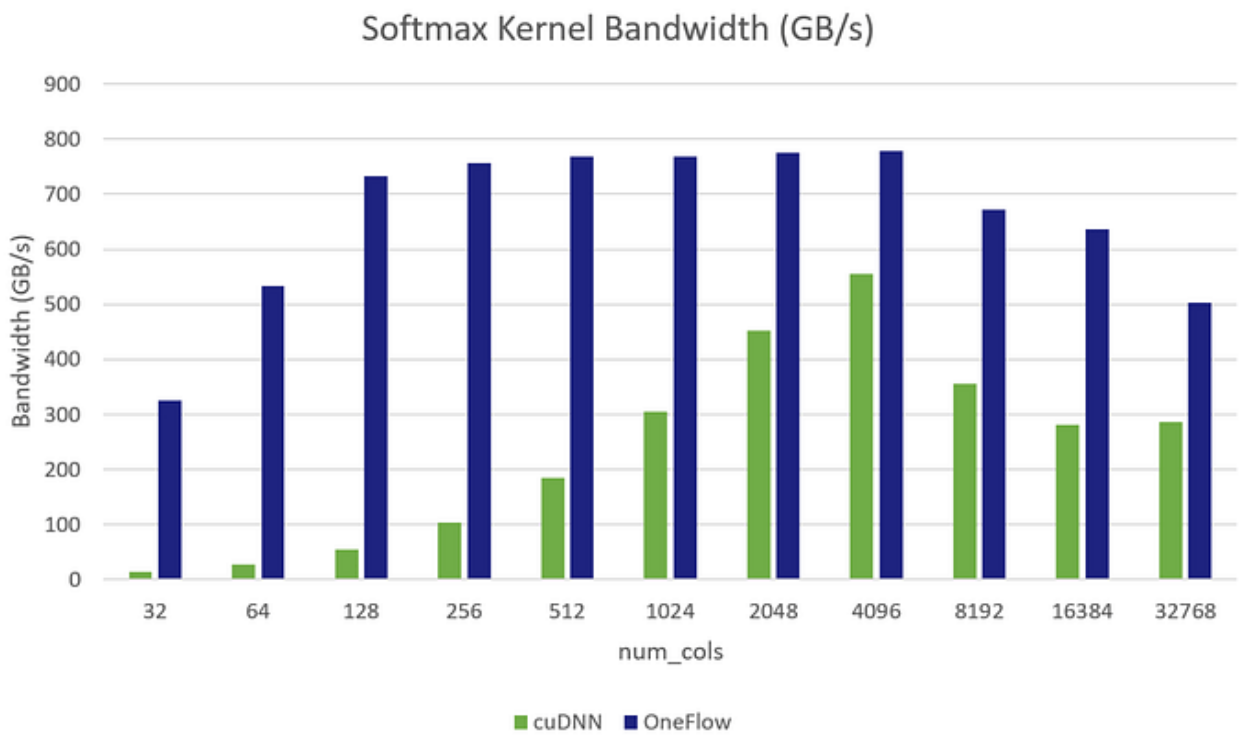
In this way, a total of `8 * D + 4 * num_rows` of access memory overhead is required. Since num_rows can be ignored compared to D, the Navie Softmax CUDA Kernel needs to access at least 8 times the data in memory, i.e.: *Naive Softmax Kernel effective memory bandwidth < theoretical bandwidth / 8.* For the GeForce RTX™ 3090 card, the theoretical bandwidth is capped at 936GB/s, so the upper bound for the Navie Softmax CUDA Kernel to utilize memory bandwidth is 936 / 8 = 117 GB/s.

In article we mention in the method Combining Kernel with Reduce computation using shared memory that the access memory of the Softmax Kernel is optimized to `2 * D`, but this is still not the ultimate optimization. With the optimizations in this article, the memory bandwidth utilization of the Softmax CUDA Kernel in OneFlow can approach the theoretical bandwidth in most scenarios.

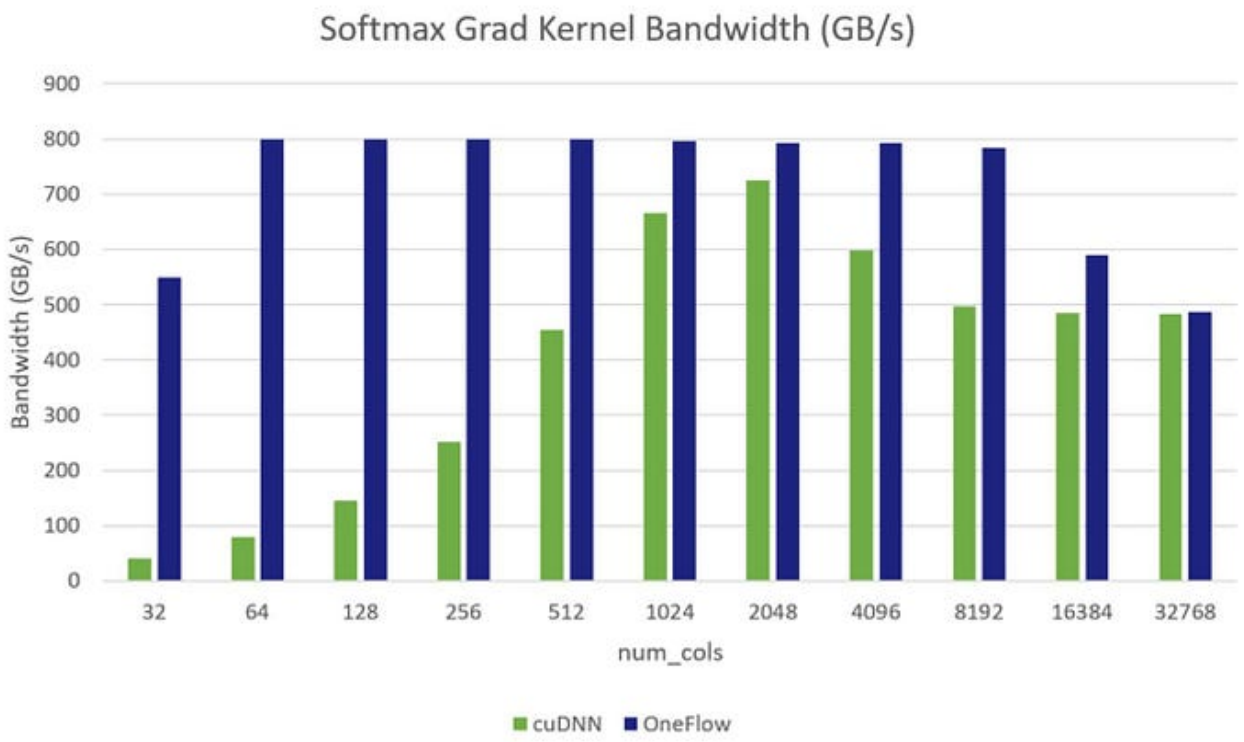
Comparison of OneFlow and cuDNN

We conducted a comparative test on the access bandwidth of Softmax CUDA Kernel after deep optimization of OneFlow and Softmax Kernel in cuDNN, and the test results are as follows:

Softmax Kernel Bandwidth:



Softmax Grad Kernel Bandwidth:



The test environment is a GeForce RTX™ 3090 GPU, the data type is half, and the Shape of Softmax = (49152, num_cols), where 49152 = 32 * 12 * 128, is the first three dimensions of the attention Tensor in the BERT-base network. We fixed the first three dimensions and varied num_cols dynamically, testing the effective memory bandwidth of the Softmax Forward Kernel and Inverse Kernel for different sizes from 32 to 32768. The two graphs above show that OneFlow can approach the theoretical bandwidth in most cases (around 800GB/s, which is comparable to the cudaMemcpy access bandwidth. The official published theoretical bandwidth is 936GB/s). And in all cases, the effective access bandwidth of OneFlow's CUDA Kernel is better than that of the cuDNN implementation.

OneFlow's Approach for Deep Optimization of

Softmax CUDA Kernel

The input shape of the Softmax function is :(num_rows, num_cols) and the variation of num_cols will have an impact on the effective bandwidth, since there is no universal optimization method that achieves transmission optimality in all cases of num_cols. Therefore, the SoftmaxKernel is optimized using a segmentation function in OneFlow: for different num_cols ranges, different implementations are chosen in order to achieve a high effective bandwidth in all cases. See softmax cuda kernel.

There are three implementations in OneFlow that optimize softmax in segments:

(1) A Warp processes one or two rows of computation for the case num_cols ⩽ 1024.

> *32 threads executing in parallel on the hardware are called a warp, and 32 threads of the same warp execute the same instruction. A warp is the basic unit of GPU scheduling and execution.*

(2) A block processes one row of computation and uses Shared Memory to store the intermediate result data, for cases where the required Shared Memory resources meet the bootable condition of Kernel Launch, which in this test environment is 1024 < num_cols ⩽ 4096.

(3) A Block processes a row of computation without using Shared Memory, and reads input x repeatedly, for cases where (1) and (2) are not supported.

Each of the three implementations is described below, using the forward computation as an example:

Implementation 1: A Warp Processes One Row of Elements

Each Warp processes one or two rows of elements, and the Reduce operation for each row needs to be a Reduce operation within the Warp. Global Max and Sum operations are performed between threads in Warp by implementing WarpAllReduce, using the Warp level primitive __shfl_xor_sync. The code is as follows.

The SoftmaxWarpImpl implementation has the following template parameters:

LOAD, STORE represent the input and output respectively, using `load.template load<pack_size>(ptr, row_id, col_id);` and `store.template store<pack_size>(ptr, row_id, col_id);` for read and write. There are two advantages using LOAD and STORE:

1. You only need to care about the compute type ComputeType in the CUDA Kernel, not the specific data type T.
2. Only a few rows of code need to be added to quickly support Softmax and other Kernel Fuse, reducing bandwidth requirements and improving overall performance. While ordinary SoftmaxKernel uses DirectLoad and DirectStore directly, FusedSoftmaxKernel such as FusedScaleSoftmaxDropoutKernel only needs to define a ScaleLoad structure and a DropoutStore structure for the input x for Scale pre-processing and output y for Dropout post-processing.

ComputeType represents the computation type. pack_size represents the number of elements in a pack for a quantized access operation, where we pack several elements to read and write to improve bandwidth utilization. cols_per_thread represents the number of elements processed by each thread.

thread_group_width represents the width of the thread group that processes the element. When cols > pack_size * warp_size, thread_group_width is warp_size, i.e. 32. When cols < pack_size * warp_size, 1/2 warp or 1/4 warp is used to process the elements of each row, depending on the cols size. With a smaller thread_group_width, WarpAllReduce requires fewer rounds to be executed.

rows_per_access represents the number of rows each thread_group processes at once. When the cols are small and the thread_group_width is not warp_size 32, if rows is divisible by 2, we let each thread process 2 rows to increase instruction parallelism and thus improve performance.

padding represents whether the padding is currently done. If cols is not a integer multiple of warp_size, we will padding it to the nearest integer multiple.

algorithm represents the algorithm used, the options are Algorithm::kSoftmax or Algorithm::kLogSoftmax.

The main loop logic of the CUDA Kernel execution is as follows. Firstly, the cols_per_thread to be processed by each thread is calculated based on the num_cols information, each thread is allocated a register of size `rows_per_access * cols_per_thread`, the input x is read into the register, and all subsequent computations are done from the register read.

In theory, Warp is the fastest way to process a row of elements, but since registers need to be used to cache the input x, but register resources are limited, and the shared memory method using method (2) is fast enough for num_cols>1024, therefore, Warp is used only when num_cols⩽1024.

Implementation 2: A Block Processes One Row of Elements

A Block processes a row of elements, and the row Reduce requires a Reduce operation within the Block, which needs to be a synchronisation of threads within the Block, using BlockAllReduce to complete the Global Max and Global Sum operations between threads within the Warp. BlockAllReduce is implemented using Cub's BlockReduce method, with the following code:

The template parameters of the CUDA Kernel are described in detail in (1), and the logic of the execution loop is as follows. The size of the Shared Memory required is calculated from num_cols as the Launch Kernel parameter, the input is saved with the help of Shared Memory, and subsequent computations are read directly from Shared Memory.

As the Shared Memory resources within the SM are also limited, the kernel will fail to boot when num_cols exceeds a certain range and the Shared Memory request exceeds the maximum limit at boot time. For this reason, the Shared Memory scheme is only used when the call to

cudaOccupancyMaxActiveBlocksPerMultiprocessor returns a value greater than 0.

In addition, it should be noted that because the threads within a Block have to be synchronized, when a Block being executed in the SM reaches the synchronization point, the executable Warp in the SM gradually decreases. And if there is only one Block executing at the same time, the executable Warp in the SM will gradually decrease to 0 at this time, which will lead to a waste of computational resources. If there are other Blocks executing at the same time, there are still other Blocks that can be executed when one Block reaches the synchronization point. The smaller the block_size, the more Blocks the SM can schedule at the same time, so the smaller the block_size the better in this case. However, if the number of Blocks that the SM can schedule at the same time remains the same when the block_size is increased, the larger the block_size, the better the parallelism. Therefore, when choosing block_size, the code calculates cudaOccupancyMaxActiveBlocksPerMultiprocessor for each block_size, and if the results are the same, the larger block_size is used.

Implementation 3: A Block Processes One Row of Elements, Does not Use Shared Memory and Reads the Input x Repeatedly

As with implementation 2, implementation 3 is still a Block processing a row of elements. The difference is that instead of caching input x with Shared Memory, it re-reads input x each time it is computed. This implementation has no limit on the maximum num_cols and can support arbitrary sizes.

Also, it is important to note that block_size should be as large as possible in this implementation. The larger block_size is, the fewer Blocks are executed in parallel in the SM, the less cache is required, and the more chances there are to hit the cache. Multiple reads of x do not access Global Memory multiple times, so in real-world tests, the effective bandwidth would not be reduced by three times by reading x three times while the Cache is available.

General Optimization Techniques

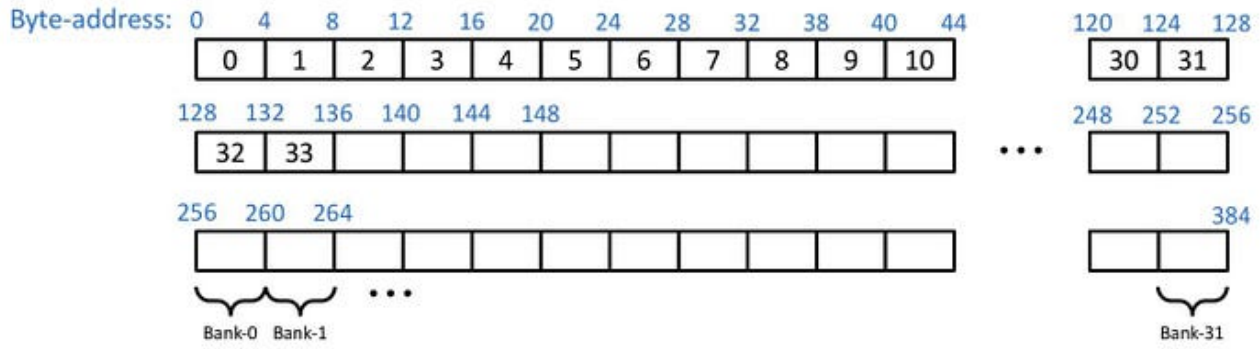
In addition to the segmentation optimization techniques for softmax described above, OneFlow uses a number of general optimization techniques not only in softmax implementations but also in other kernel implementations. Here are two kinds:

1. Pack Half types into Half2 for access, increasing instruction transfer without changing latency, similar to CUDA template for element-wise kernels optimization.

2. Bank Conflicts in Shared Memory.

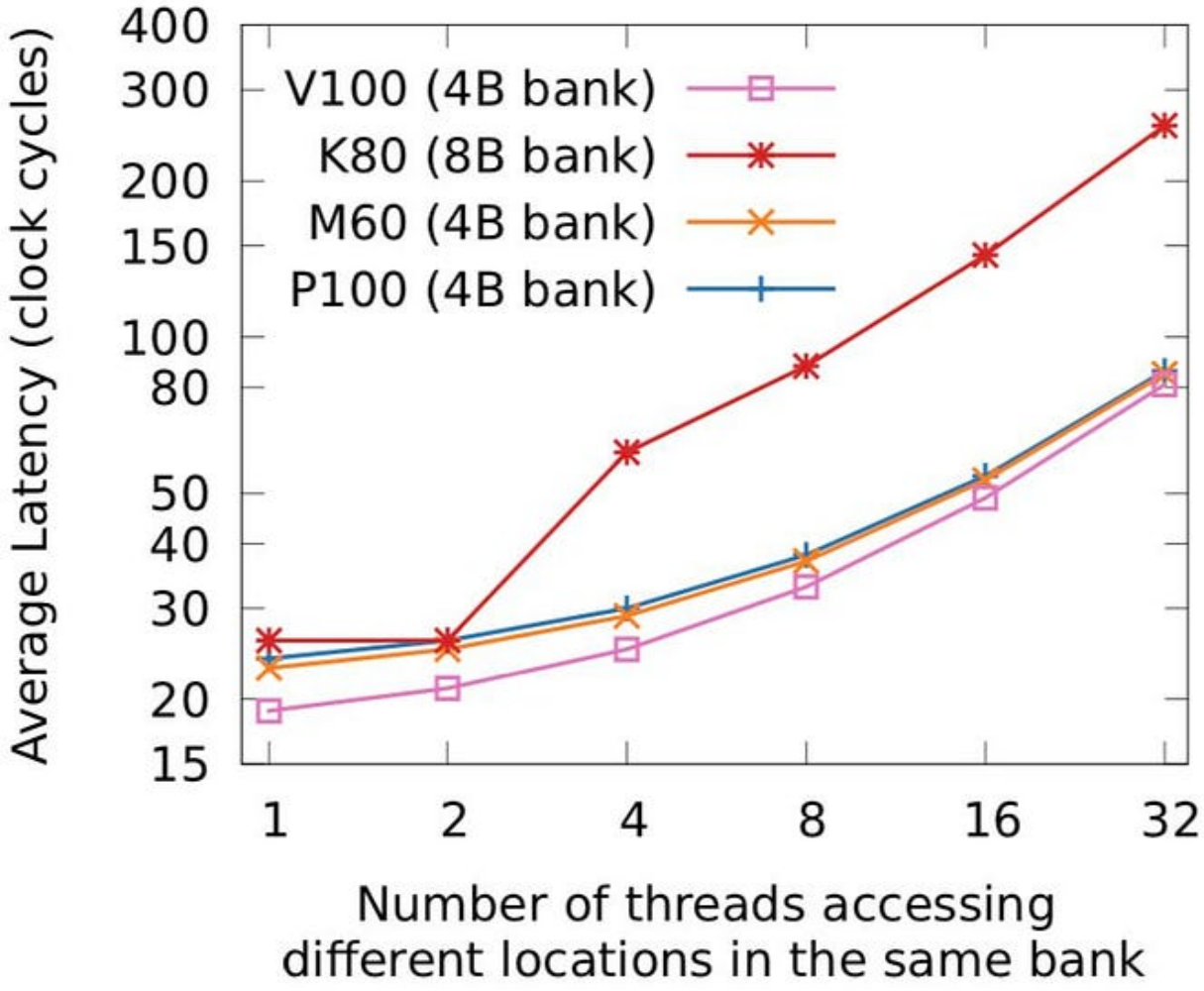
CUDA divides Shared Memory into 32 banks of 4 bytes or 8 bytes in size. For the Volta architecture it is

4 bytes, and the figure below shows 4 bytes as an example, with addresses 0-128 bytes in banks 0-31 and 128-256 in banks 0-31 respectively.



> *Note: This figure and the following Bank Conflicts figure are from VOLTA Architecture and performance optimization*

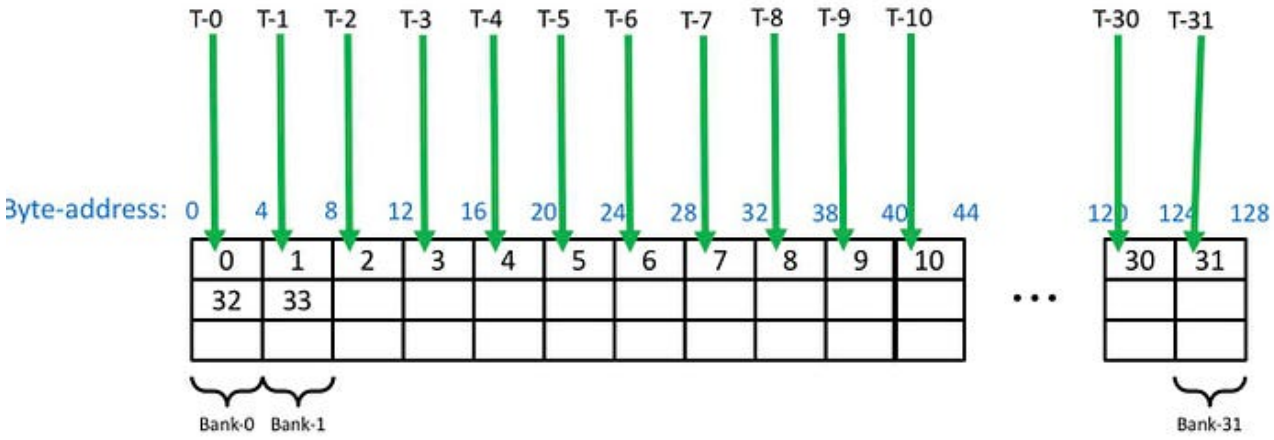
Bank Conflicts occur when different threads within a Warp access different addresses in the same bank. When Bank Conflicts occur, threads can only be accessed sequentially to increase latency. The figure below shows the latency when n threads in a Warp access different addresses in the same bank at the same time.



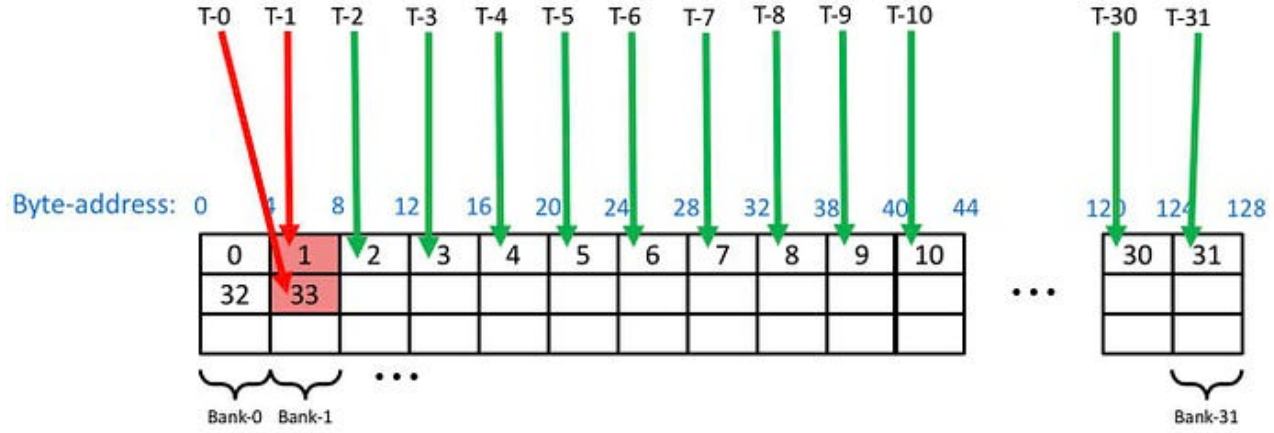
Note: Figure from [Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking]

Several situations for Bank Conflicts:

No Bank Conflicts



2-way Bank Conflict



If each thread in a Warp reads 4 bytes and accesses sequentially, there will be no Bank Conflicts. If each thread in a Warp reads 8 bytes, then the address accessed by thread 0 in Warp is in the 0th and 1st bank, the address accessed by thread 1 is in the 2nd and 3rd bank. Similarly, thread 16 accesses the address in the 0th and 1st bank, and the thread 0 accesses a different address in the same bank, which generates Bank Conflicts.

In the previous implementation (2), when assigning values to Shared memory, the following method would

generate Bank Conflicts when pack size=2 and each thread writes two consecutive 4-byte addresses.

Therefore, in implementation (2), a new memory layout is used for Shared memory to avoid different addresses of the same bank accessed by the same Warp to avoid Bank Conflicts.

References:

Using CUDA Warp-Level Primitives | NVIDIA Developer Blog

CUDA Pro Tip: Increase Performance with Vectorized Memory Access

Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking

VOLTA Architecture and performance optimization

I hope this article will help you in your deep learning projects😊. If you want to experience the functions of OneFlow, you can follow the method described in this article. If you have any questions or comments💡 about use, please feel free to leave a comment in the comments section below. Please do the same if you have any comments, remarks or suggestions for improvement. In future articles, we'll introduce more details of OneFlow.

Related articles: