**1 Introduction**

While deep learning has become popular, many famous deep learning frameworks are facing a common performance problem during model inference: frequent calls to algebraic operators seriously affect the execution efficiency of the model. Especially under the circumstance that real-time processing model require lower latency and fewer memory consumption. For example尤其是在一些需要告诉推理的模型，比如光路追踪

Generally, in order to provide better flexibility, operators in most deep learning frameworks are defined as independent algebraic kernel operations, such as convolutional kernel, matrix multiplication kernel and tensor padding kernel etc. Each operation function has a precompiled executable kernel implementation that the executor dispatches to. Moreover, common deep learning frameworks (e.g., PyTorch, TensorFlow, and Caffe2) abstract the data flow in a deep learning model as a Directed Acyclic Graph (DAG) composed of some basic operators, and the engine sequentially schedules and individually executes the kernel functions corresponding to these nodes in the certain topological order while training or inferring a model. Generally, computational frameworks have hundreds or even thousands of operators. Because of the low-level abstraction of these operators, the data flow graph of a real model often includes thousands of nodes, and the inference of these nodes on the GPU becomes thousands of kernel executions on the device. Although these lower-level precompiled kernel functions provide flexibility, their frequent invocations become a significant factor affecting the performance of many deep learning frameworks. The model performance overhead mainly comprises the scheduling overhead of the data flow, the startup overhead of GPU kernel functions, and the data transfer overhead between kernel functions.

A straightforward approach to solve this performance problem is Kernel Fusion, which fuses multiple serial kernel functions into a single kernel function, so that the entire data flow can be completed with a single function call, thus reducing the overhead of platform scheduling and kernel startup. In addition, by assigning the storage location of input and output data for different kernel functions (e.g., using shared memory or registers on the GPU), the data transfer efficiency can be greatly improved, thus enhancing the overall computational performance.

这里介绍stateoftheart fusion。XLA 书签

XLA (Accelerated Linear Algebra) is a famous kernel fusion compiler which can accelerate models and reduce the memory usage. Put differently, XLA does not execute each operation function individually, but provides a sequence of fused kernels, which are generated codes specifically for the portions of given model. These kernel fusion steps take place in the portion of computation graph where the neighboring nodes are able to be fused in an efficient manner by XLA compiler, and these neighboring nodes will be segmented into clusters called Just In Time Compilation (JIT) units. For example, the function tf.reduce\_sum(x + y \* z) launches three kernels serially without XLA: one kernel for y\*z multiplication, one kernel for the x+y\*z addition and one kernel for the reduction operation. These three neighboring nodes in the graph can be clustered and fused by XLA compiler into a single pipelined GPU kernel without even writing out the intermediate values produced by y\*z and x+y\*z to other memory location, by keeping them entirely in the GPU register.

加个图 XLA笔记(1) -- HLO IR Introduction

Despite of the fact that XLA is an effective compiler for kernel fusion, its shortages are also obvious. For most deep learning models, XLA cannot guarantee to fuse the entire network into one single kernel, since not all kernels can be transferred to XLA Intermediate Representation (IR), which is XLA’s High Level Optimizer (HLO). This means only partial intermediate nodes can be fused based on the analysis of XLA compiler. Computational graph is actually much more complex than in figure 1, and HLO fusion is doing an unsupervised clustering task on the graph. HLO provides a limited operator set but model researchers are continuously developing new operations and structures. To pursue the optimal performance, engineers tend to implement the specific fully-fused model for their own models in the industrial projects, rather than depending on XLA automatic compiler. Additionally, PyTorch-XLA supports only Google Cloud TPUs currently, with limited accessibility, since Google Cloud platform is necessary. Most importantly, the kernel fusion mechanism of XLA chooses the fusion strategy from the HLO operator set, which means the operator algorithm are pre-defined. Although new kernel codes are generated for the given model, the user cannot optimize the computation algorithm in kernel-level further.

By designing fully-fused models, the better performance can be pursued by optimizing the low-level kernel algorithm, avoiding the limitation and fixed pattern in other deep learning frameworks. This inspires many creative ideas of optimizing kernel fusion in many aspects, such as optimization based on mixed-precision, cuBLAS, CUTLASS [4], Tensor Cores [2], etc.

Among the hundreds of operations in the deep learning models, GEMM (General Matrix Multiply) plays a core building block. In the fully-connected layers, tensor operations are computed directly using GEMM, and for convolutional layers in Convolutional Neural Networks (CNN), the convolutional operations are reduced to GEMM in an efficient way, for example, these ideas are adapted by the cuDNN library [1]. Given the importance of GEMM, Nvidia introduced Tensor Cores [2] in 2017 with Volta V100 GPUs, it is available on Volta and all subsequent GPU architectures, serving as an essential dedicated hardware component.

State-of-the-art kernel fusion methods for GEMM try to use shared memory by storing the intermediate results on chip and avoiding the global memory as much as possible, because the shared memory is located on chip and has much lower latency than the global memory. However, each multiprocessor has only 64KB of on-chip memory that needs to be partitioned between L1 cache and shared memory. By default, the shared memory is set to 48KB and L1 cache is 16KB, this limits the size of intermediate values of the model.

This thesis focuses on the optimization of GEMM using Tensor Cores in kernel fusion tasks. A new method for intermediate results caching in register is raised, which shuffles between threads in a warp and avoids additional usage of shared memory. A fully-connected neural network will be taken as an example of typical fully-fused network, and more creative features, which are not yet introduced by fused network using Tensor Cores, will be implemented based on this fully-connected network and its code generation framework, such as residual blocks, flexible hidden channels and arbitrary activation functions.

2 Related Works

**2 Background**

**2.1 Nvidia GPU Memory Hierarchy**

Nvidia devices have a memory hierarchy with several spaces and levels, which all have distinct usages and unique characteristics in the application, such as global, local, shared, texture memory and register.

图示, 示意图

描述已自动生成

来自<https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#device-memory-spaces> Figure 2. Memory spaces on a CUDA device

**2.1.1 Off Chip Memory**

Dynamic Random Access Memory (DRAM) is usually considered as the GPU device memory space, which communicates with the host chipset and needs to be refreshed dynamically, otherwise the data on DRAM capacitors will be lost.

Global memory is the most plentiful space on the whole device, however, global, local and texture memory have the greatest access latency, followed by constant memory[toolkit], because DRAM is placed on the outside of the GPU chip, in this way all the memory units on DRAM are named as off chip memory. Global memory typically uses parallelization to increase the speed of data access, known as throughput. Each time a device accesses the data on a location in DRAMs, a series of contiguous locations, including the target location, are actually being accessed. These consecutive locations accessed and transferred in the DRAMs are called DRAM bursts. To utilize this characteristic of global memory bursts, the optimal thread access on global memory should be in a coalesced pattern, where the contiguous threads load the values on the contiguous global memory locations parallelly, in this way the device can merge all these requests into as few as possible access request to contiguous locations using bursts. Each merged request is called transaction, during the implementation, a good global memory access pattern needs try to minimize the number of necessary transactions, to improve the instruction throughput accordingly. By this idea, all the data flow in this thesis between global memory and GPU threads are arranged as coalesced pattern with data storing on global memory in 1D arrays. When parallel data accessing of size 1, 2, 4, 8, or 16 bytes that are aligned, compiler translates this to one global memory instruction. But if size and alignment requirements are not met, compiler would translate it to multiple global memory access instructions. For 2D or 3D array allocation, CUDA provides a force alignment operation \_\_align\_\_(n) so that every row starts at a n-byte boundary address. But force alignment on 2D or 3D array may waste some memory space, therefore, 1D pattern data storing in the arrays should be considered with the highest priority if possible, especially when memory size is usually the bottleneck in kernel fusion tasks, that handle with millions of batches input data, pursuing a maximal usage of GPU memory.

Local memory is named of its scope to the threads. Each thread has private local memory and register. Variables that cannot allocated in register will be arranged in local memory off chip. If there are too many variables in a thread that exceed the space of registers, then the registers will overflow and local memory will be needed to store them, thus the program will run slower. Since local memory is considered as a backup space of register, kernel fusion methods normally do not optimize the program based on local memory.

**2.1.2 Constant and Texture Memory**

On chip memory is the unit cached in GPU chip, providing higher effective bandwidth by decreasing the memory accesses to off chip DRAM. Kernel fusion is considered to maximize the data streaming on chip and avoid data flow through off chip memory as much as possible. However, the faster the access speed, normally the smaller the storage space. This is the common bottleneck of the kernel fusion methods.

文本

描述已自动生成

Figure on chip memory structure

Constant memory is constrained to be read-only and can save bandwidth when compared to accessing the same data from off chip global memory. Because the read-only cached memory can broadcast to many threads when these threads are accessing the same address without incurring any additional memory traffic. Access to different addresses by threads within a warp is serialized, so the cost increases proportionally to the number of unique addresses read by all threads within a warp. Constant memory on Nvidia GPU is normally 64KB, which is not much larger than shared memory on chip.

Texture memory is another variety of read-only space that can allocate much more space than constant memory. Texture memory is widely used not only in graphics processing but also in general-purpose computing. Allocating an array from GPU global memory to texture memory using Texture Object API is called binding, it specifies a cache region being used as texture memory and reorganize the data to optimize the read-only pattern. Then a texture object can be fed into the kernel function, where the texture array elements can be accessed by the build-in instruction tex2D<type>(texureObject, index1, index2). The array indices in texture memory can be an integer or normalized floating point number between 0 and 1, a 2D normalized coordinate, representing the relative spatial location in an array. For graphics application, texture memory is specifically designed for the data processing pattern with spatial locality, such as interpolation, rotation or filtering of an image. The texture cache is optimized for 2D spatial localization, so texture accessing will get the best performance if threads in the same warp read spatially nearby texture addresses. Texture memory handles especially well with the computing of an element with its the spatial neighbors, this attribute can accelerate the image processing by design but is not as commonly used in general computing. In the matrix multiplication framework Warp-Level Matrix Multiply and Accumulate (WMMA) the index is always an integer and the matrix is loaded referring to one single pointer of an array. So, texture memory is not suitable for WMMA API, since each element on texture memory must be accessed using index with Texture Object API, it doesn’t have the mechanism to load a matrix based on the pointer of starting element and arrange the successor elements automatically.

**2.1.3 L2 cache memory**

L2 cache is on-chip, which potentially provides higher bandwidth and lower latency accesses to global memory. L2 cache is also larger than other on chip memory, usually with 5MB space. So L2 cache is considered to be a potential optimization method for kernel fusion tasks. However, during the experiment of assigning the weights matrix to L2 cache, not obvious improvement is observed. This will be discussed in the experiment chapter. On the GPU chip, a portion of L2 cache memory can be set aside for access to a data region in global memory. Instruction stream\_attribute.accessPolicyWindow.base\_ptr = reinterpret\_cast<void\*>(ptr) will stream a window starting from the pointer to L2 cache. This set-aside portion is designed to be later used by persistent accesses, which means that the data on L2 cache will not be accessed only once but frequently during the program. In the kernel fusion of a fully-connected network, the input matrix will be access only once, but the weight matrices can be access multiple times, since the device may compute millions batches of input vectors in several serial loops, in this way, the weight matrices of each layer will be accessed multiple times. L2 cache can potentially benefit the persistent data access of weight matrices. One point to note is that the sliding window size needs to fit the persistent data to achieve better performance. With an appropriate hit-ratio the performance can be increased by 50% according to the CUDA best practices guide, however, once the persistent data exceeds the size of L2 cache portion, the performance is observed with 10% decrease.

**2.1.4 Shared Memory**

Shared memory has much higher bandwidth and lower latency than local and global memory in the case of no bank conflicts between threads. It enables cooperation between the threads in the same block, since each block has 48KB shared memory by default, which can be allocated using the \_\_shared\_\_ memory space specifier. Shared memory is suitable for exchanging intermediate results with a relatively small size. Take half-precision as an example, each half-precision float16 data spans 2 bytes, and 48 KB shared memory has a space of 48 \* 1024 = 49152 bytes, which are exactly 24576 half-precision values.

|  |  |  |
| --- | --- | --- |
| Hidden channels | Batch size in a warp | Warp number in a block |
| 128 | 64 | 3 |
| 128 | 32 | 6 |
| 128 | 16 | 12 |
| 128 | 8 | 24 |
| 64 | 32 | 6 |
| 64 | 16 | 24 |

这个表最好放在实验部分。

If a fully-connected network with 128 hidden channels, the intermediate results in each warp have dimension size of 128 multiplied with batch size in each warp.

**2.1.2 Bank Conflicts**

Global memory accessing doesn’t suffer bank conflicts, since memory bursts on global memory always group the access together to minimize the request. However, shared memory can be access by the threads individually in a warp. Unlike the global memory, the shared memory is arranged as memory banks with equal size which can be accessed simultaneously, to improve access concurrency. This means if threads are loading or storing of data in n different addresses that locate in n distinct shared memory banks, these banks can be accessed parallelly. On the contrast, if one bank address is requested by multiple threads, these requests will be executed sequentially. Only one exception is that when multiple threads are accessing the same address, the bank controller can broadcast the value to all threads.

In each block, the 48KB shared memory is divided into 16 banks, and there’re 32 threads in a warp divided into 2 group with 16 threads, this is named half-warp. Take an arrays of 128 int values as example, the bank arrangement is as figure bellow:

表格

描述已自动生成

|  |  |  |
| --- | --- | --- |
| Property | Value | Instruction line (cudaDeviceProp) |
| Device name | GeForce RTX 2080 Ti | name |
| Clock rate | 1.54 GHz | clockRate |
| Memory bus width | 352 bits | memoryBusWidth |
| Global memory | 11019 MB | totalGlobalMem |
| Constant memory | 64 KB | totalConstMem |
| Texture memory size 1D | 131072 | maxTexture1D |
| L2 cache size | 5.5 MB | L2cacheSize |
| Shared memory per block | 48 KB | sharedMemPerBlock |
| Register per block | 65536 x 32-bits | regsPerBlock |
| Threads per block | 1024 | maxThreadsPerBlock |
| Warp size | 32 | warpSize |
| Number of SM | 68 | multiProcessorCount |
| Maximum block dimension | 1024 x 1024 x 64 | maxThreadsDim[] |
| Maximum grid dimension | 2147483647 x 65536 x 65536 | maxGridSize[] |

**2.1 Warp-Level GEMM**

Tensor Cores are delivered

**2.2 Tensor Cores**

Tensor Cores are delivered by Nvidia specifically for computationally intensive tasks such as fully-connected and convolutional layers.

Workloads must use mixed precision to take advantage of Tensor Cores, because Tensor Cores can only be activated when certain elements in a layer are divisible by 8 or 16 (for FP16 data or INT8 data) [3]. In addition, a fully-connected layer that follow this rule for batch size and number of input and output can activate Tensor Cores, and similarly, convolutional layers for number of input and output channels is doing the same. Layers that cannot meet this requirement will use 32-bit CUDA cores instead of Tensor Cores, which is a fallback option without acceleration by comparison. By this setup, all inputs and outputs to Tensor Core in this thesis are set to half precision, all numbers of batch size and weights dimensions are set to be divisible by 16.

The tensor cores are exposed as Warp-Level Matrix Operations in the CUDA C++ API：

[1] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro,

and E. Shelhamer, “cuDNN: Efficient primitives for deep learning,”

arXiv preprint arXiv:1419.0759, vol. abs/1410.0759, 2014.

[2] NVIDIA. (2018) NVIDIA Tesla V100 GPU architecture. [Online]. Available: <https://images.nvidia.com/content/voltaarchitecture/pdf/volta-architecture-whitepaper.pdf>

[3] NVIDIA. Automatic Mixed Precision for Deep Learning. [Online]. Available: <https://developer.nvidia.com/automatic-mixed-precision>

[4] K. Andrew, M. Duane, D. Julien, and T. John. (2018) Cutlass. [Online].

Available: <https://github.com/NVIDIA/cutlass>