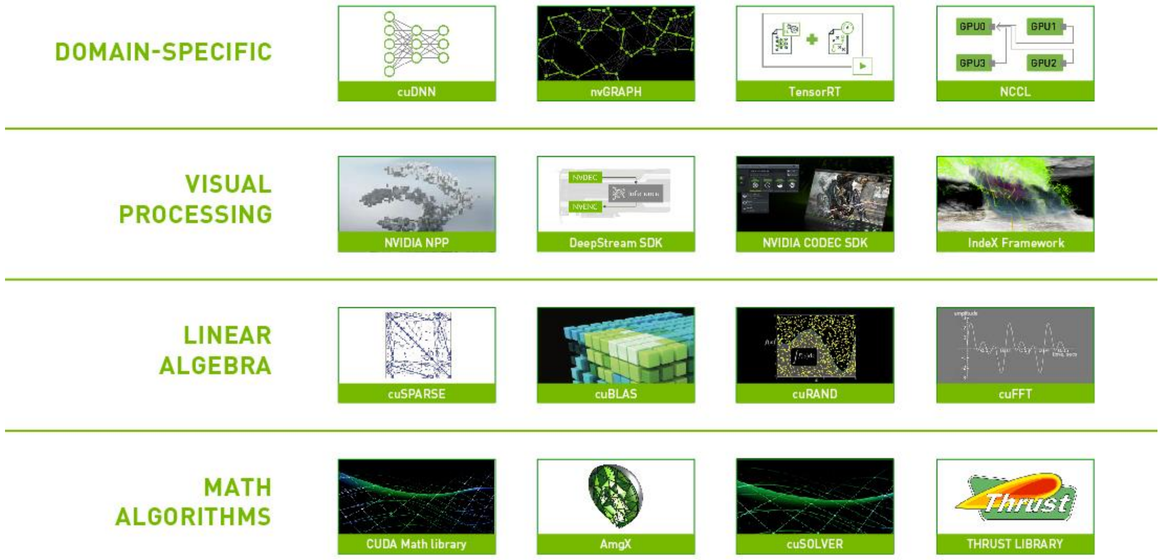
ADDING CUSTOM CUDA C++ OPERATIONS IN TENSORFLOW FOR BOOSTING BERT INFERENCE

Minseok Lee, Developer Technology Engineer, 2nd July

CUDA

NVIDIA’s Parallel Computing Platform and Programming Model

- Language Integration – C/C++, Fortran, …
- Integrated Development Environment
- Doman Specific Libraries
- High Performance



TENSORFLOW

An End-to-End Open Source Deep Learning Framework

- Easy/Flexible Model Building based on Python and Keras
- Robust ML Production Anywhere
- GPU Accelerated Performance based on **CUDA**

TensorFlow

CUDA KNOWLEDGE + TENSORFLOW

Customized Performance Synergy

- Help analyze and understand GPU-related behavior

  e.g., Am I fully utilizing my GPU(s)? If not, what is the bottleneck?

- Enable to tune and squeeze training/inference performance

  e.g., Increase the parallelism of CUDA kernel mapped to a TF Op

  e.g., Implement a new optimized operation for your case

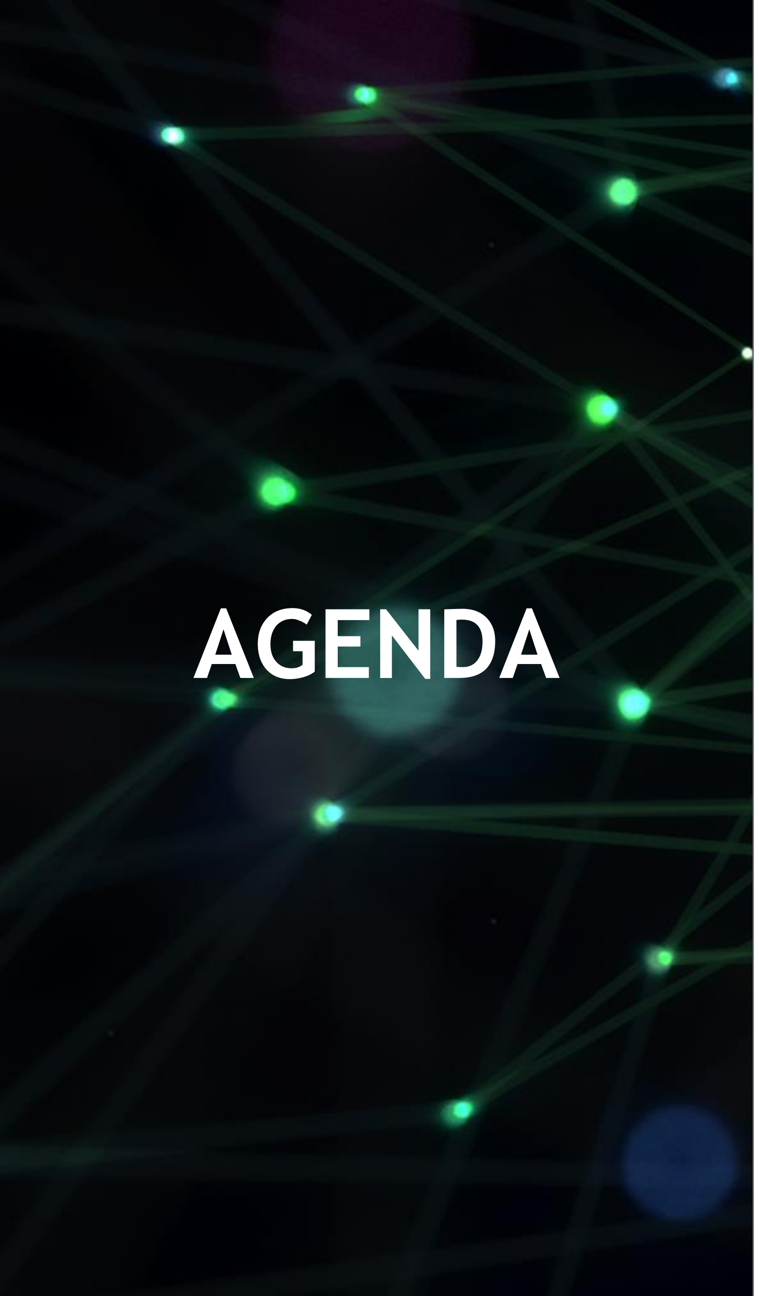
- It sounds great, but how can it be enabled?

TensorFlow

+

NVIDIA

CUDA



AGENDA

- What is TensorFlow Custom Op
- Case Study: BERT SQuAD Inference
- Tips and Other Options

TENSORFLOW CUSTOM C++ OP

Interface to Add New Operations beyond Existing TensorFlow Library

Motivation:

- Difficult/Impossible to express your operation as a composition of existing ones
- The composite one doesn't have decent performance
- The existing op is not efficient for your use case

Custom C++ Op is one of the sensible options to customize TF's feature and performance

CUSTOM C++ OP INCORPORATION

Bob Ross Style Guideline – That Easy, Right?

1. Define (or Register) Op’s interface in C++ (op’s name, input/output and their shapes, …)

2. Implement Op (or Kernel) in **CUDA** C++ (override OpKernel::Compute to call the kernel)

3. Implement Gradient in Python (not necessary if you only focus on Inference)

4. Build its shared library and use it in your Python code

CASE STUDY:
BERT SQUAD INFERENCE

WHY JUMPING INTO BERT SUDDENLY?

To Apply Custom Ops to BERT

- To provide a pragmatic example rather than a boring “Hello, World!” style example

- Transformer and BERT are being hyped everywhere nowadays

WHAT IS BERT?

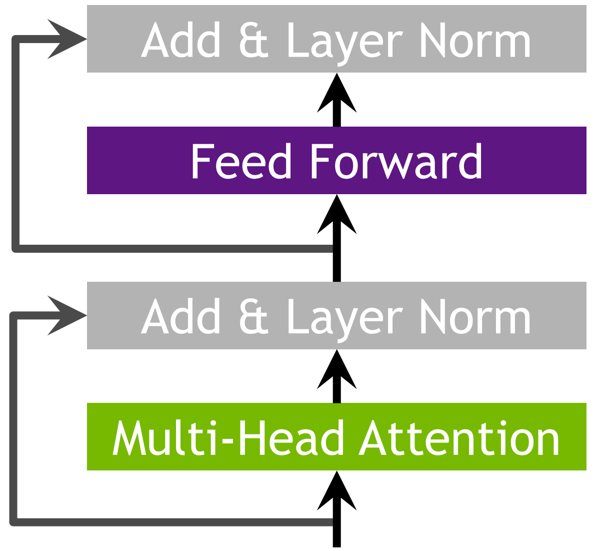
Bidirectional Encoder Representations from Transformers

A new method of pre-training language representations for a wide array of NLP tasks

Model Architecture is a multi-layer bidirectional Transformer encoder which embraces

- Multi-Head Attention
- Fully Connected Feed Forward with a GELU activation

  "Intermediate" sub-layer in the code
- Residual Connections



TARGET CONFIGURATION

Let’s focus on BERT SQuAD Inference Case

Batch size and sequence length can be varied across difference tasks and environments

- Based on what you want, the best optimization approach can be varied

BERT-Large checkpoint fine tuned for SQuAD is used

- 24-layer, 1024-hidden, 16-head
- max_seq_length: 384, batch_size: 8 (default from NVIDIA GitHub repo)

For the sake of simplicity, **only the inference case** is covered

FIRST CUSTOM OP: GELU

GELU ACTIVATION FUNCTION

Why and How to Make its Custom Op

Google's Implementation in modeling.py

```
def gelu(x):
  cdf = 0.5 * (1.0 + tf.tanh(
      (np.sqrt(2 / np.pi) * (x + 0.044715 * tf.pow(x, 3)))))
  return x * cdf
```

Single input, single output function, e.g., out[4] = gelu(in[4])

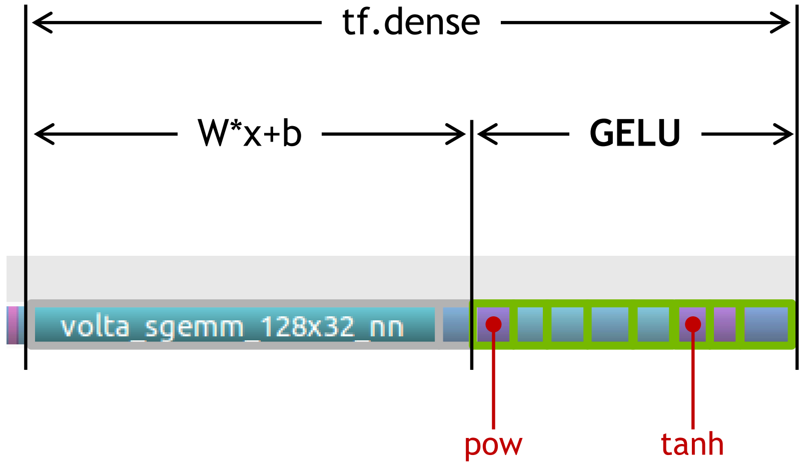
- Easy to write in Python by compositing existing TF ops

But how about its performance? How many CUDA kernels does it execute?

- Let's trying profiling!

PROFILING GELU

Result based on NVIDIA Visual Profiler (NVVP)

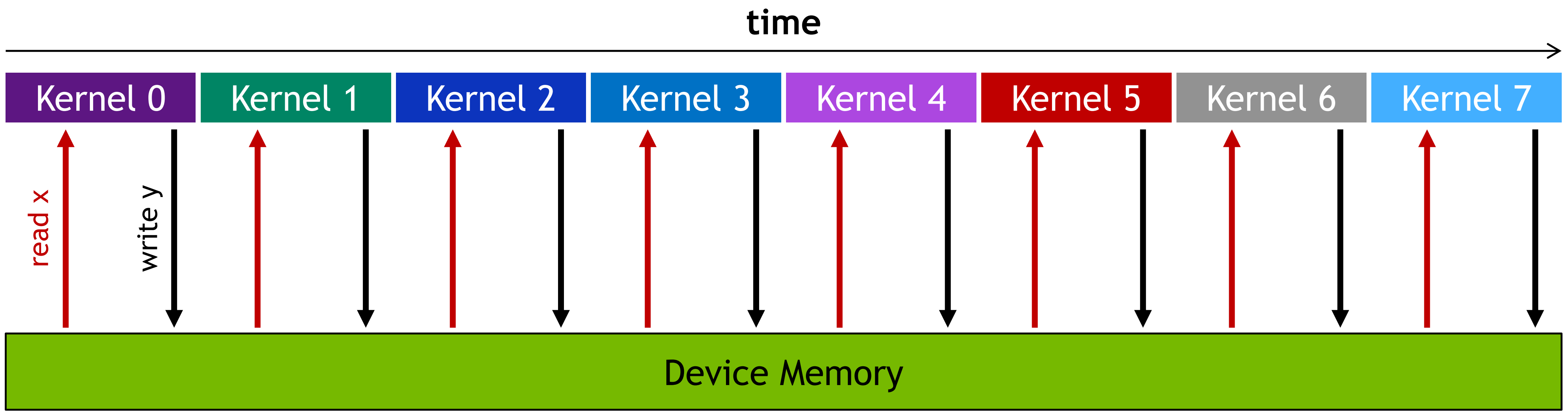


GELU activation in Python results in **8 CUDA kernels** in C++

- Their aggregated runtime is almost similar to W*x+b!

PERFORMANCE ANALYSIS

Why Multi-Kernel GELU is So Slow?



Each kernel reads the input array x and writes the output array y

- Total 8 reads and 8 writes for the same arrays!
- What if we can read and write once? **Kernel fusion**

STEP 1. REGISTER OP'S INTERFACE IN C++

Specify Name, Inputs, Outputs, Attributes and etc

gelu_op.cc

```
#include "tensorflow/core/framework/op.h"
#include "tensorflow/core/framework/shape_inference.h"

namespace tensorflow {

using GPUDevice = Eigen::GpuDevice;

REGISTER_OP("GeluOp") Op name
  .Attr("T: type") Attr name: “T”, type: “type”
  .Input("in: T") Input name: “in”, type: “T”
  .Output("out: T") Output name: “out”, type: “T”
  .SetShapeFn([](shape_inference::InferenceContext *c) {
      c->set_output(0, c->input(0)); set 0th output shape to 0th input shape
      return Status::OK();
   });

} // namespace tensorflow
```

STEP 2.1. INHERIT OPKERNEL IN C++

Override **Compute** function to do CUDA calls

gelu_op.cc

```
template <typename Device, typename T>
class GeluOp : public OpKernel {
 public:
  explicit GeluOp(OpKernelConstruction* context): OpKernel(context) {}
  void Compute(OpKernelContext* context) override {
    const Tensor& input_tensor = context->input(0)
```

Get 0th input tensor

```
    Tensor* output_tensor = nullptr;
    OP_REQUIRES_OK(context,
        context->allocate_output(0, input_tensor.shape(), &output_tensor))
```

Create a 0th output tensor in the same shape of 0th input

```
    const T* input_ptr = input_tensor.flat<T>().data();
    T* output_ptr = output_tensor->flat<T>().data();
    int num_elements = input_tensor.NumElements();
    ...
```

Get in/out pointers and their size

```
  }
 private:
  functor::GeluOpFunctor<Device, T> functor_;
};
```

STEP 2.1. INHERIT OPKERNEL IN C++

Override **Compute** function to do CUDA calls

gelu_op.cc

```
template <typename Device, typename T>
class GeluOp : public OpKernel {
 public:
  explicit GeluOp(OpKernelConstruction* context): OpKernel(context) {}
  void Compute(OpKernelContext* context) override {
    ...
    functor_.run(
        context->eigen_device<Device>(),
        input_ptr,
        output_ptr,
        num_elements);
  }
 private:
  functor::GeluOpFunctor<Device, T> functor_;
};
```

Do what you want by using the device, in/out pointers and parameters

STEP 2.2. REGISTER IMPLEMENTATION

Specify Device and Type Constraints

gelu_op.cc

```
#ifdef GOOGLE_CUDA

#define REGISTER_GPU(T)                                           \
    REGISTER_KERNEL_BUILDER(                                      \
        Name("GeluOp").Device(DEVICE_GPU).TypeConstraint<T>("T"), \
        GeluOp<GPUDevice, T>)
REGISTER_GPU(float);
#undef REGISTER_GPU

#endif // GOOGLE_CUDA
```

- Device = Eigen::GPUDevice
- TypeContraint<T>(“T”): attr “T” must be T
- REGISTER_GPU(float): it works only when T is float

STEP 2.3. CUDA PROGRAMMING

CUDA Kernel Configuration and Launch

gelu_op.cu.cc

```
#ifdef GOOGLE_CUDA
#define EIGEN_USE_GPU

#include "gelu_op.h"
```

The template class declaration of GeluOpFunctor

```
#include "gelu_kernel.h"

#include "tensorflow/core/framework/op_kernel.h"

namespace tensorflow {

namespace functor {

using GPUDevice = Eigen::GpuDevice;
```

Partial specialization for GPUDevice

```
template <typename T>
struct GeluOpFunctor<GPUDevice, T> {
  GeluOpFunctor() {
    int device;
    cudaGetDevice(&device);
    cudaDeviceGetAttribute(&num_devices_, cudaDevAttrMultiProcessorCount, device);
  }
  ...
};
...
```

CUDA API Call to get # SMs for the current device

STEP 2.3. CUDA PROGRAMMING

CUDA Kernel Configuration and Launch

gelu_op.cu.cc

```
...
template <typename T>
struct GeluOpFunctor<GPUDevice, T> {
  ...
  void run(const GPUDevice& d, const T* in, T* out, int n_elements) {
    GeluKernelLauncher(in, out, n_elements, num_devices_, d.stream());
  }
  int num_devices_;
};

template struct GeluOpFunctor<GPUDevice, float>;

} //namespace functor

} //namespace tensorflow

#endif
```

Pass # SMs

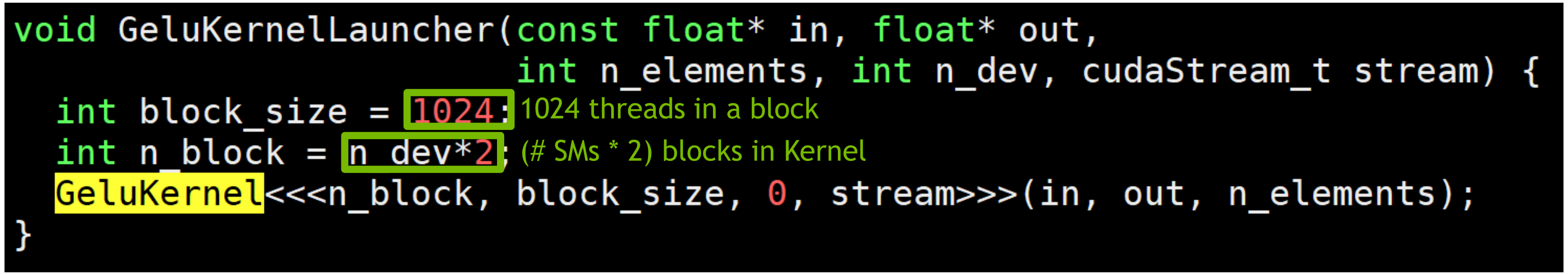
Pass CUDA Stream

Template instantiation for float

STEP 2.3. CUDA PROGRAMMING

Slight Optimization: Balance between Parallelism and Iteration

gelu_op.cu



```
void GeluKernelLauncher(const float* in, float* out,
                        int n_elements, int n_dev, cudaStream_t stream) {
  int block_size = 1024;
  int n_block = n_dev*2;
  GeluKernel<<<n_block, block_size, 0, stream>>>(in, out, n_elements);
}
```

Why We decide # threads based on # SMs, not # elements?

- A single GPU can run (# MAX threads per SM * # SMs) threads **concurrently**

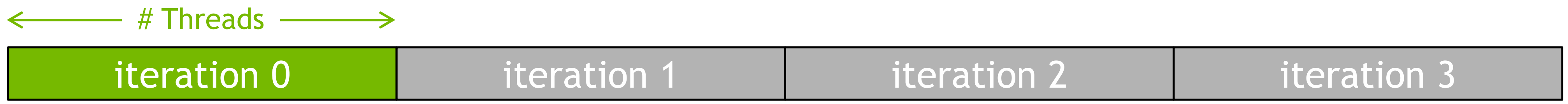
  e.g., V100 has **80 SMs** and each SM can run **up to 2048 threads** (=163,840 threads)
  If a kernel has more threads, it runs the first 163,840 threads with the others pended

- To minimize the inter-thread redundant operations, e.g., np.sqrt(2 / np.pi)

  Let's make each thread handle multiple elements

STEP 2.3. CUDA PROGRAMMING

CUDA Kernel Implementation



gelu_op.cu

```
template <typename T>
__global__ void GeluKernel(const T* in, T* out, int n_elements) {
  int gid = blockIdx.x * blockDim.x + threadIdx.x;
  int chunk_size = blockDim.x * gridDim.x;

  const T scale = sqrt(T(2) / CUDART_PI);

  for(int i=gid; i<n_elements; i+=chunk_size) {
    T x = in[i];
    T cdf = T(1) + tanh(scale * (x + T(0.044715) * (x * x * x)));
    cdf *= T(0.5);
    out[i] = x * cdf;
  }
}
```

Calculated only once per thread

Efficient than pow(x, 3)

The whole calculation is done in register

STEP 3.1. BUILD OP SHARED LIBRARY

Generate a SO file from *.cc and *.cu

No matter how you build the code, e.g., CMake, clearly specify the following information

- Tensorflow header/library file location

  e.g., tf.sysconfig.get_include() or /usr/local/lib/python3.5/dist-packages/tensorflow/include

- Library dependencies

  e.g., -lcublas, -lcudart, -tensorflow_framework

- -D_GLIBCXX_USE_CXX11_ABI=0
  (Omitting it leads to undefined symbol error for GCC >= 5.0)

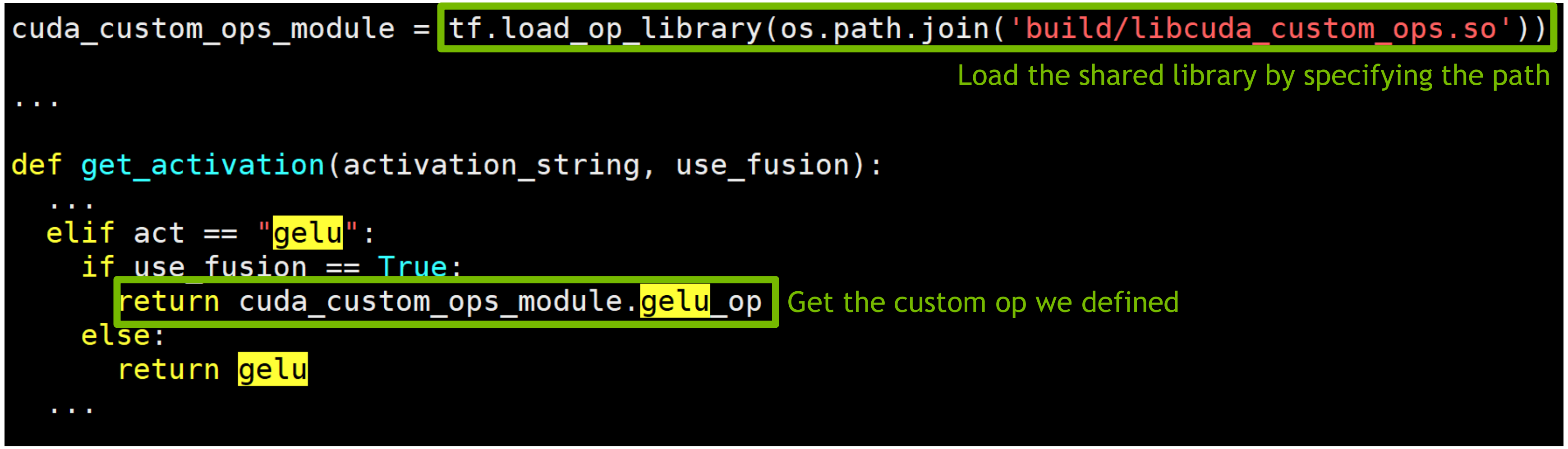
- -DGOOGLE_CUDA=1

- --std=C++11 --expt-relaxed-constexpr --expt-extended-lambda

STEP 3.2. LOAD AND USE OP IN PYTHON

How to Bind C++ code to Python code

modeling.py



```
cuda_custom_ops_module = tf.load_op_library(os.path.join('build/libcuda_custom_ops.so'))
...

def get_activation(activation_string, use_fusion):
  ...
  elif act == "gelu":
    if use_fusion == True:
      return cuda_custom_ops_module.gelu_op
    else:
      return gelu
  ...
```

Load the shared library by specifying the path

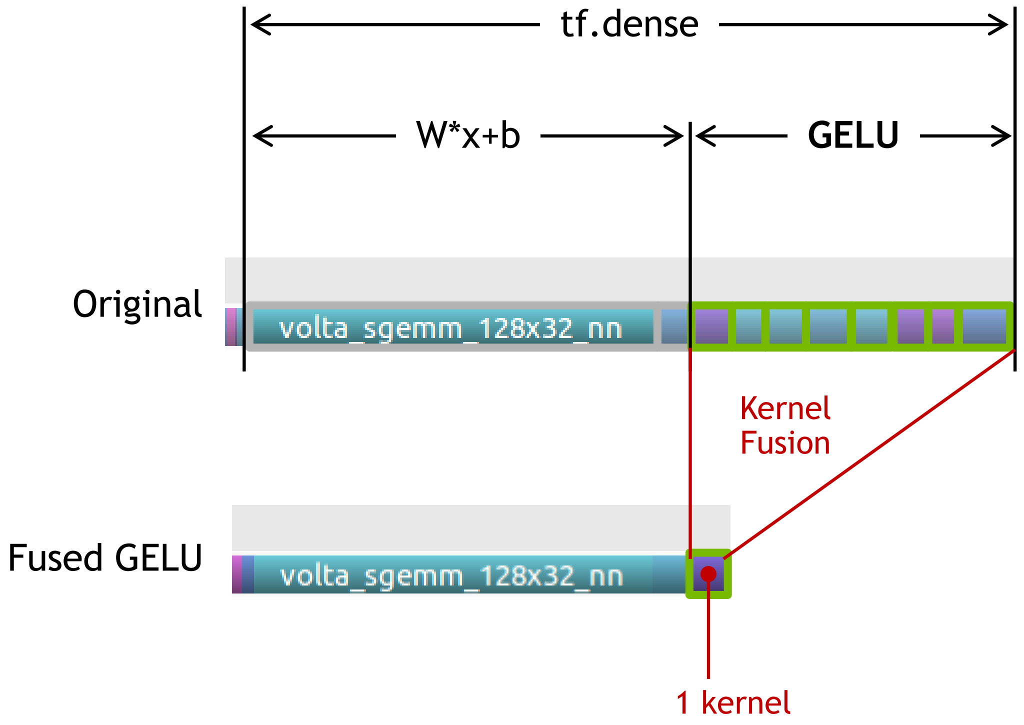
Get the custom op we defined

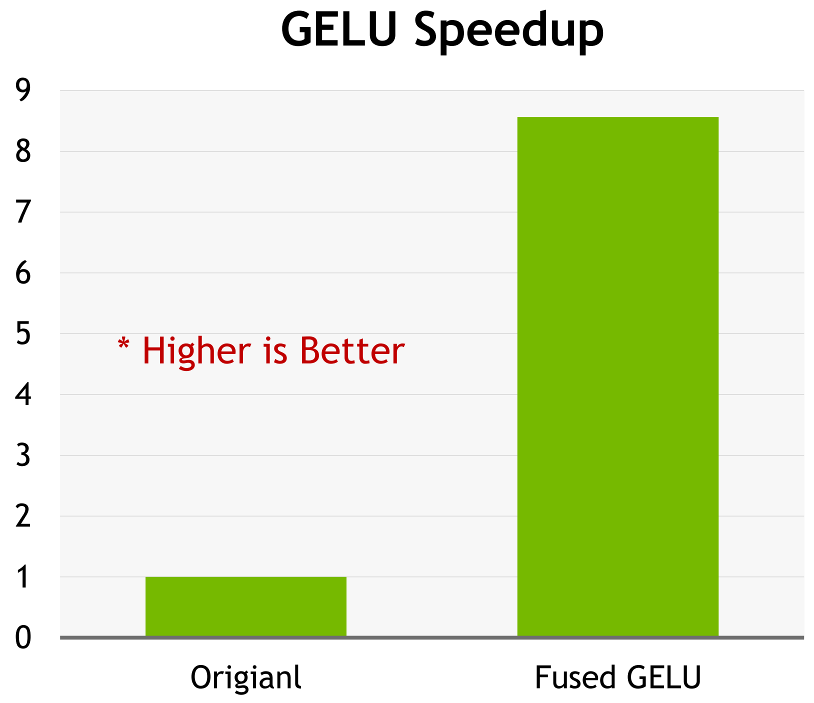
The relationship between Python op name and C++ op name

- CamelCase in C++ to snake_case in Python
- e.g., if C++ op name is **GeluOp**, Python op name is **gelu_op**

PROFILING FUSED GELU

Result based on NVIDIA Visual Profiler (NVVP)

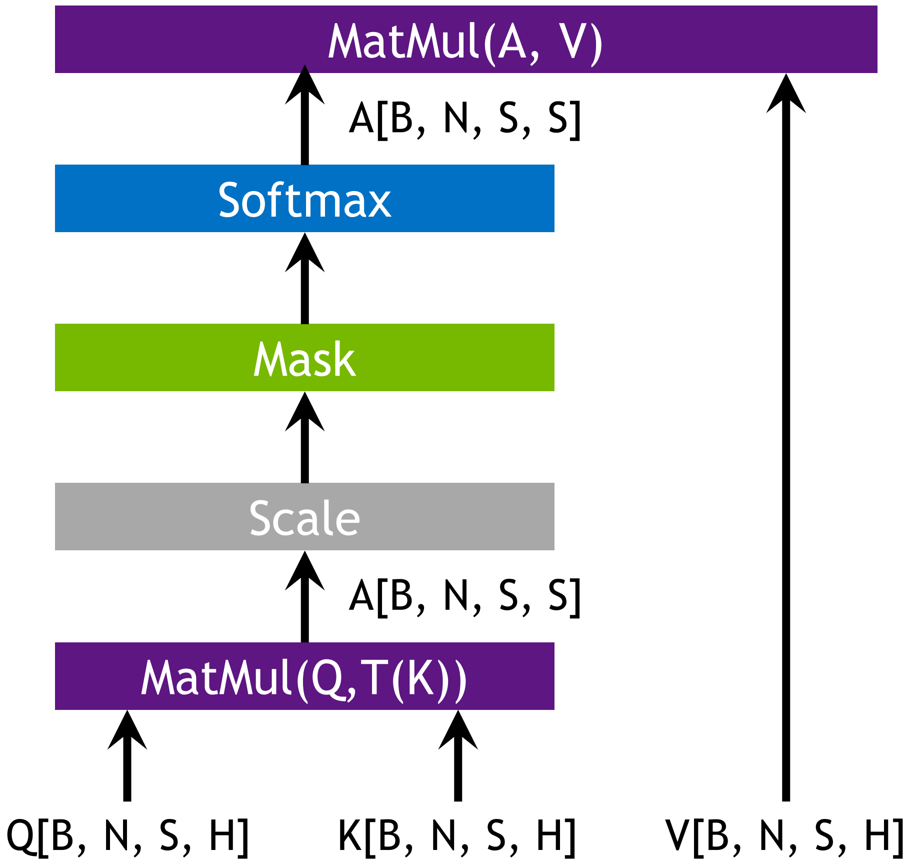




SECOND CUSTOM OP: MULTI-HEAD ATTENTION

MULTI-HEAD ATTENTION

Most Important Function in Transformer



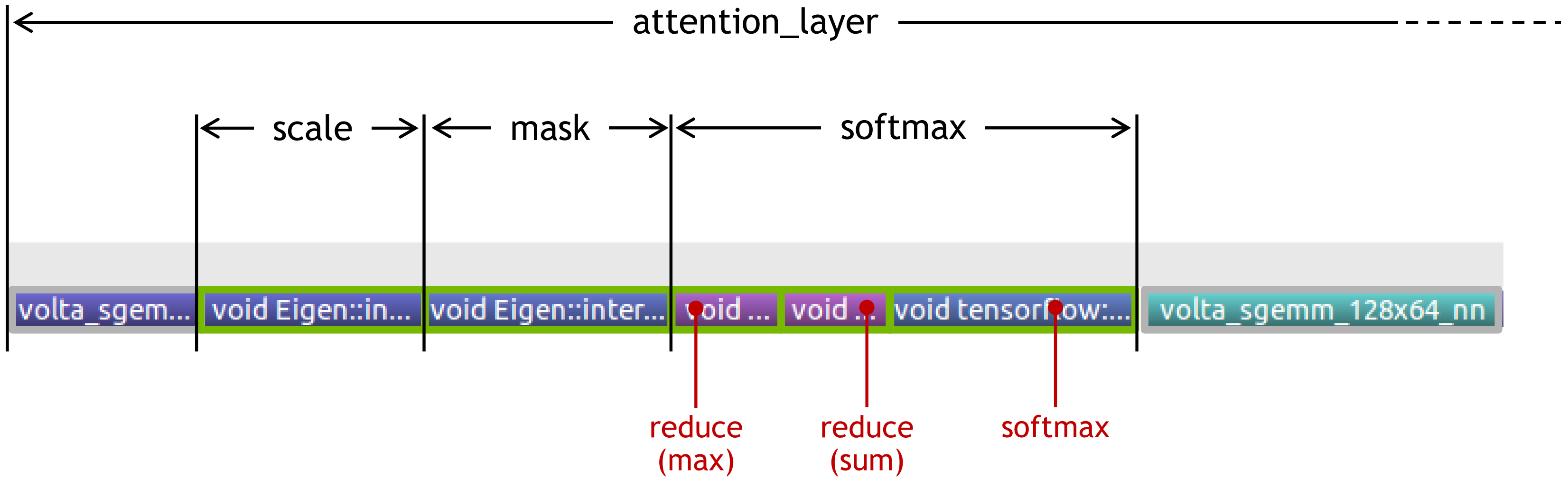
Legends:

- B: batch size (number of sequences)
- N: number of attention heads
- S: sequence length
- H: size of each attention head

How many CUDA kernels it leads to?

PROFILING MULTI-HEAD ATTENTION

Result based on NVIDIA Visual Profiler (NVVP)



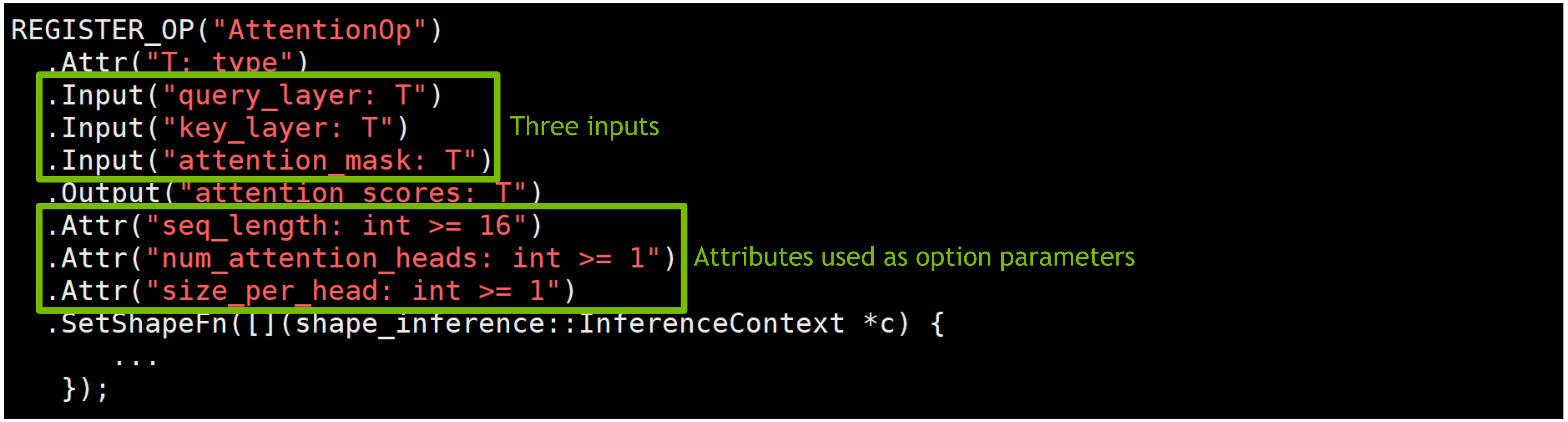
scale, mask and soft max results in **4 CUDA kernels** in C++

- Their aggregated runtime is even longer than the two gemm kernels!
- Kernel fusion can help again

STEP 1. REGISTER OP’S INTERFACE IN C++

Specify Name, Inputs, Outputs, Attributes

attention_op.cc



```
REGISTER_OP("AttentionOp")
  .Attr("T: type")
  .Input("query_layer: T")
  .Input("key_layer: T")
  .Input("attention_mask: T")
  .Output("attention_scores: T")
  .Attr("seq_length: int >= 16")
  .Attr("num_attention_heads: int >= 1")
  .Attr("size_per_head: int >= 1")
  .SetShapeFn([](shape_inference::InferenceContext *c) {
       ...
   });
```

Three inputs

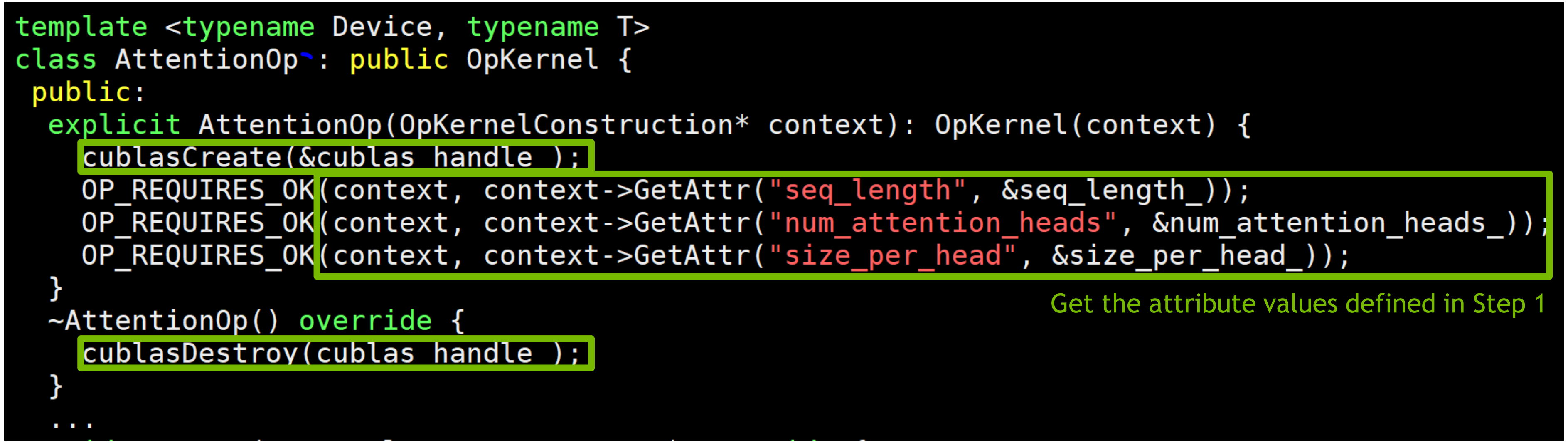
Attributes used as option parameters

- How to implement its shape function is in Appendix

STEP 2.1. INHERIT OPKERNEL CLASS

Initialization and Finalization

attention_op.cc



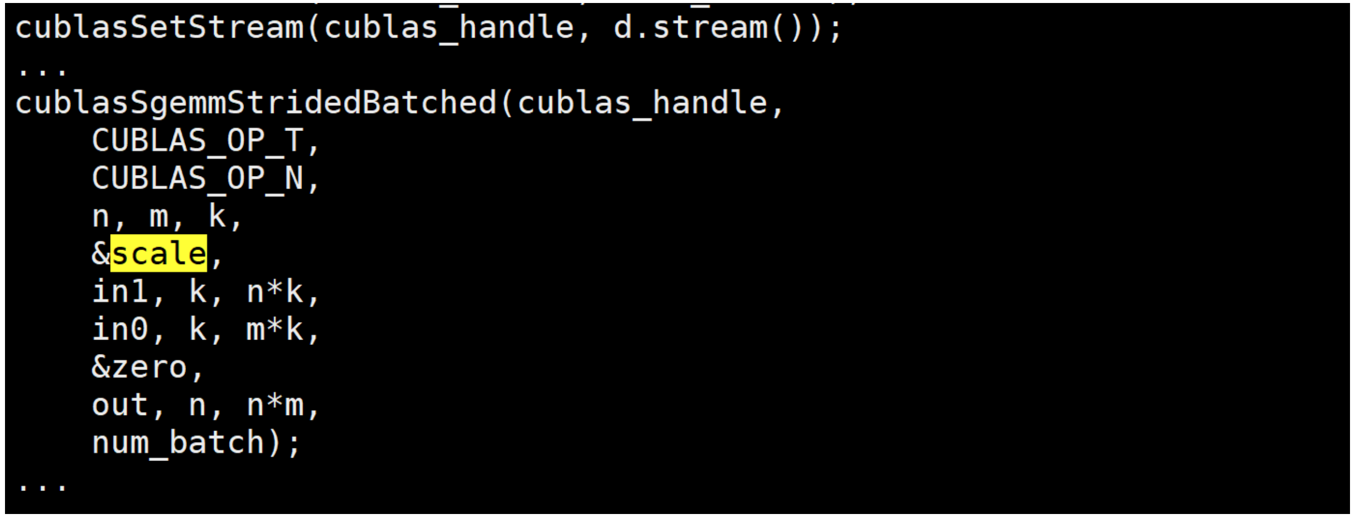
```
template <typename Device, typename T>
class AttentionOp : public OpKernel {
 public:
  explicit AttentionOp(OpKernelConstruction* context): OpKernel(context) {
    cublasCreate(&cublas_handle_);
    OP_REQUIRES_OK(context, context->GetAttr("seq_length", &seq_length_));
    OP_REQUIRES_OK(context, context->GetAttr("num_attention_heads", &num_attention_heads_));
    OP_REQUIRES_OK(context, context->GetAttr("size_per_head", &size_per_head_));
  }
  ~AttentionOp() override {
    cublasDestroy(cublas_handle_);
  }
  ...
```

Get the attribute values defined in Step 1

STEP 2.2. FUSE SCALE WITH GEMM

How to Use CUBLAS API

attention_op.cu.cc



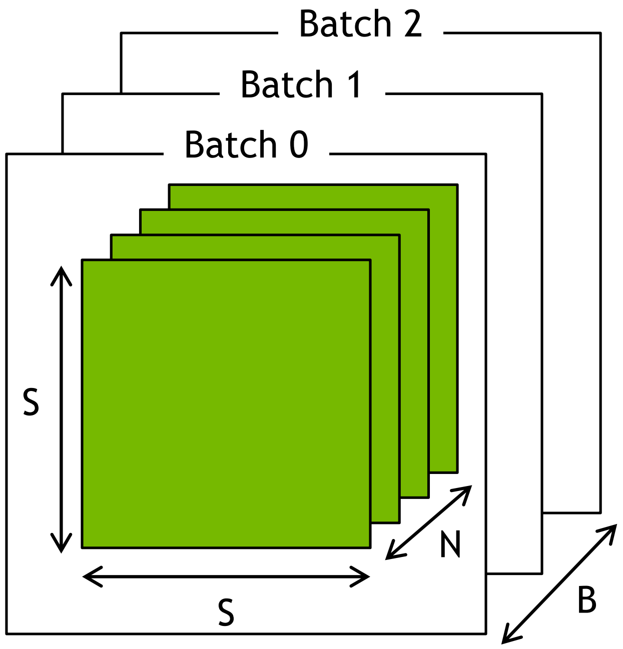
```
cublasSetStream(cublas_handle, d.stream());
...
cublasSgemmStridedBatched(cublas_handle,
    CUBLAS_OP_T,
    CUBLAS_OP_N,
    n, m, k,
    &scale,
    in1, k, n*k,
    in0, k, m*k,
    &zero,
    out, n, n*m,
    num_batch);
...
```

cublas General Matrix Multiplication (GEMM) APIs support **in-register** scaling

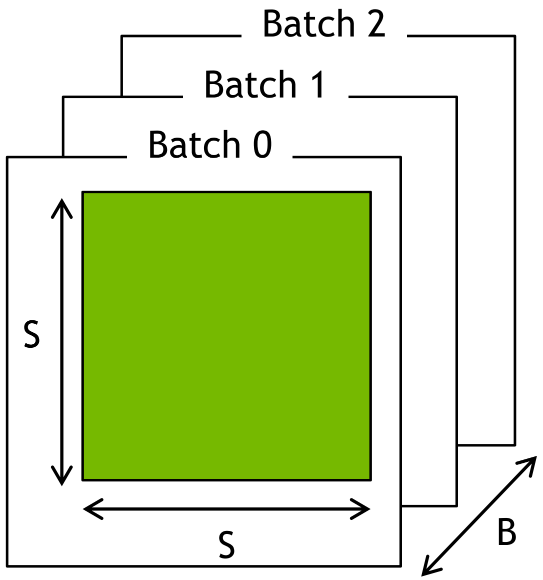
- $C = s \cdot (A \times B)$
- C is accessed **only once** for the final write

STEP 2.3. FUSE SOFTMAX WITH MASK

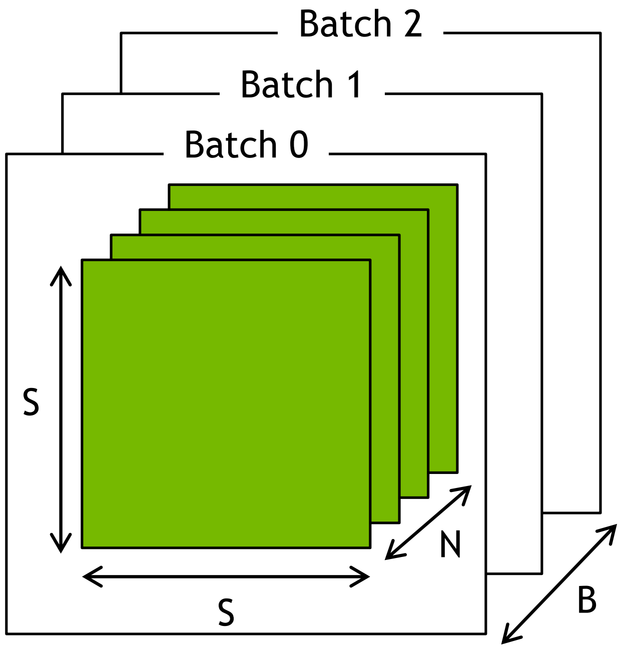
Data Layout After Step 2.2



in[B, N, S, S]



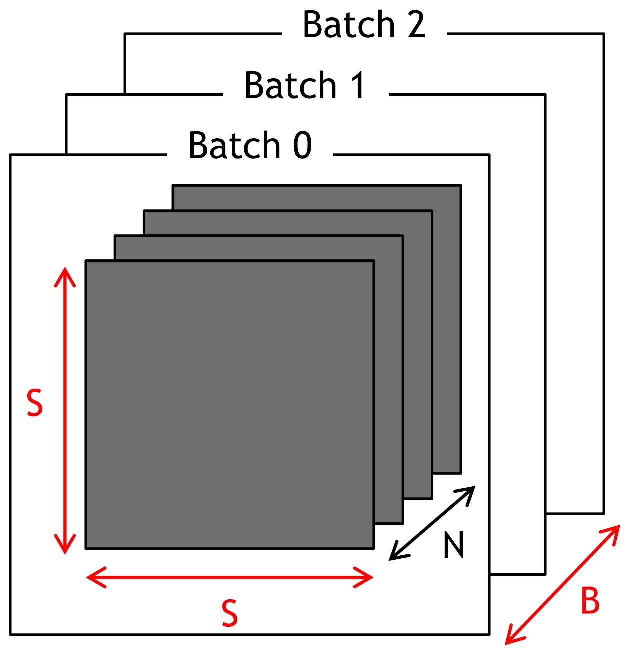
mask[B, 1, S, S]



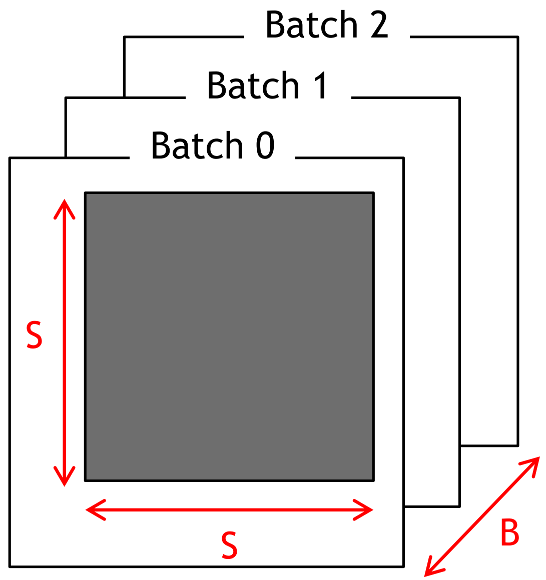
out[B, N, S, S]

STEP 2.3. FUSE SOFTMAX WITH MASK

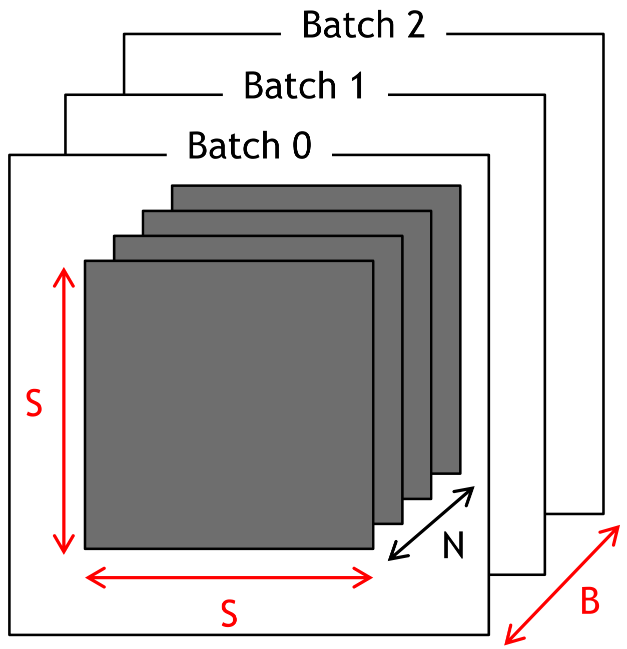
A Possible Parallelization Approach (suppose B=8, S=384)



in[B, N, S, S]



mask[B, 1, S, S]

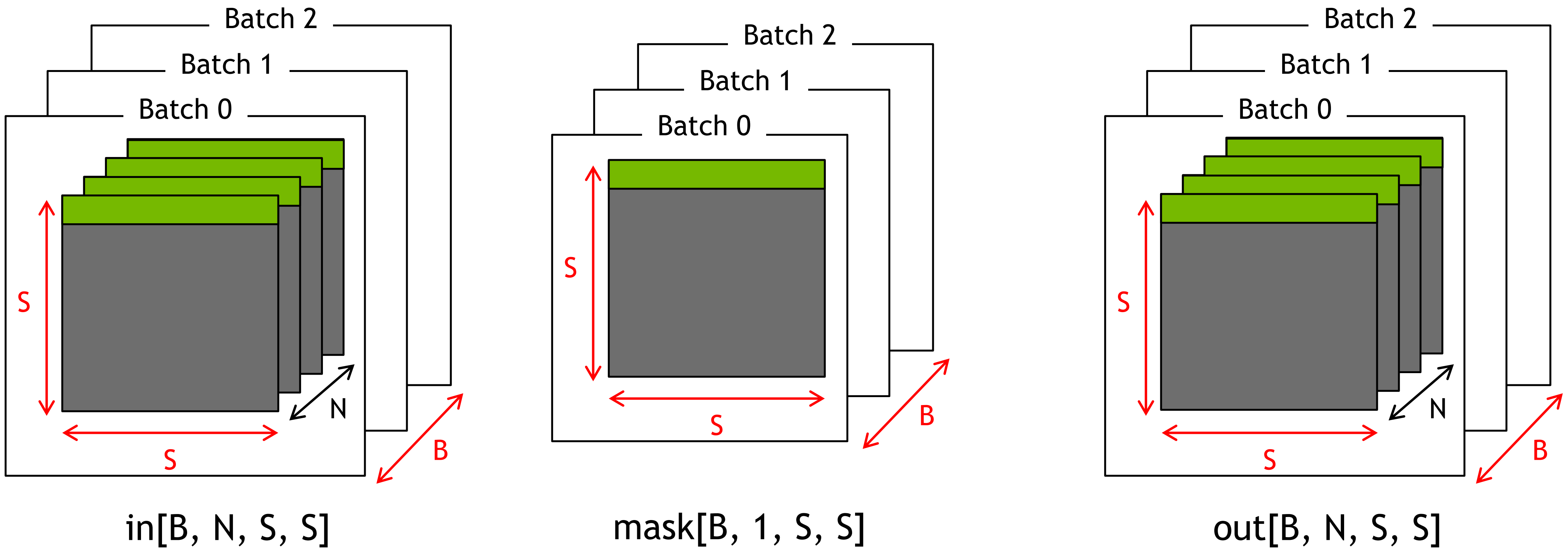


out[B, N, S, S]

- Parallelize for sequence length (S) and batch size (B)

STEP 2.3. FUSE SOFTMAX WITH MASK

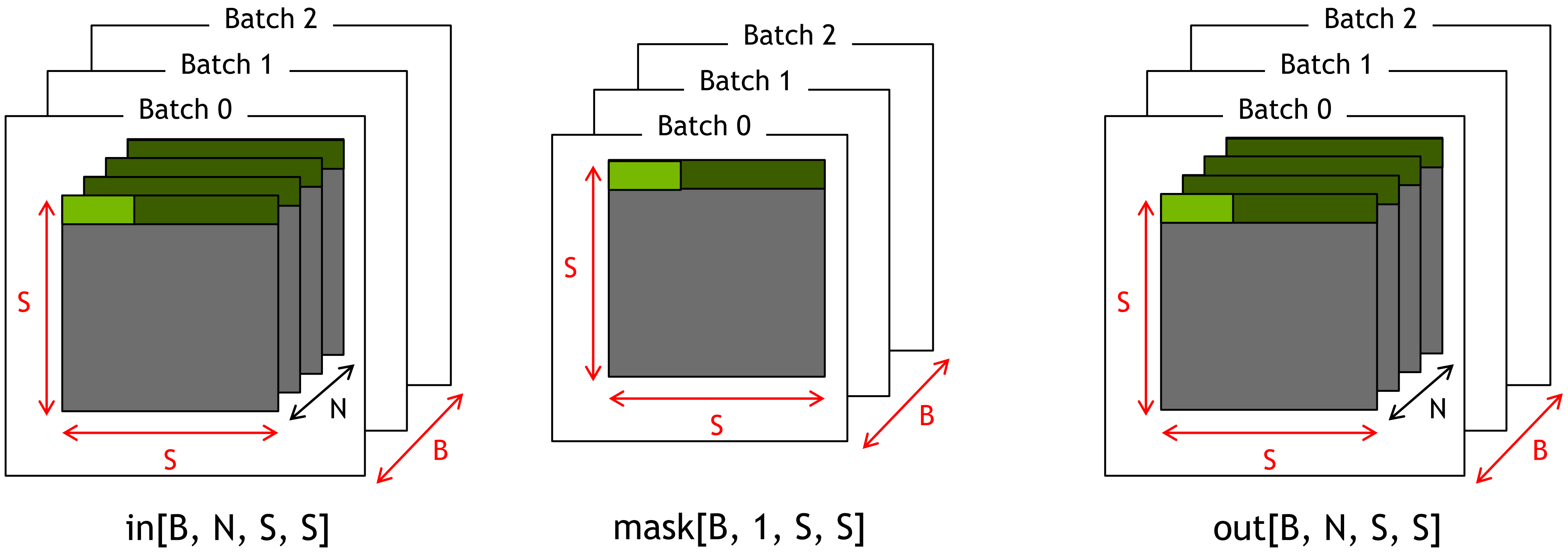
Data accessed by A Thread Block (suppose B=8, S=384)



- Total (B x S) thread blocks = 3072 thread blocks

STEP 2.3. FUSE SOFTMAX WITH MASK

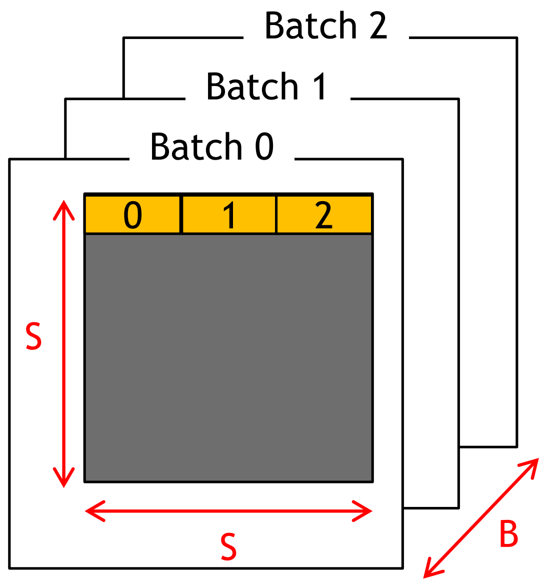
Set Block Size to 128



- If S=384, each block has **3 iterations** within the same row

STEP 2.3. FUSE SOFTMAX WITH MASK

Reuse Mask across N Attention Heads in Batch



attention_kernel.cu

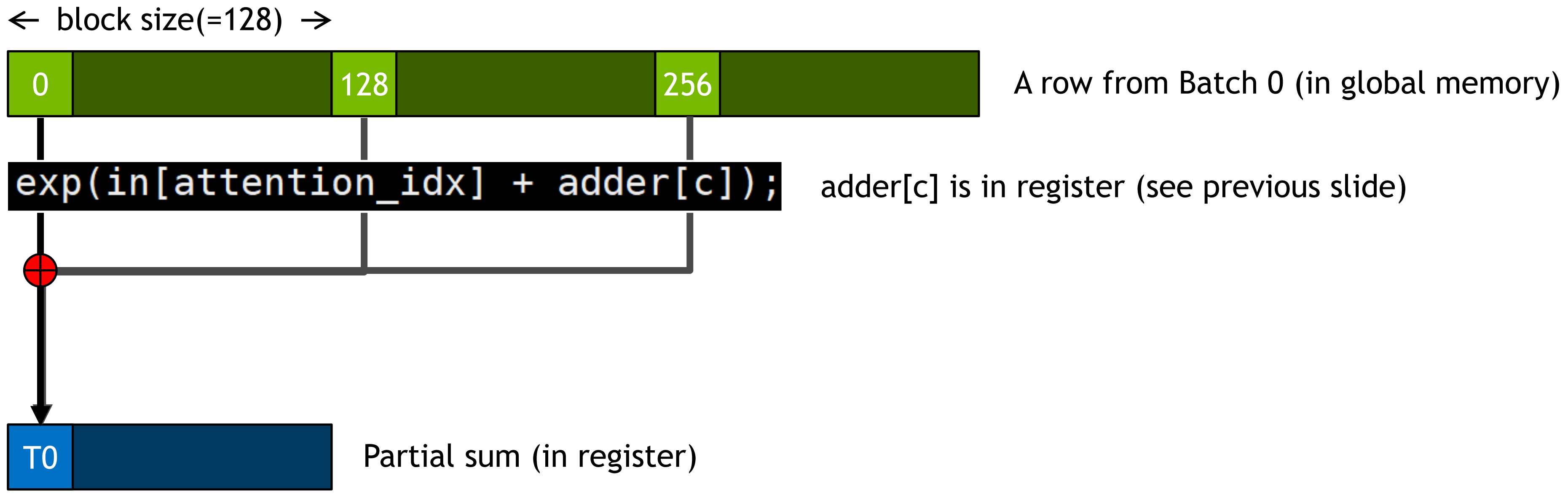
```
T adder[n_col_chunk_per_block];
for(int c=0; c<n_col_chunk_per_block; c++) {
  adder[c] = (T(1.0) - attention_mask[attention_mask_idx]) * T(-10000.0);
  attention_mask_idx += chunk_length;
}
```

- Let each thread load a mask into **a local variable** (stored in register)

  That's why we didn't parallelize across N attention heads (will be revisited in later slide)

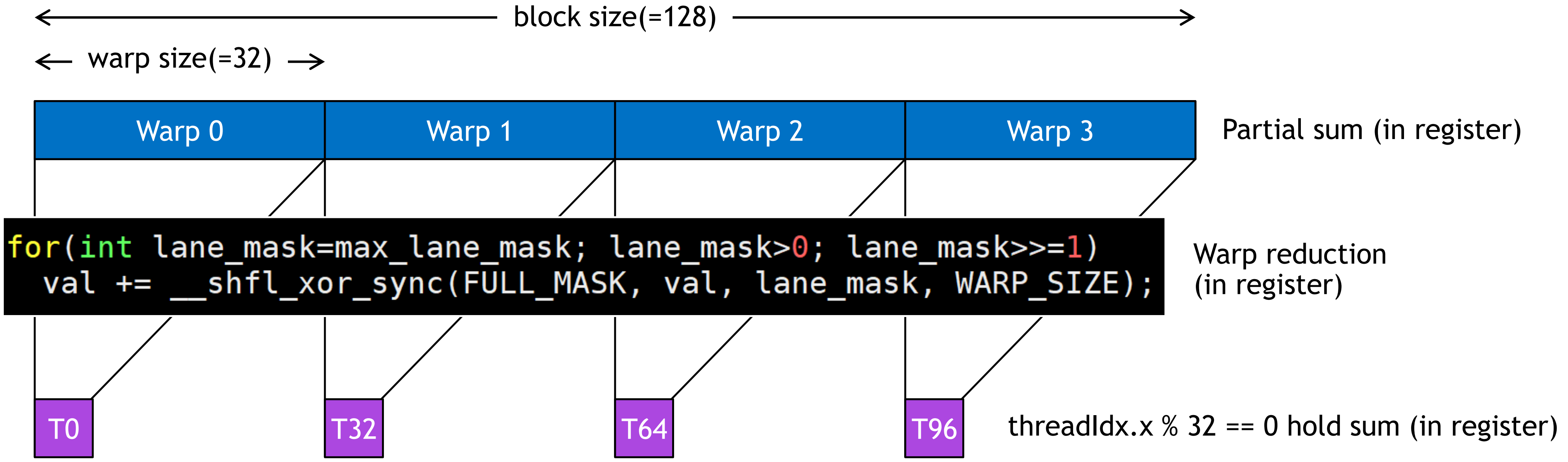
STEP 2.3. FUSE SOFTMAX WITH MASK

Parallel Row Reduction to Get Sum (1/3)


← block size(=128) →
0
128
256
A row from Batch 0 (in global memory)
exp(in[attention_idx] + adder[c]);
adder[c] is in register (see previous slide)
T0
Partial sum (in register)

STEP 2.3. FUSE SOFTMAX WITH MASK

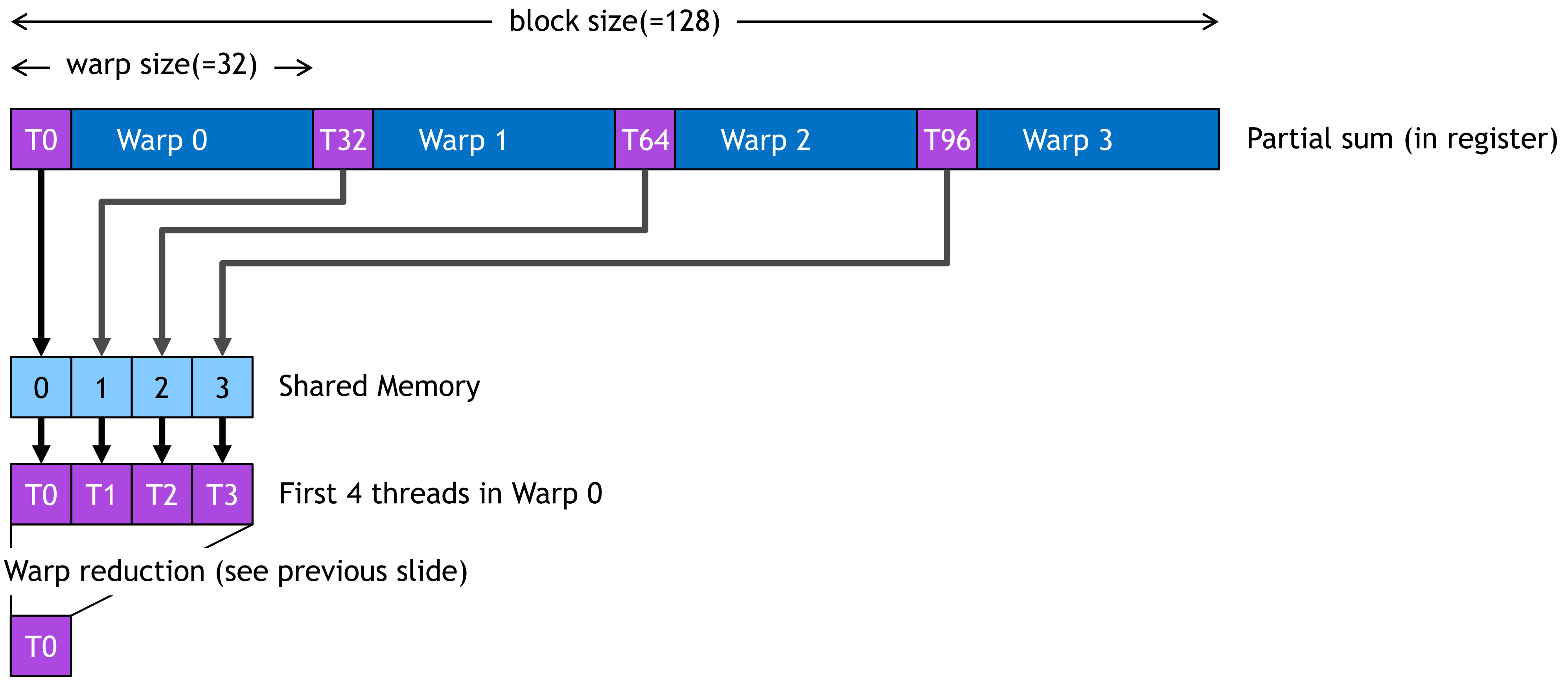
Parallel Row Reduction to Get Sum (2/3)


block size(=128)
warp size(=32)
Warp 0
Warp 1
Warp 2
Warp 3
Partial sum (in register)
Warp reduction (in register)
T0
T32
T64
T96
threadIdx.x % 32 == 0 hold sum (in register)

```
for(int lane_mask=max_lane_mask; lane_mask>0; lane_mask>>=1)
  val += __shfl_xor_sync(FULL_MASK, val, lane_mask, WARP_SIZE);
```

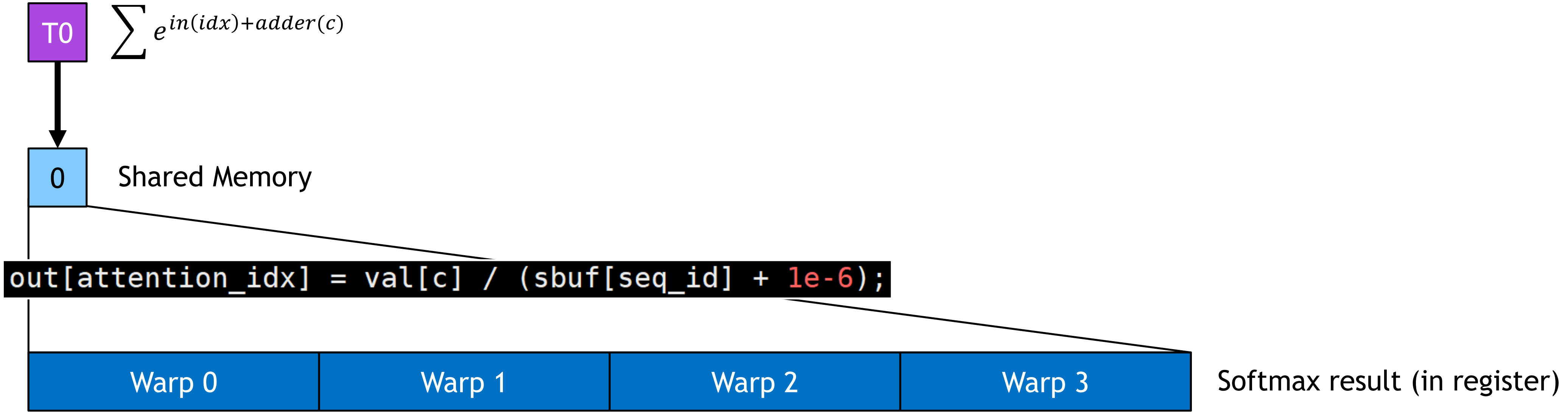
STEP 2.3. FUSE SOFTMAX WITH MASK

Parallel Row Reduction to Get Sum (3/3)



STEP 2.3. FUSE SOFTMAX WITH MASK

Broadcast Sum and Do Softmax



```
out[attention_idx] = val[c] / (sbuf[seq_id] + 1e-6);
```

- Everything was done in register or shared memory

  Read “in” and “mask” once and write “out” once to global memory

STEP 2.3. FUSE SOFTMAX WITH MASK

Repeat for N Attention Heads

attention_kernel.cu

```
...
T adder[n_col_chunk_per_block];
for(int c=0; c<n_col_chunk_per_block; c++) {
  adder[c] = (T(1.0) - attention_mask[attention_mask_idx]) * T(-10000.0);
  attention_mask_idx += chunk_length;
}
...
for(int h=0; h<num_attention_heads; h++) {
  ...
}
```

Row reduction & Softmax adder is reused N times

STEP 3. USE CUSTOM OP IN PYTHON

How to Substitute Original Implementation

modeling.py

```
ref_tensor = tf.matmul(query_layer, key_layer, transpose_b=True)
ref_tensor = tf.multiply(ref_tensor, 1.0 / math.sqrt(float(size_per_head)))
adder = (1.0 - tf.cast(attention_mask, ref_tensor.dtype)) * -10000.0
ref_tensor += adder;
ref_tensor = tf.nn.softmax(ref_tensor)
```

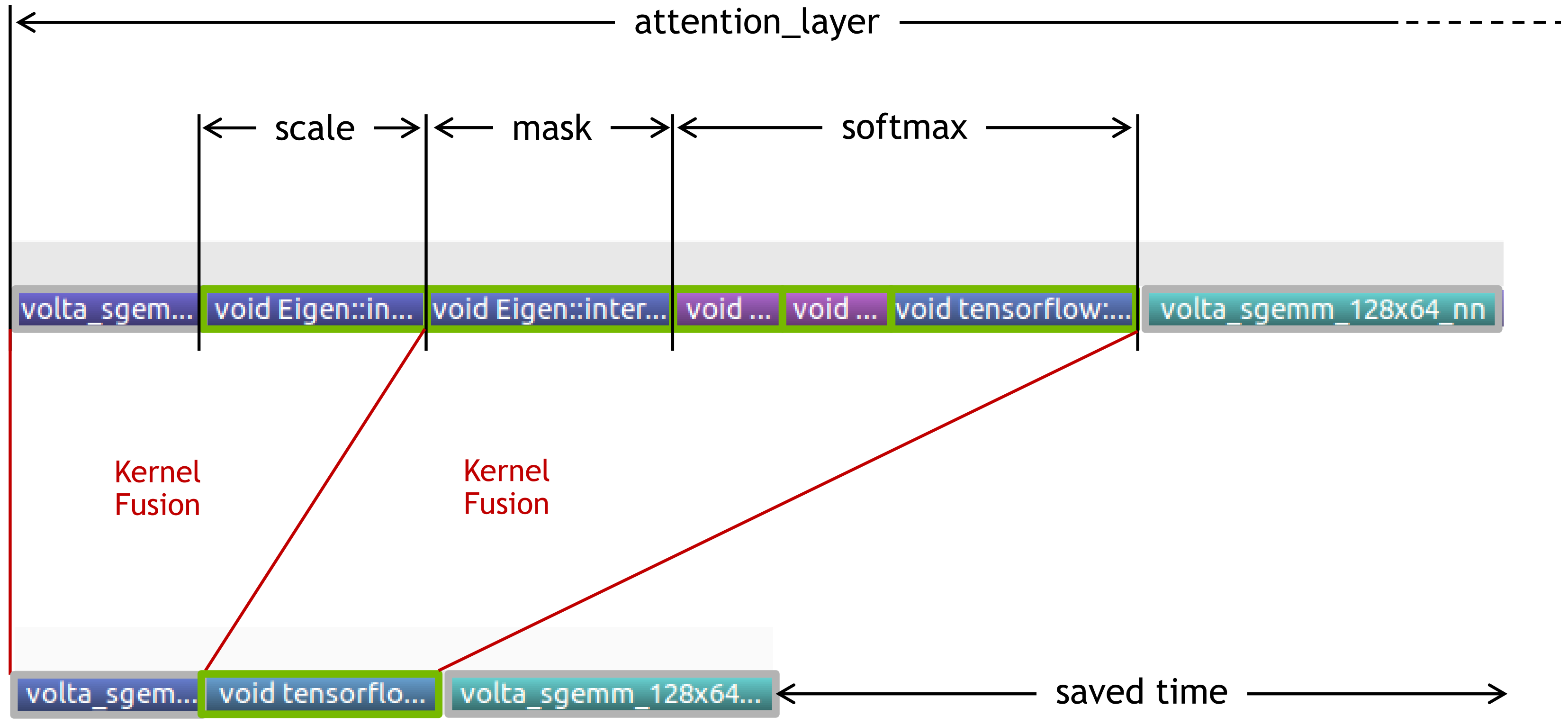
Original Implementation

```
out_tensor = cuda_custom_ops_module.attention_op(query_layer, key_layer, attention_mask,
    seq_length=seq_length,
    num_attention_heads=num_attention_heads,
    size_per_head=size_per_head)
```

Custom Implementation

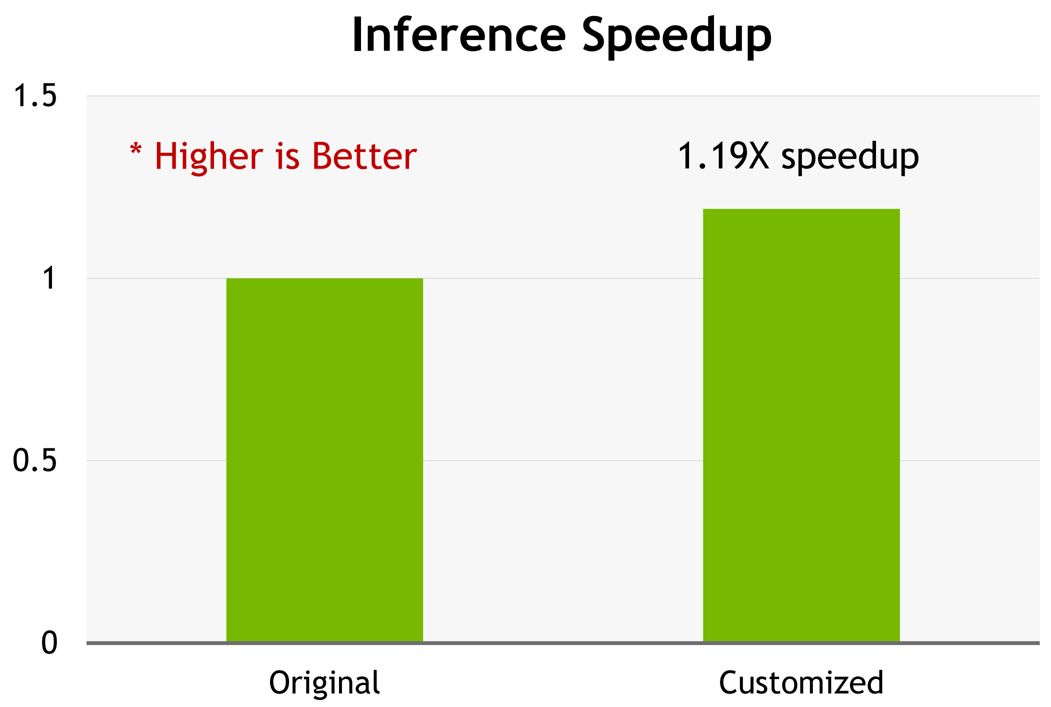
PROFILING FUSED MULTI-HEAD ATTENTION

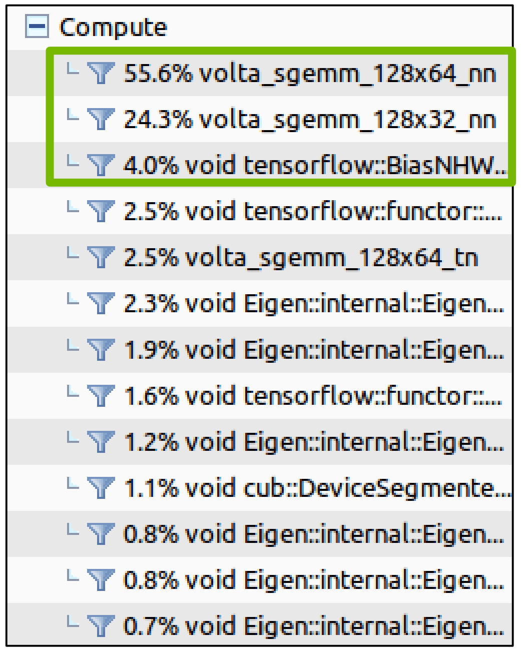
Result based on NVIDIA Visual Profiler (NVVP)



END-TO-END PERFORMANCE

SQuAD Inference on a single V100 16G





Compute
55.6% volta_sgemm_128x64_nn
24.3% volta_sgemm_128x32_nn
4.0% void tensorflow::BiasNHW...
2.5% void tensorflow::functor::...
2.5% volta_sgemm_128x64_tn
2.3% void Eigen::internal::Eigen...
1.9% void Eigen::internal::Eigen...
1.6% void tensorflow::functor::...
1.2% void Eigen::internal::Eigen...
1.1% void cub::DeviceSegmente...
0.8% void Eigen::internal::Eigen...
0.8% void Eigen::internal::Eigen...
0.7% void Eigen::internal::Eigen...

GEMM and add bias represent 83.9% now

- To get more speedup, GEMM-centric optimization is required

WHAT WE CAN DO MORE

Further Optimization Options

More Fusion and/or custom kernels in “Feed Forward” and “Multi-Head Attention”

- GEMM + add bias + layer norm + activation
- GEMM + scale + mask + softmax + GEMM

Apply **Quantization** or Use lower precision, e.g., FP16, INT8

FasterTransformer will be released soon

- Highly optimized BERT Transformer for Inference based on custom CUDA kernels and CUBLAS
- Will support various sequence length and multi-precisions

MORE ABOUT TENSORFLOW CUSTOM OP

Pitfalls and Tips

Use allocate_output or allocate_temp instead of explicit cudaMalloc()

- allocate_output: used to allocate output tensor
- allocate_temp: used to allocate temporary memory, not exposed as a tensor
- cudaFree() is not necessary because TF has its own memory management

If you encounter the undefined symbol error in building, check the list below

- -D_GLIBCXX_USE_CXX11_ABI=0 (if GCC >= 5.0)
- Library dependency, template specialization

RELATED SESSIONS IN AI CONFERENCE

Learn More About Inference and Profiling

Deep Learning Inference 가속화를 위한 NVIDIA의 기술 소개 - 이종환 (NVIDIA)

- 13:50 – 14:30, Track 2

TensorRT를 이용한 OCR Model Inference 성능 최적화 - 이현수 (카카오)

- 14:40 – 15:20, Track 2

GPU Profiling 기법을 통한 Deep Learning 성능 최적화 기법 소개 - 홍광수 (NVIDIA)

- 16:30 – 17:10 Track 3

WE ARE HIRING!

AI Developer Technology Engineer

Study and Develop cutting-edge techniques in DL/ML, and **perform in-depth analysis and optimization** to achieve the best possible performance on GPU architectures

Work directly with key developers to understand and solve the practical problems using GPUs

Collaborate closely with the architecture, libraries, tools and research teams at NVIDIA to influence the design of next-generation architectures, software, and programming models

In short, we do what was discussed in this session ☺

APPENDIX

RULES IN STEP 1

Things to Remember

- Multiple inputs and outputs are allowed

- .Attr(“AttrName”: “AttrType”) is used to configure the op

  int32, float, double, bool, **type** and etc are allowed as “AttrType”

- .SetShape() defines the output shape

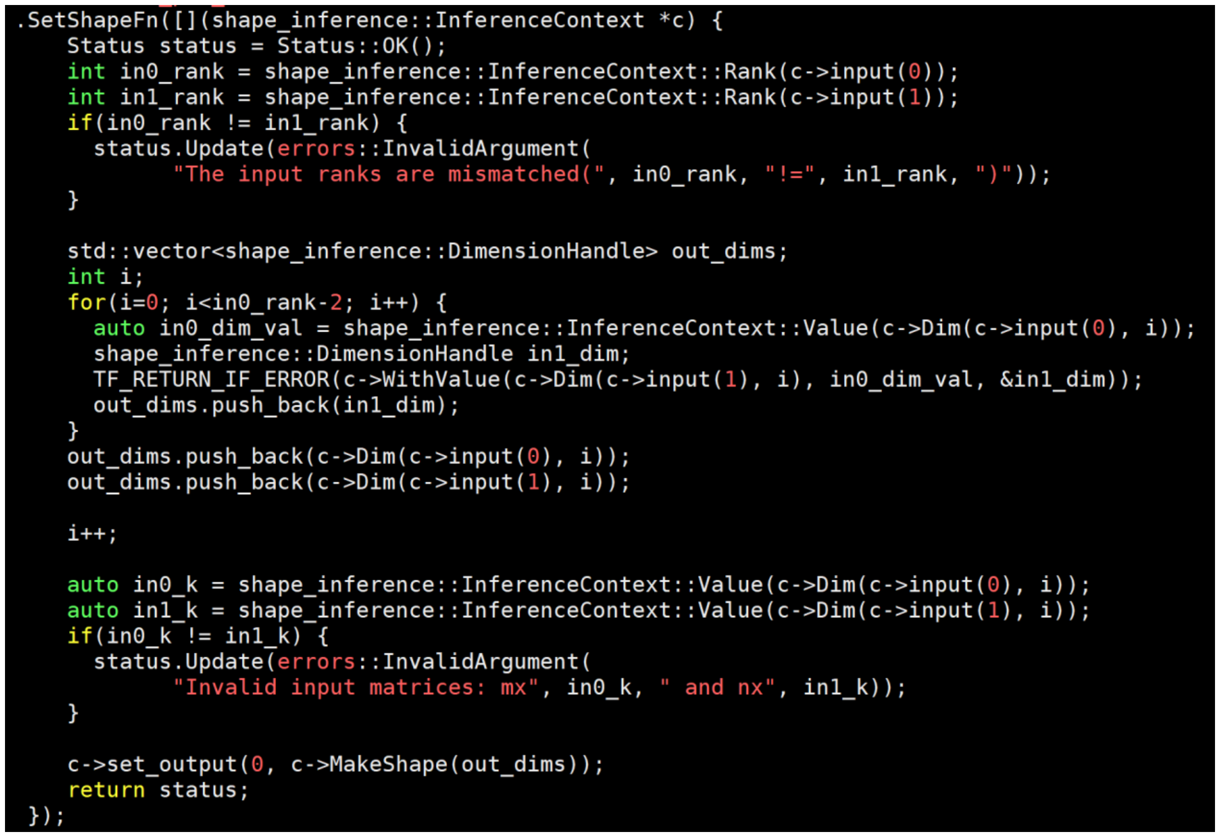
  c->set_output(i, shape): set ith output’s shape

  c->input(i): get ith input’s shape

STEP 1. REGISTER OP’S INTERFACE IN C++

Calculate Output Shape and check errors

attention_op.cc



```
.SetShapeFn([](shape_inference::InferenceContext *c) {
    Status status = Status::OK();
    int in0_rank = shape_inference::InferenceContext::Rank(c->input(0));
    int in1_rank = shape_inference::InferenceContext::Rank(c->input(1));
    if(in0_rank != in1_rank) {
      status.Update(errors::InvalidArgument(
            "The input ranks are mismatched(", in0_rank, "!=", in1_rank, ")"));
    }

    std::vector<shape_inference::DimensionHandle> out_dims;
    int i;
    for(i=0; i<in0_rank-2; i++) {
      auto in0_dim_val = shape_inference::InferenceContext::Value(c->Dim(c->input(0), i));
      shape_inference::DimensionHandle in1_dim;
      TF_RETURN_IF_ERROR(c->WithValue(c->Dim(c->input(1), i), in0_dim_val, &in1_dim));
      out_dims.push_back(in1_dim);
    }
    out_dims.push_back(c->Dim(c->input(0), i));
    out_dims.push_back(c->Dim(c->input(1), i));

    i++;

    auto in0_k = shape_inference::InferenceContext::Value(c->Dim(c->input(0), i));
    auto in1_k = shape_inference::InferenceContext::Value(c->Dim(c->input(1), i));
    if(in0_k != in1_k) {
      status.Update(errors::InvalidArgument(
            "Invalid input matrices: mx", in0_k, " and nx", in1_k));
    }

    c->set_output(0, c->MakeShape(out_dims));
    return status;
 });
```

Top 4 things to notice

- How to get input dim and rank
- How to make shape object
- How to handle invalid arguments
- How to set output shape