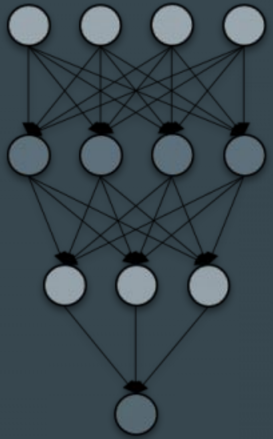
Improving Data Locality by Kernel Fusion in DNNs



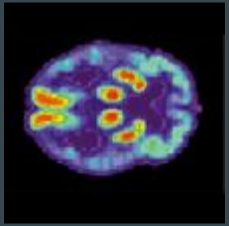
May 17, 2019

Snehil Verma, Bagus Hanindhito, Joseph Dean

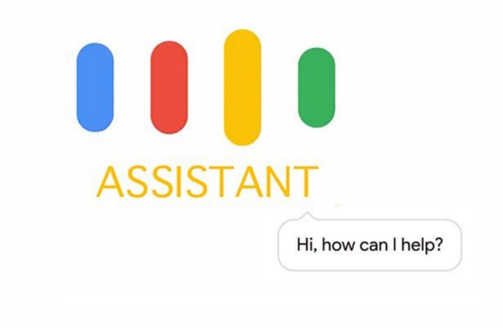
Overview

1. Applications of DNNs - Translation.
2. Fairseq's implementation of Translation - from RNNs to CNNs.
3. Working of Encoder/Decoder.
4. Motivation for kernel fusion.
5. Hardware setup and Methodology
6. Different approaches (bare-metal CUDA, CUTLASS, CuDNN).
7. Integrating with PyTorch.
8. Results and insights
9. Failed attempts
10. What did we learn?
11. Future goals
12. Acknowledgements

DNNs in today's world



- Medical Diagnosis - Image classification
- Self driving cars – Object detection
- **Language translation**
- Personal Assistant - Speech recognition, Recommendation
- AlphaGo - Reinforcement learning



Translation - Sequence to Sequence Learning via Fairseq

https://github.com/pytorch/fairseq/blob/master/fairseq.gif

Fairseq model implementations

LSTM

Luong et al. (2015)

Wiseman and Rush (2016)

CNN

Dauphin et al. (2017)

Gehring et al. (2017): Convolutional Sequence to Sequence Learning

Edunov et al. (2018)

Fan et al. (2018)

Transformer

Vaswani et al. (2017)

Ott et al. (2018)

Edunov et al. (2018)

Baevski and Auli (2018)

Shen et al. (2019)

LightConv and DynamicConv

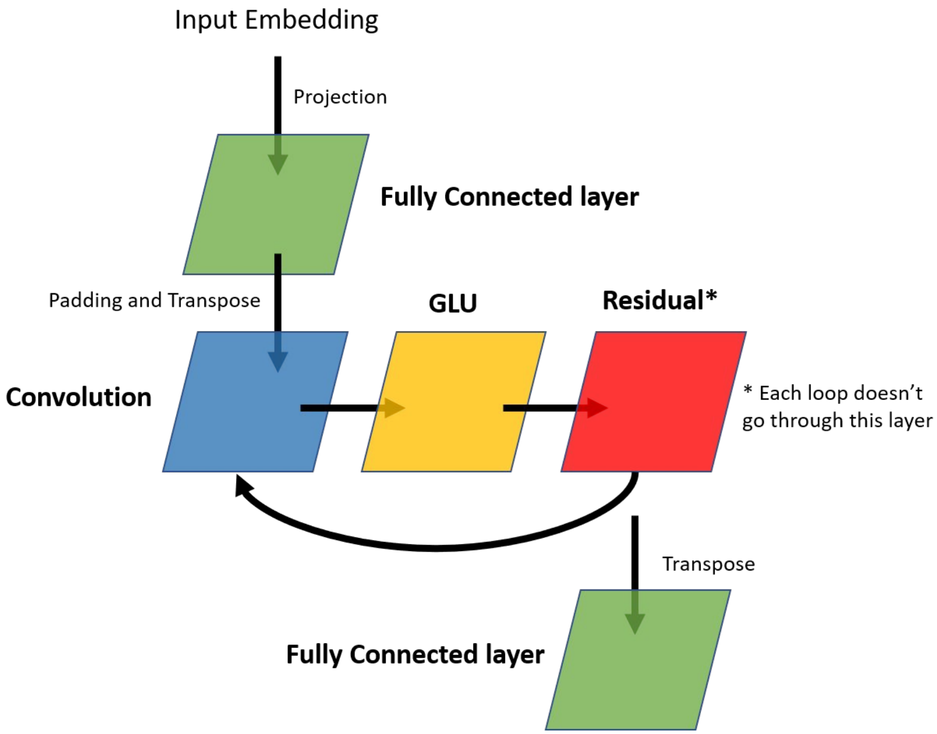
Wu et al. (2019)

Software Setup

Model	● Gehring et al. (2017): Convolutional Sequence to Sequence Learning
Framework	● PyTorch
Language and Tools	● CUDA, C++ ● cuBLAS, CUTLASS, cuDNN
Dataset	● WMT14 English-French

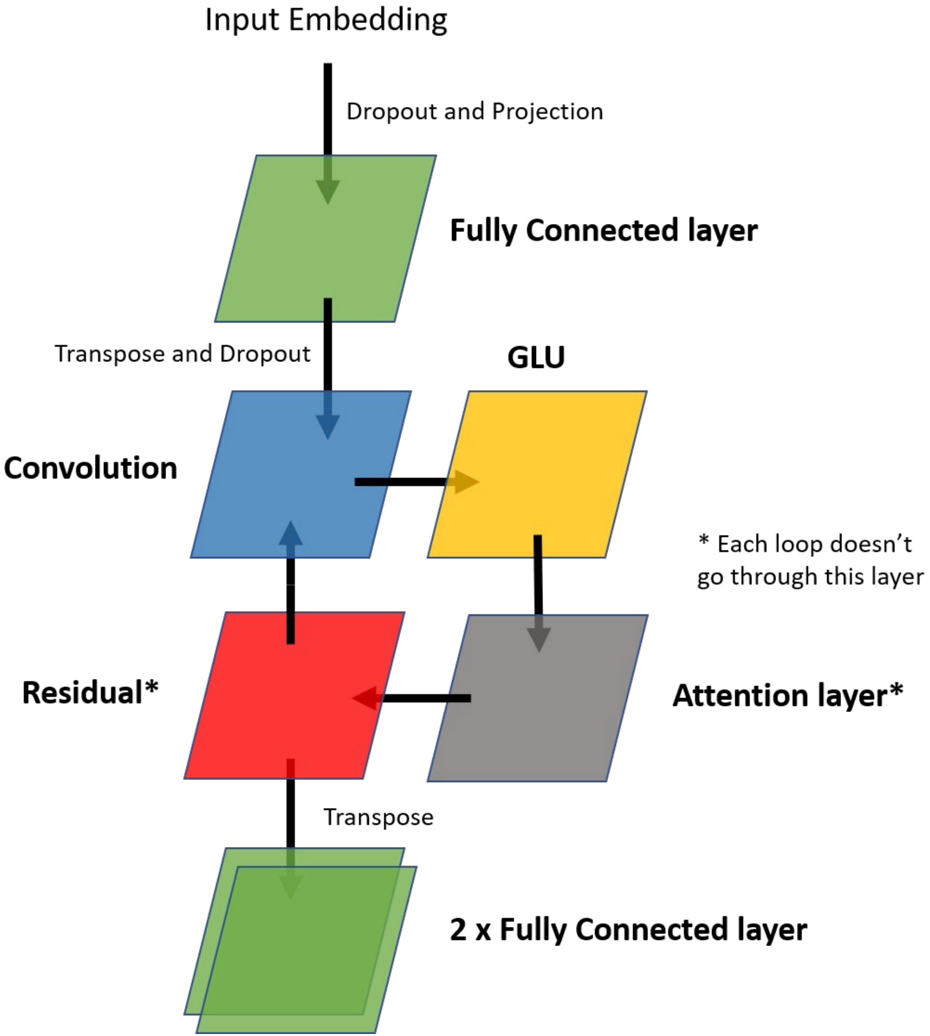
Encoder

Computational Graph



Decoder

Computational Graph



Input Embedding

Dropout and Projection

Fully Connected layer

Transpose and Dropout

GLU

Convolution

* Each loop doesn't go through this layer

Residual*

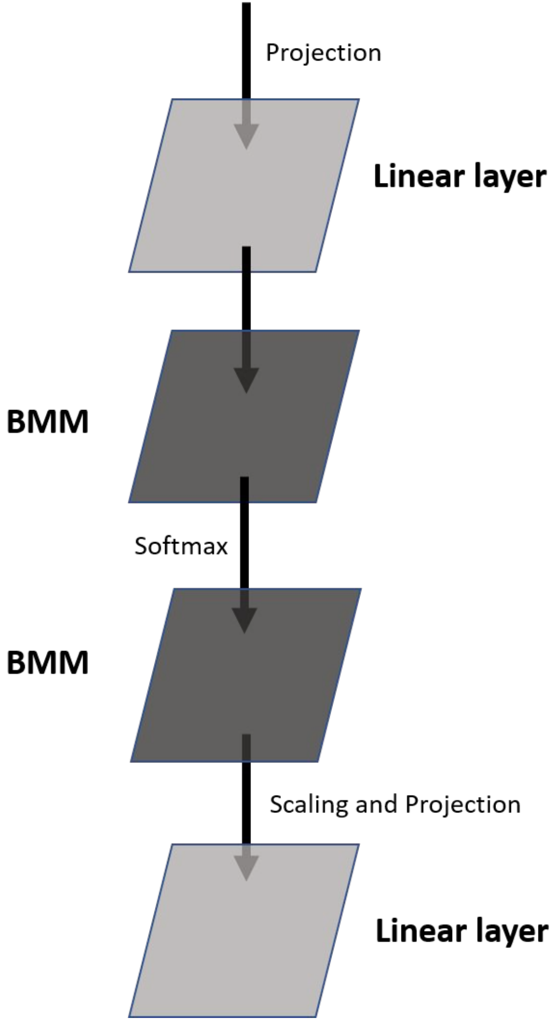
Attention layer*

Transpose

2 x Fully Connected layer

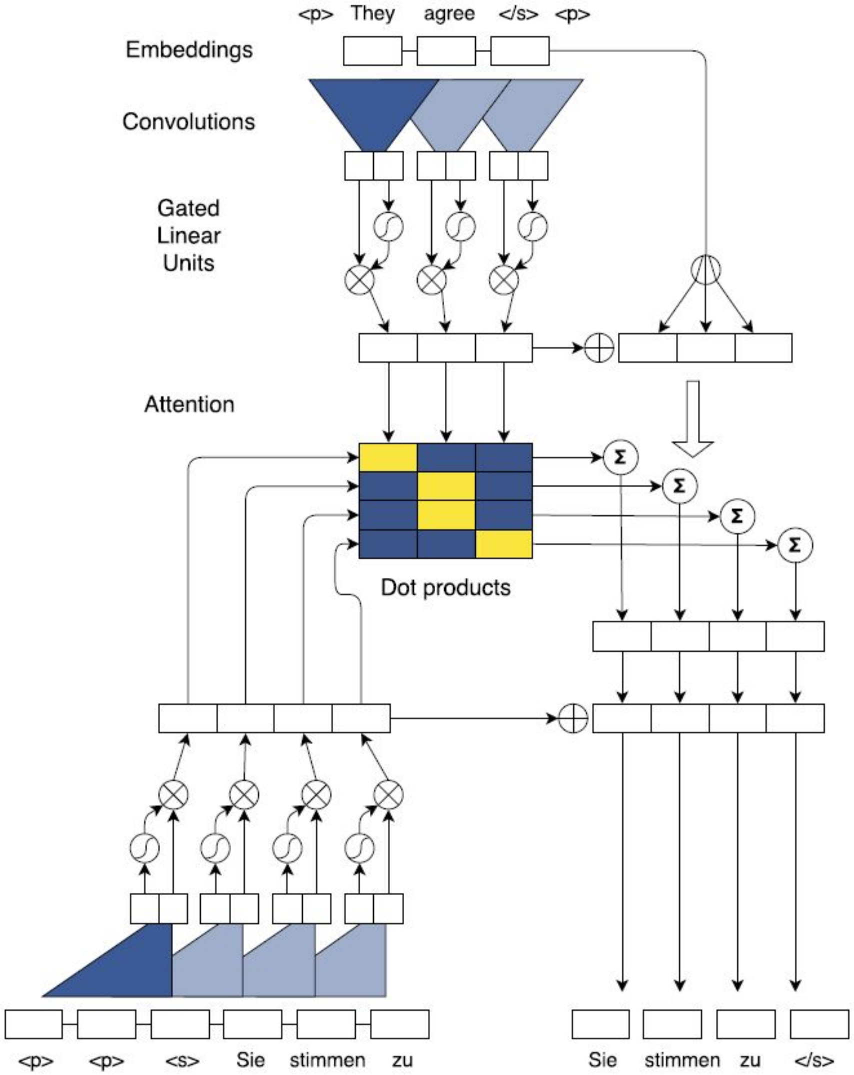
Diving deeper into

Attention Layer



Overall Architecture

Gehring et al. (2017)

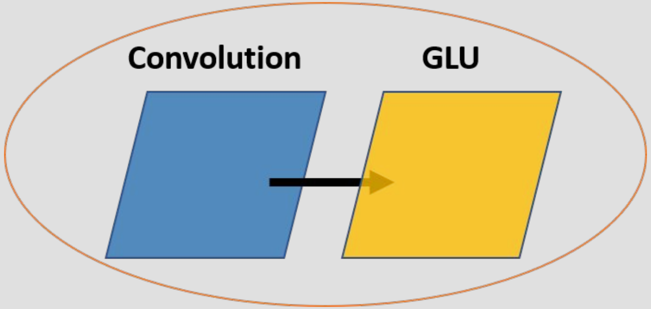


PyTorch (python) interface lacks support of managing memory explicitly.

Scope of improving **data locality**
by fusing kernels!

Project objective: Improving data locality by kernel fusion.

- Fuse Convolution layer with GLU layer.



PyTorch Autograd analysis

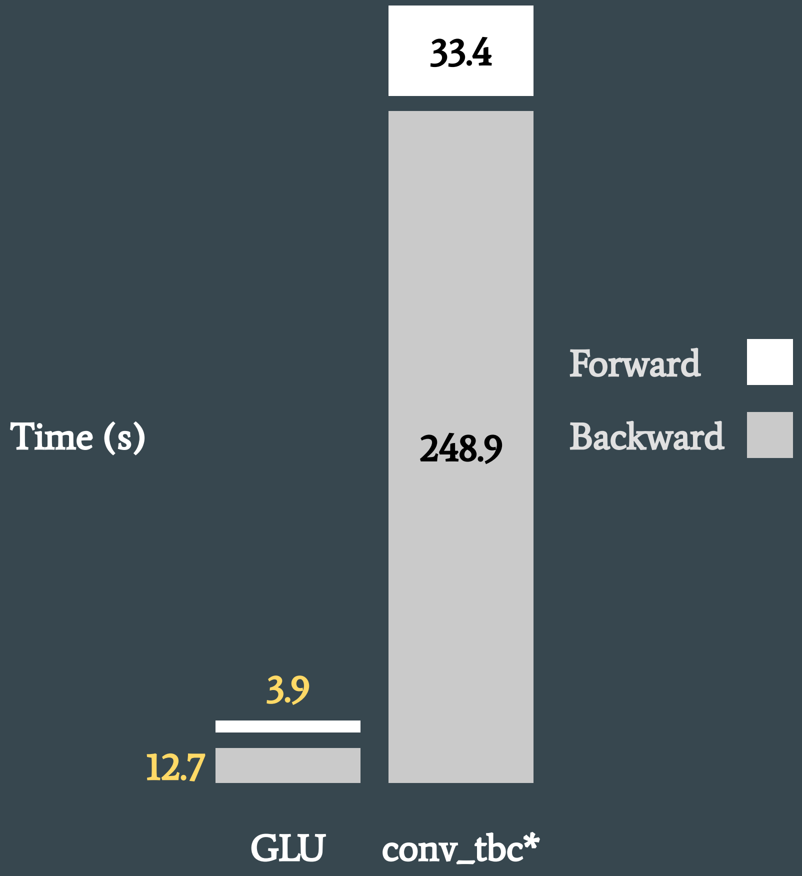
Profiled 3000 updates of the second epoch.

Findings

GLU operation takes around **6%** time of the convolution operation (including both forward and backward path).

Implications:

- Major performance improvement cannot be attained in fusing the two layers.
- However, fusing these layers is the first step towards improving data locality.

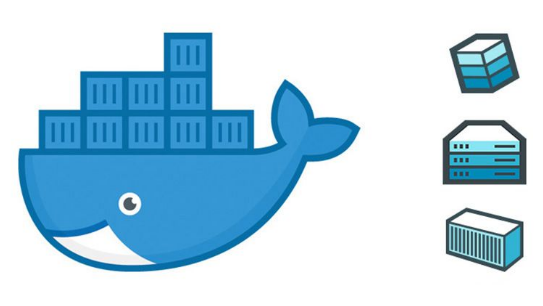


* Convolution TBC (Time, Batch, Channel)

Ethical Practice

In order to make changes at the kernel level and understand the working of PyTorch we took several steps to ensure correct and fair experimentation.

- Created `DockerFile`.
- Used `nvidia-docker` containers for experimentations.
- The `DockerFile` ensured that PyTorch is installed from source.



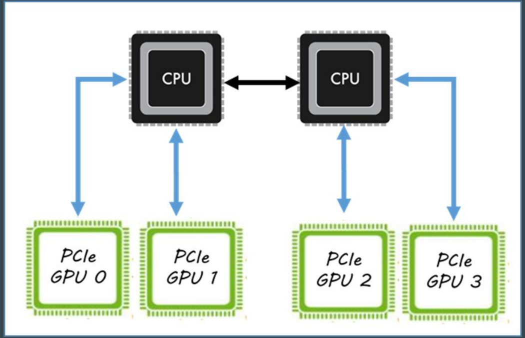
Note that, PyTorch was a new framework for everyone and hence, it took sometime for us to get familiar with it. It was important to understand how the python interface leads to C++ and CUDA library calls.

Hardware Setup

Dell PowerEdge T640

4 NVIDIA Tesla V100 GPUs

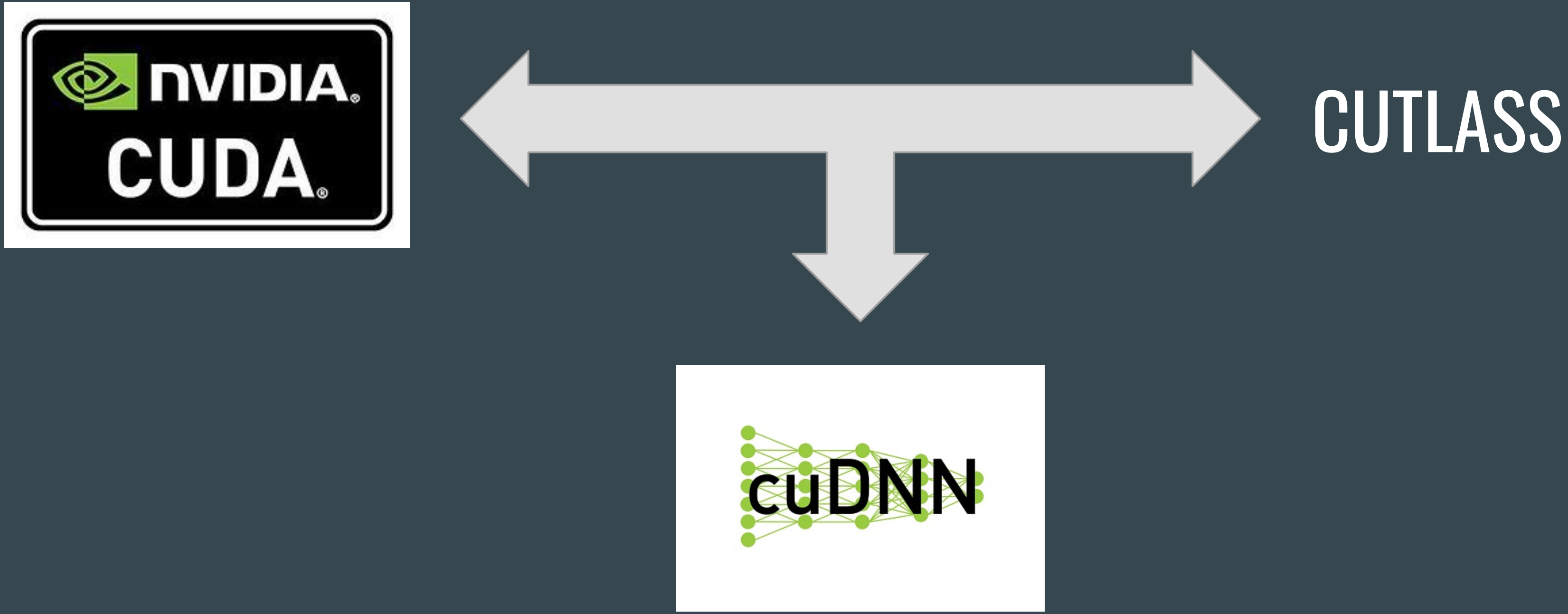
System Architecture



DELL EMC

NVIDIA

3 different approaches to improve Data Locality



CUTLASS

Methodology

1. Since the fairseq implementation on PyTorch consisted of many files, we wanted to first understand the dataflow of the implementation - link: https://github.com/UT-LCA/FusedConvGLU/blob/master/fairseq_dataflowgraph.pdf
2. We followed the python function calls to the C++ backend, where we observed the computation that was occurring.
3. The fairseq implementation heavily relied on a function called `conv_tbc` (Convolution TBC (Time, Batch, Channel)), which is similar to the 1-D convolution - however, the order of the dimensions of the tensors were different.
4. Note that, default fairseq implementation uses NVIDIA's cuBLAS library function calls which were kind-of blackbox to us.
5. Additionally, each approach was verified using a simple C++ convolution and GLU function.

CUDA implementation

Bare-metal CUDA implementation

- More control over the data.
- The code’s performance would not be comparable to the library performance.

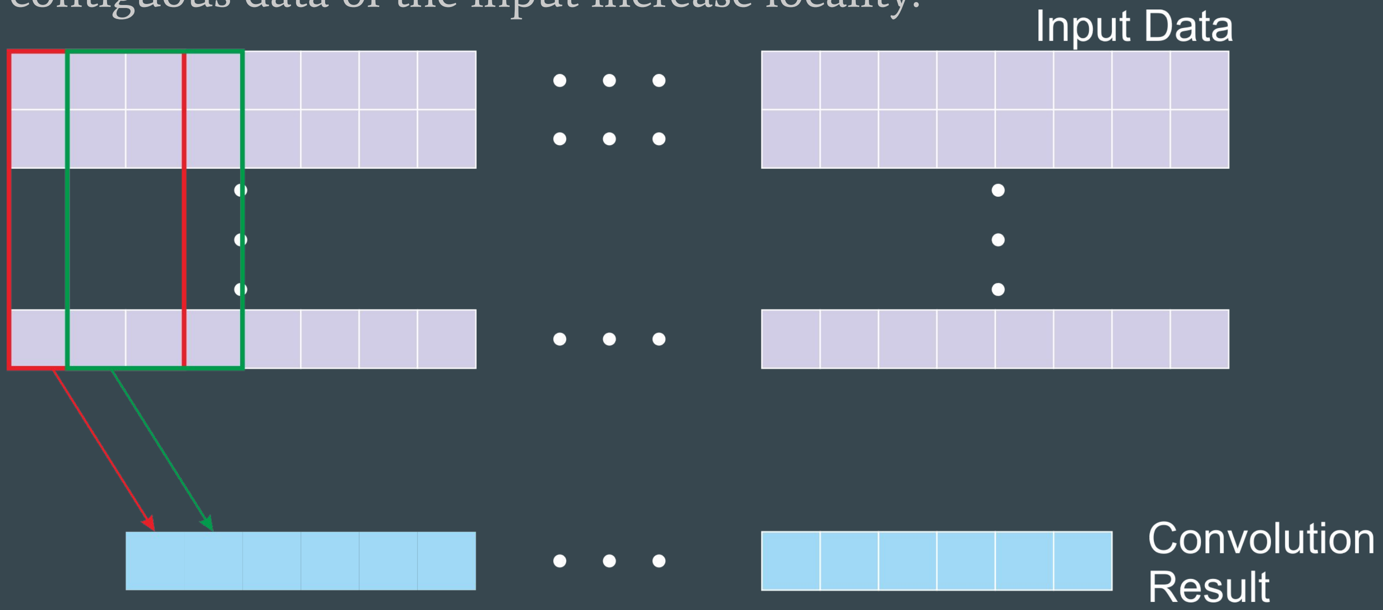
Note that, the goal of the project is to perform kernel fusion and understand it’s benefits, not to optimize the convolution function.

CUDA implementation (cont'd)

Convolution normally can be done by sliding the kernel into the input contiguously.

❑ Each computation for each kernel position is highly parallelizable.

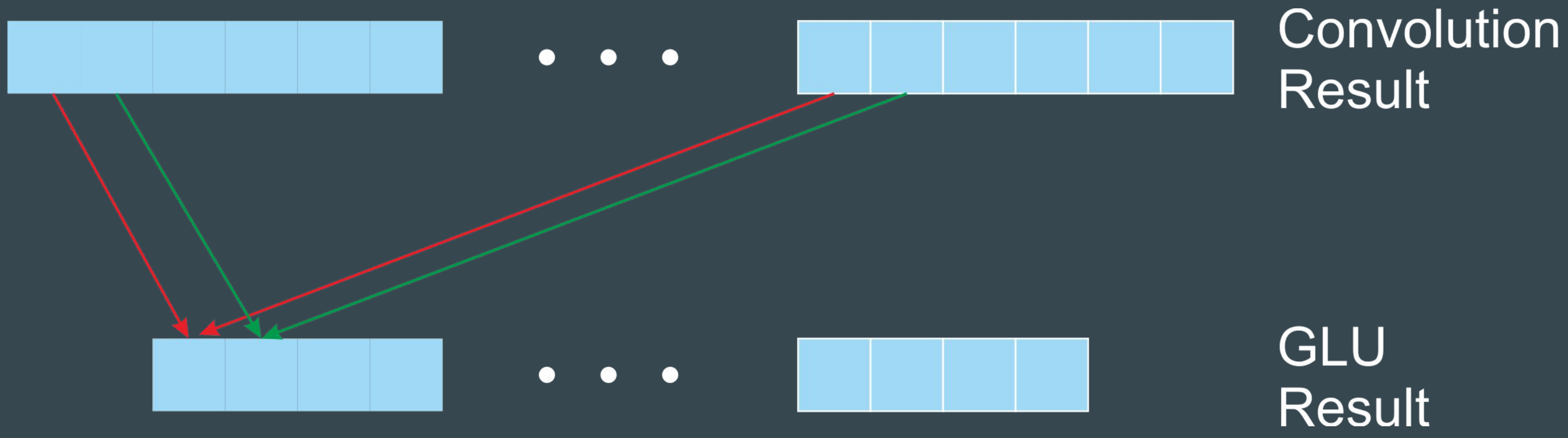
❑ Operating on contiguous data of the input increase locality.



CUDA implementation (cont'd)

The GLU divides the data into two parts of equal size and operates on one element from each parts at a time.

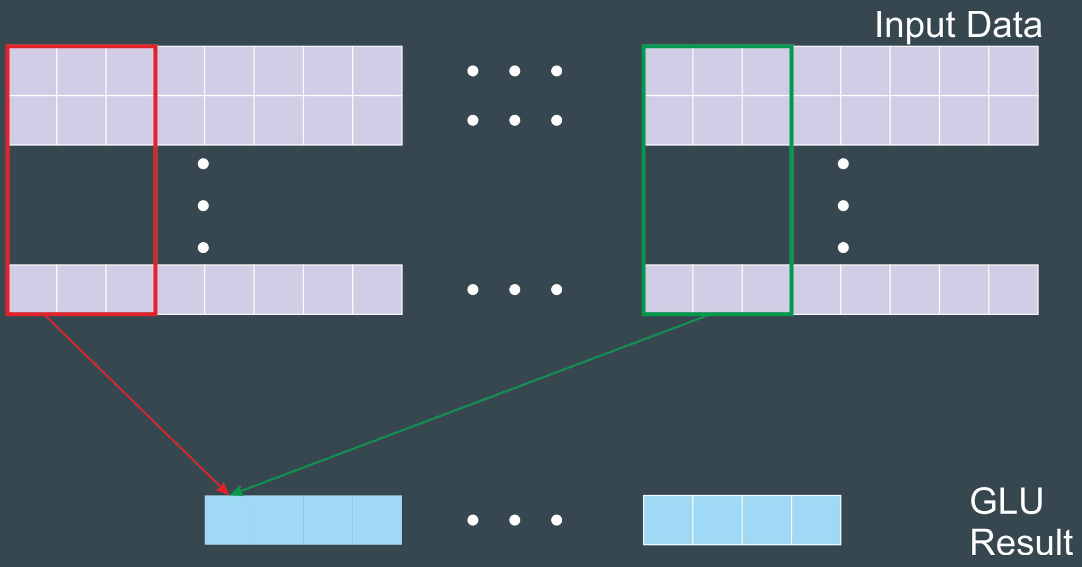
❑ Access pattern needs to be considered to fuse the GLU and Convolution together.



CUDA implementation (cont’d)

To enable GLU fusing, we need to modify the convolution operations so that it can produce two results required for GLU.

- ❑ Losing some locality for convolution because of non-contiguous operation on input data.
- ❑ Guarantee that the convolution results are still stored in register.
- ❑ Minimize the data that needs to be stored back into memory.



CUTLASS approach

CUTLASS is an open source “template” library that provides for fast linear algebra in CUDA and C++.

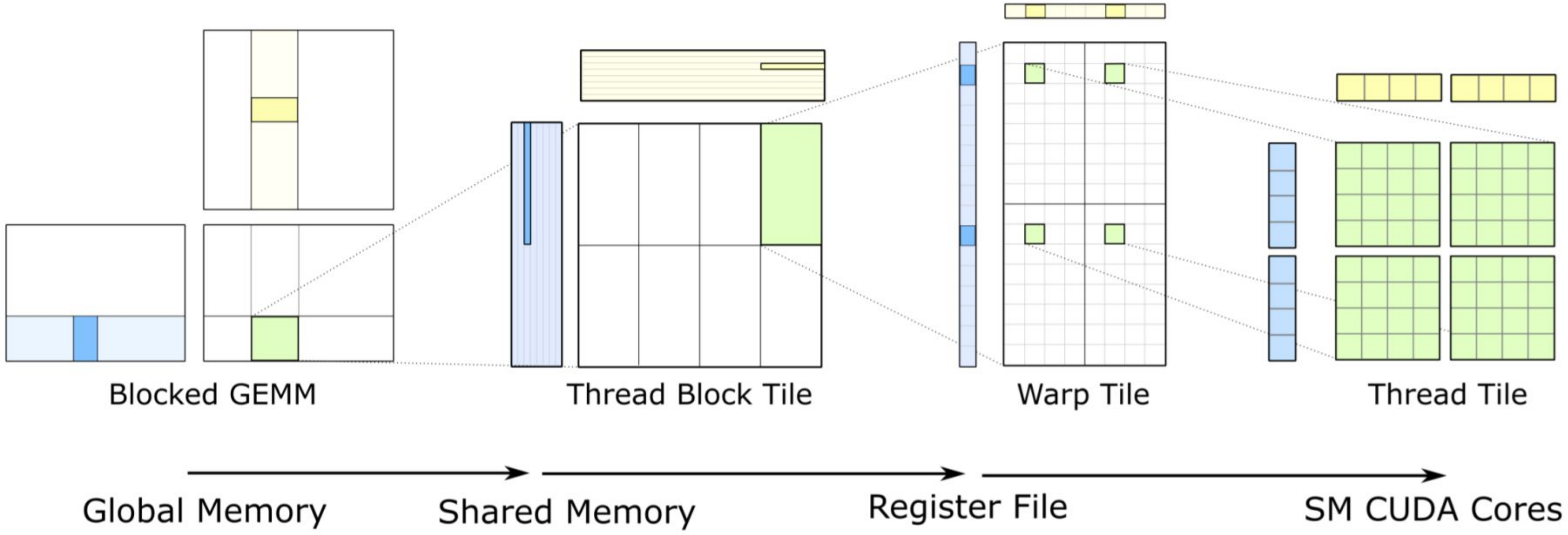
- The library is made available by NVIDIA in the year 2018.
- It is supposed to perform around 90-95% efficient relative to cuBLAS.
- Aims to provide templates for kernel fusion as well.
- The code is well optimized for gemm kernels.
    - Multi-level blocking
    - Software pipelining
    - Double-buffering
    - many more

CUTLASS limitations

- A major drawback is that the library’s documentation is negligible (one blog post, and a few slides).
- Additionally, there is no template provided for convolution.

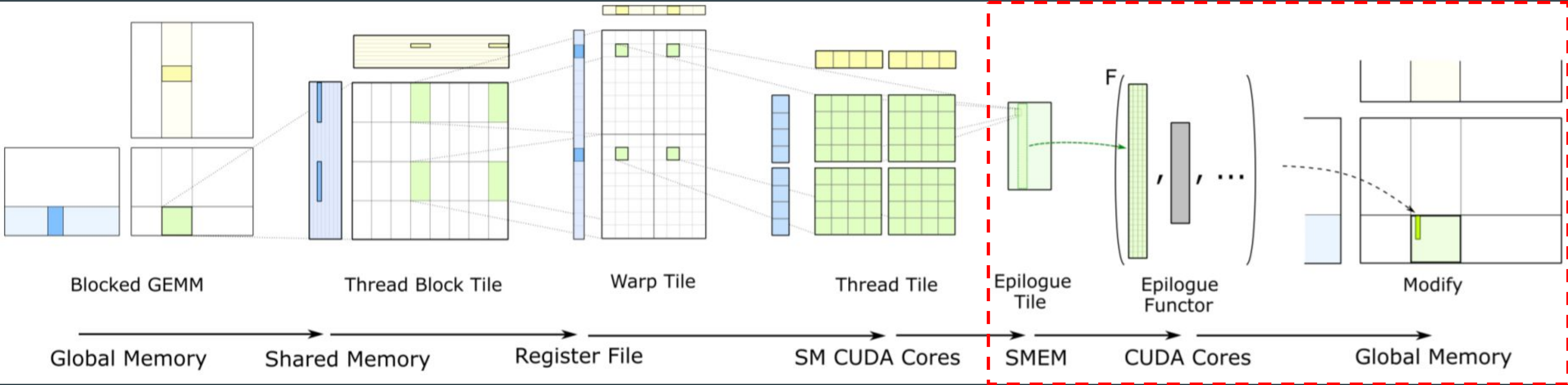
Note that, being a template library the repository consisted of a lot of header files (single precision gemm, double precision gemm, warp-synchronous matrix multiply-accumulate, etc.) with limited number of examples.

CUTLASS implementation



As these optimizations are similar to that introduced in class (multi-tiling), these were relatively easy to understand and work with.

More on CUTLASS implementation (challenges)



However, these new templates were very difficult to understand and modify.

- We were successful in performing convolution via gemm calls using the library templates. Note that, this requires significant understanding of the library code.
- But we were not able to add GLU as a new Epilogue Functor in the given time.
- This was mainly because GLU reduces the dimension of the Tensor and this support was not provided by CUTLASS library (currently supports functors like ReLU). This change would require significant changes in the code which wasn't possible in the given time.

cuDNN approach

To look at how well we can manage the data from a high level we also tried to improve data locality by appropriately using the cuDNN library calls.

Note that, this approach is not “kernel-fusion” because we can only call cuDNN kernels from our C++ code.

Additionally, using cuDNN is not trivial, one has to read tutorials and documentation to understand it’s working.

cuDNN workflow

1. Create cuDNN context.
2. Create and set descriptors according to input and weight dimension size.
3. Make sure the data is located on the GPU (data pointers take aliases if already on GPU, otherwise create GPU tensor and copy over)
4. Execute forward functions (`cudnnConvolutionForward()`, `cudnnActivationFoward()`).
5. Package and return results.
6. Execute backward functions to get the gradients of bias, inputs and weights respectively (`cudnnActivationBackward()`, `cudnnConvolutionBackwardBias()`, `cudnnConvolutionBackwardData()`, `cudnnConvolutionBackwardFilter()`).

cuDNN challenges faced

The default tensor layout that cuDNN accepts to perform convolution is NCHW or NHWC, none of it aligns with TBC.

After struggling a while we figured out that, effectively, we can make NCHW from TBC by provisioning a stride on each dimension.

Due to lack of time, we weren’t able to finish the cuDNN implementation in time, however, we did learn the way to use the same.

TBC dim = {2, 2, 2}
TBC stride = {4, 2, 1}

NCHW dim = {2, 2, 1, 2}
NCHW stride = {2, 1, 4, 4}

T0 B0 C0
T0 B0 C1
T0 B1 C0
T0 B1 C1
T1 B0 C0
T1 B0 C1
T1 B1 C0
T1 B1 C1

Integrating with PyTorch: adding C++ and CUDA extensions

Initially it seemed very difficult for us in how to call our own C++ and CUDA functions from the PyTorch. It looked very complicated to extend the libraries as the functions are wrapped in wrappers which were then wrapped in some other wrappers and so on (they even used YAML which we didn’t understand).

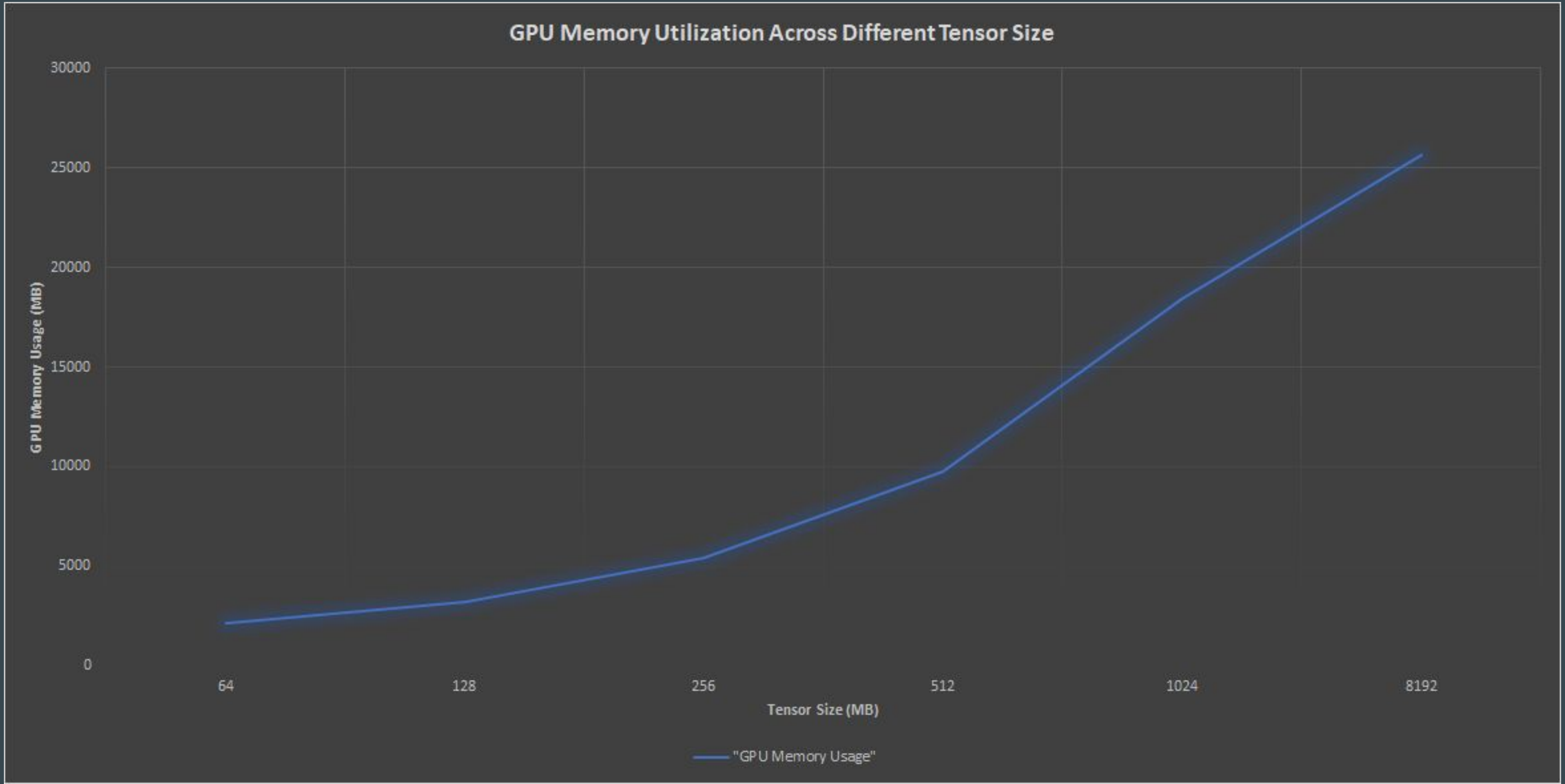
After struggling for a significant amount of time...

Fortunately, on May 7th Peter Goldsborough added a tutorial on PyTorch’s official page on “Custom C++ and CUDA extensions”. This document made our life easier and we were able to integrate our C++ and CUDA extensions on PyTorch.

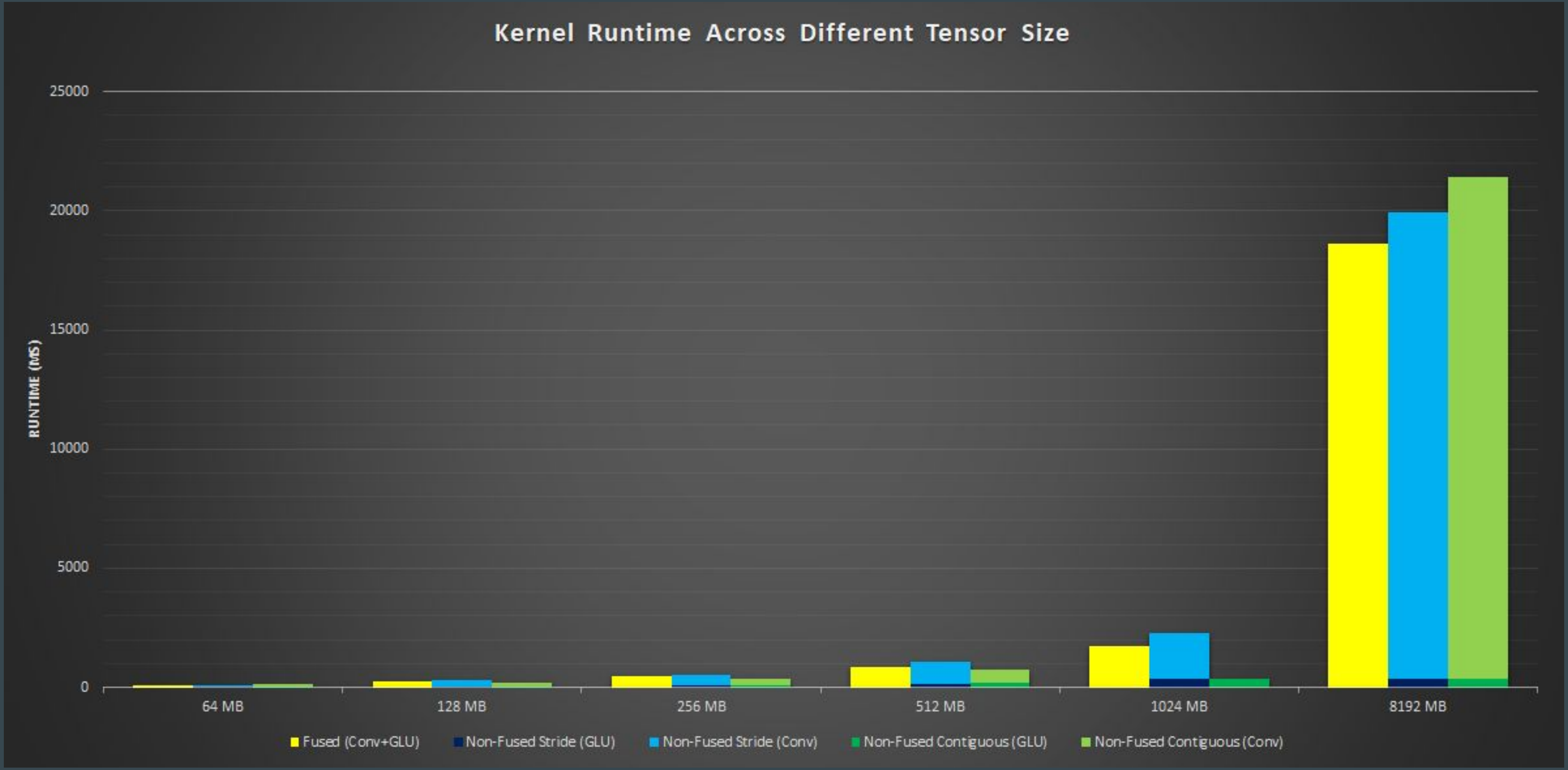
Results and main insights:

Observed on CUDA implementation.

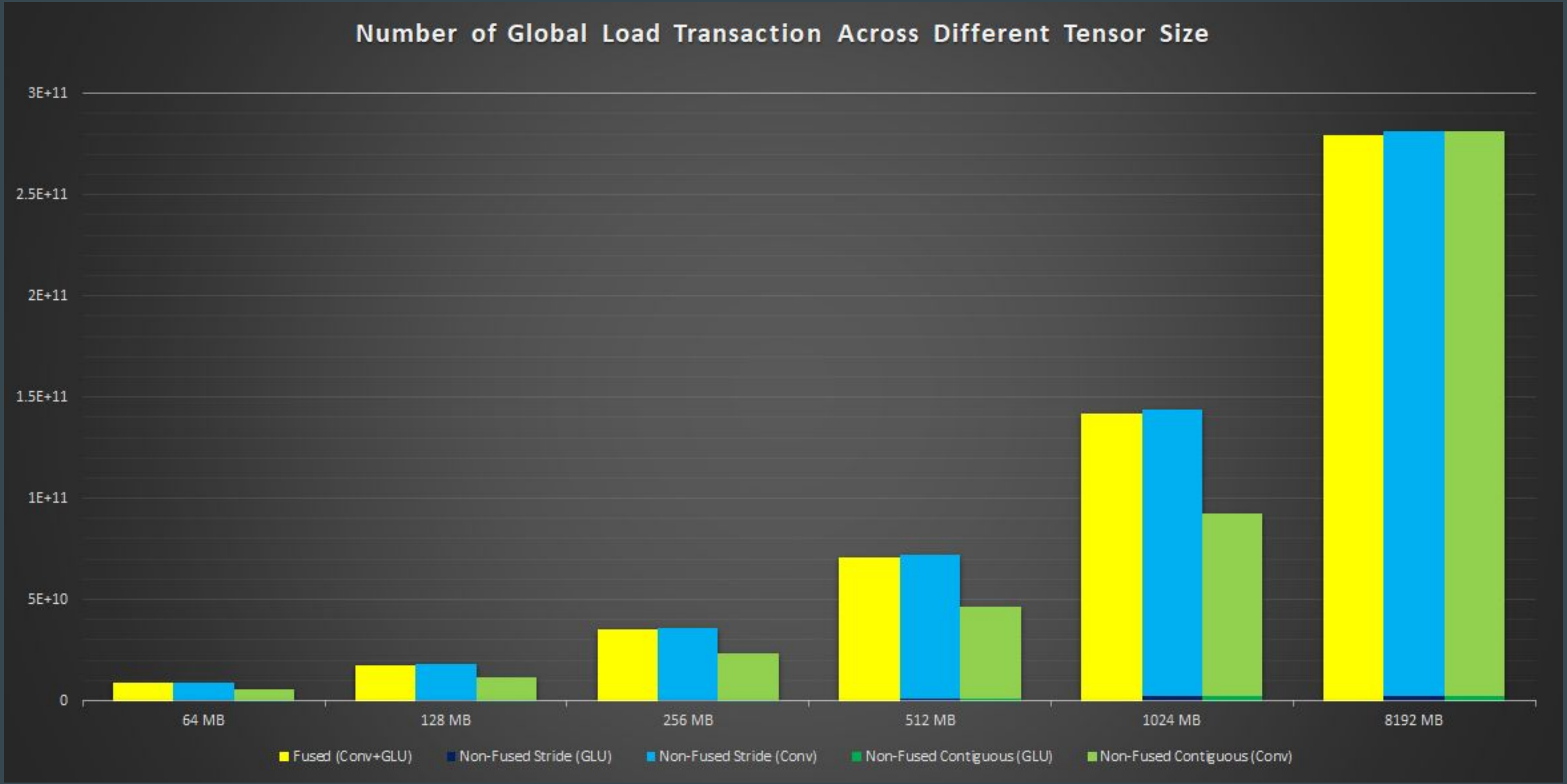
Results - Memory usage



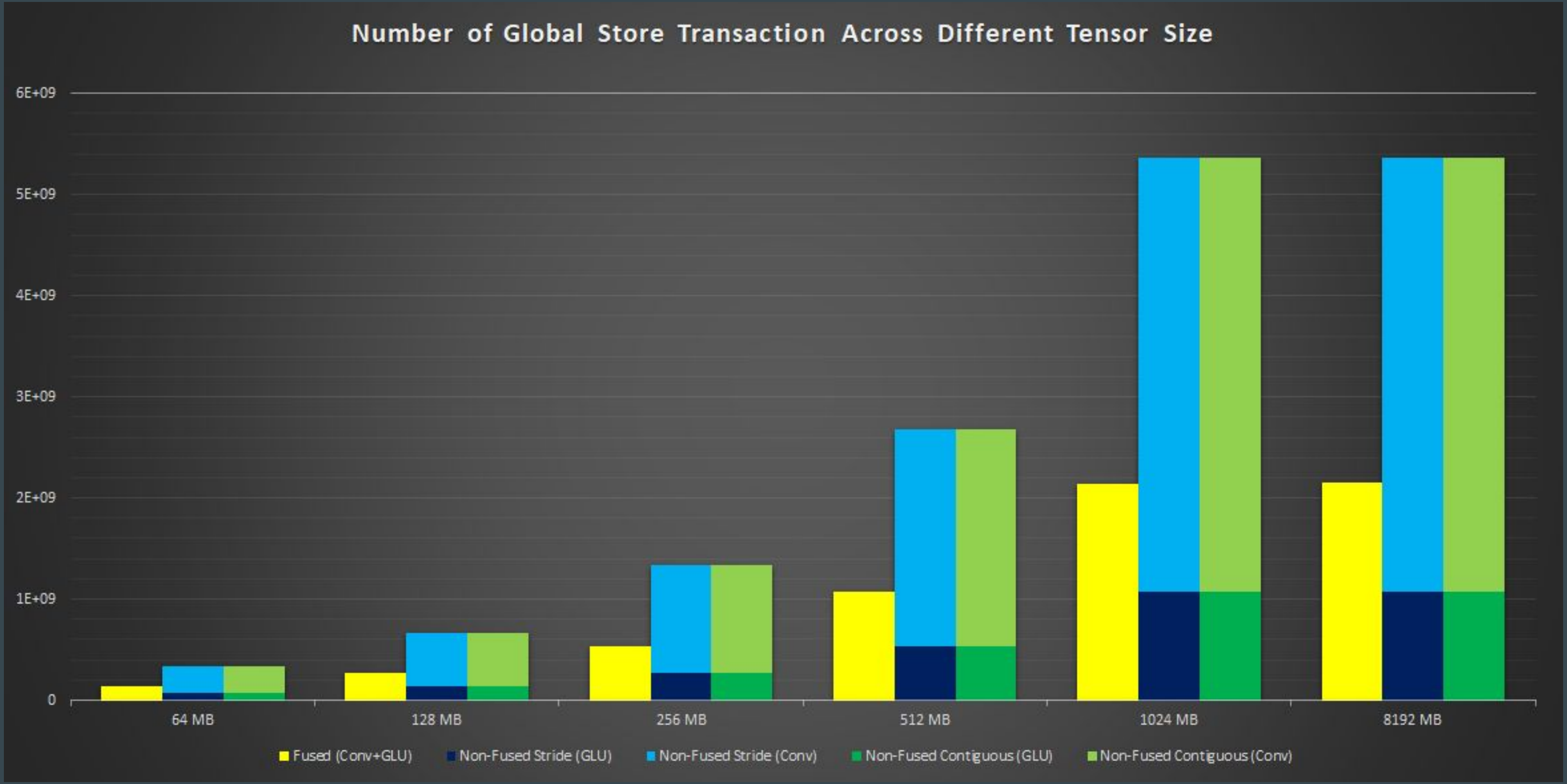
Results - Performance in seconds



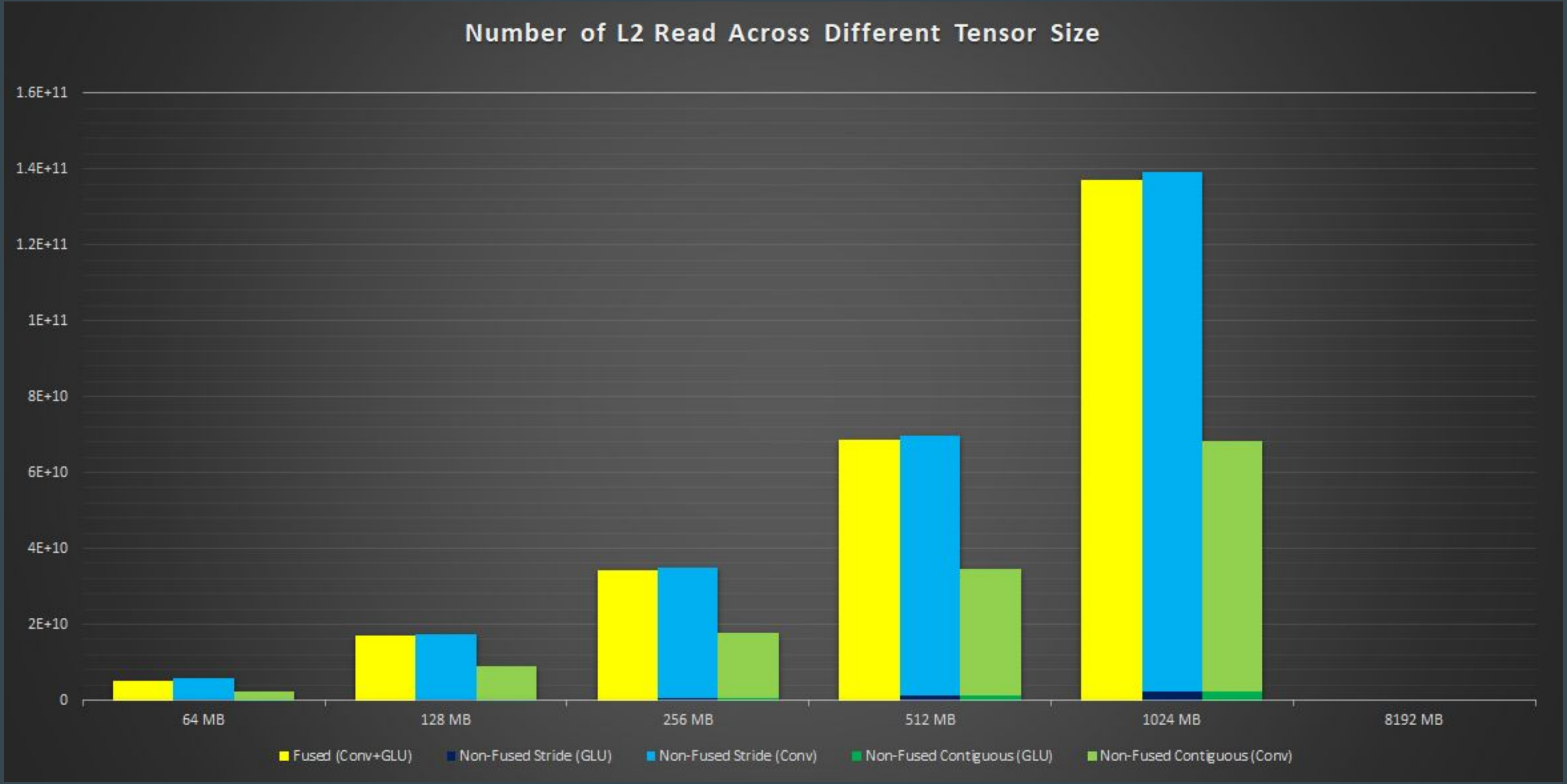
Results - Global Memory loads



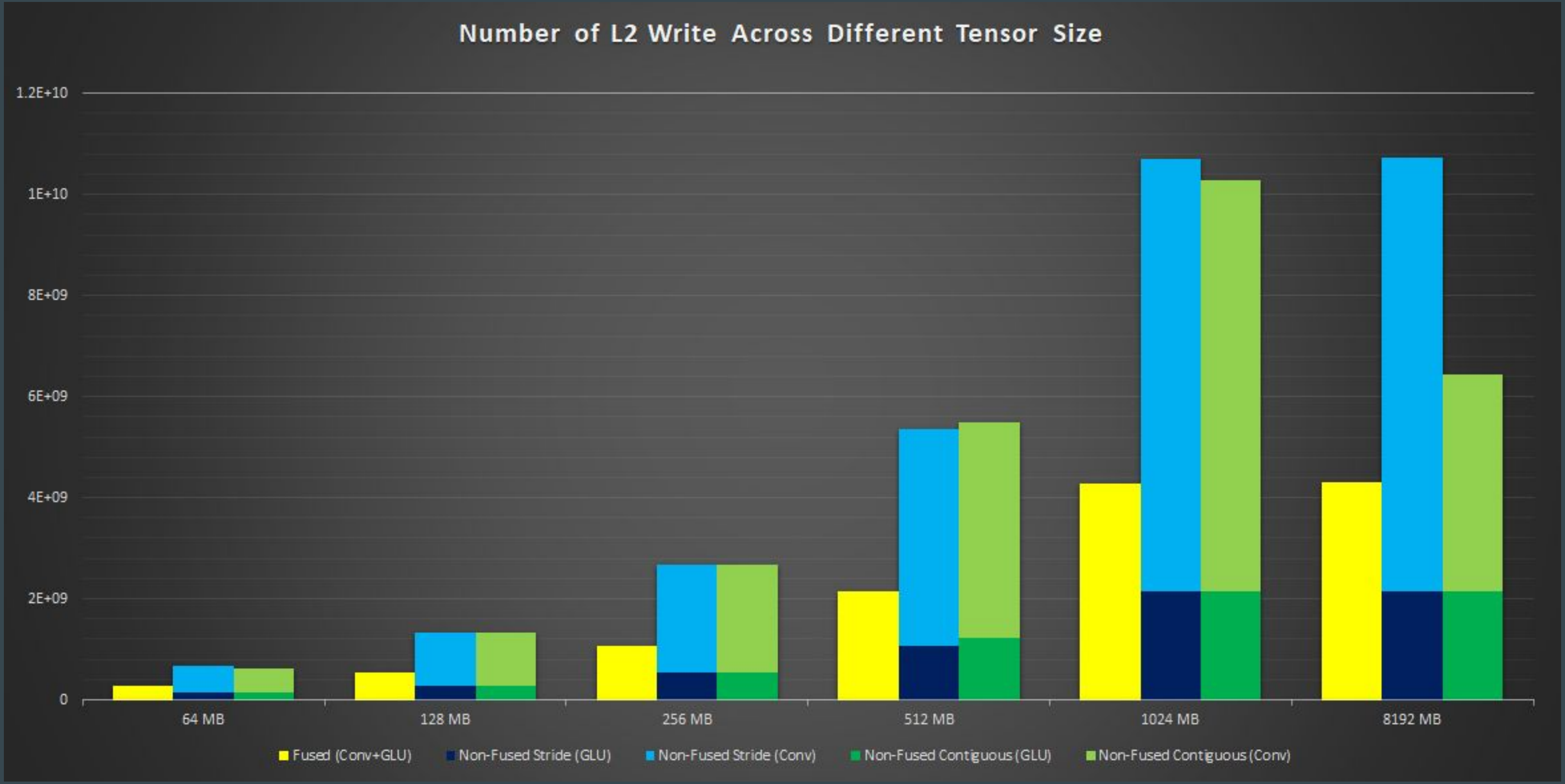
Results - Global Memory stores



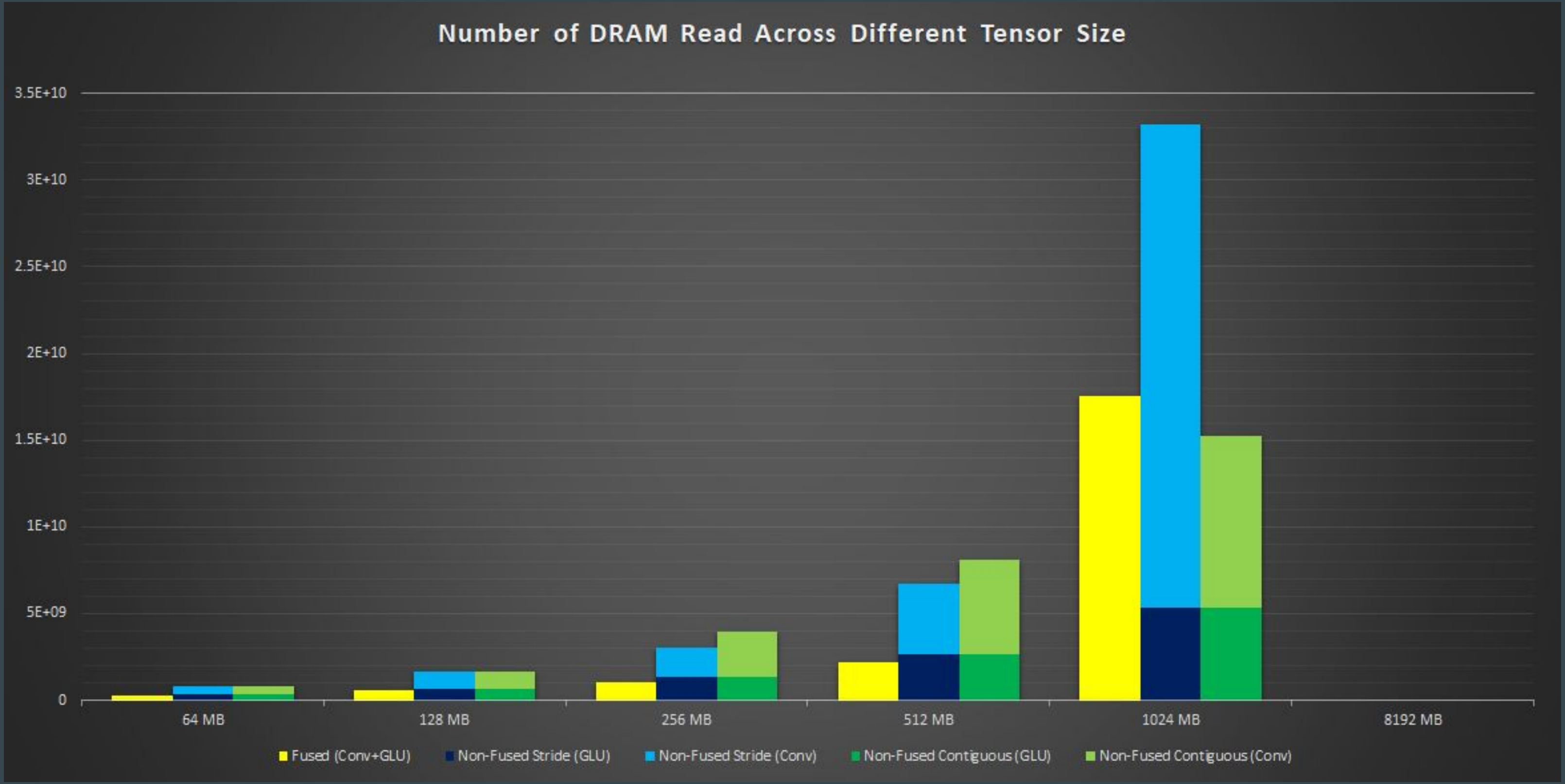
Results - L2 Read



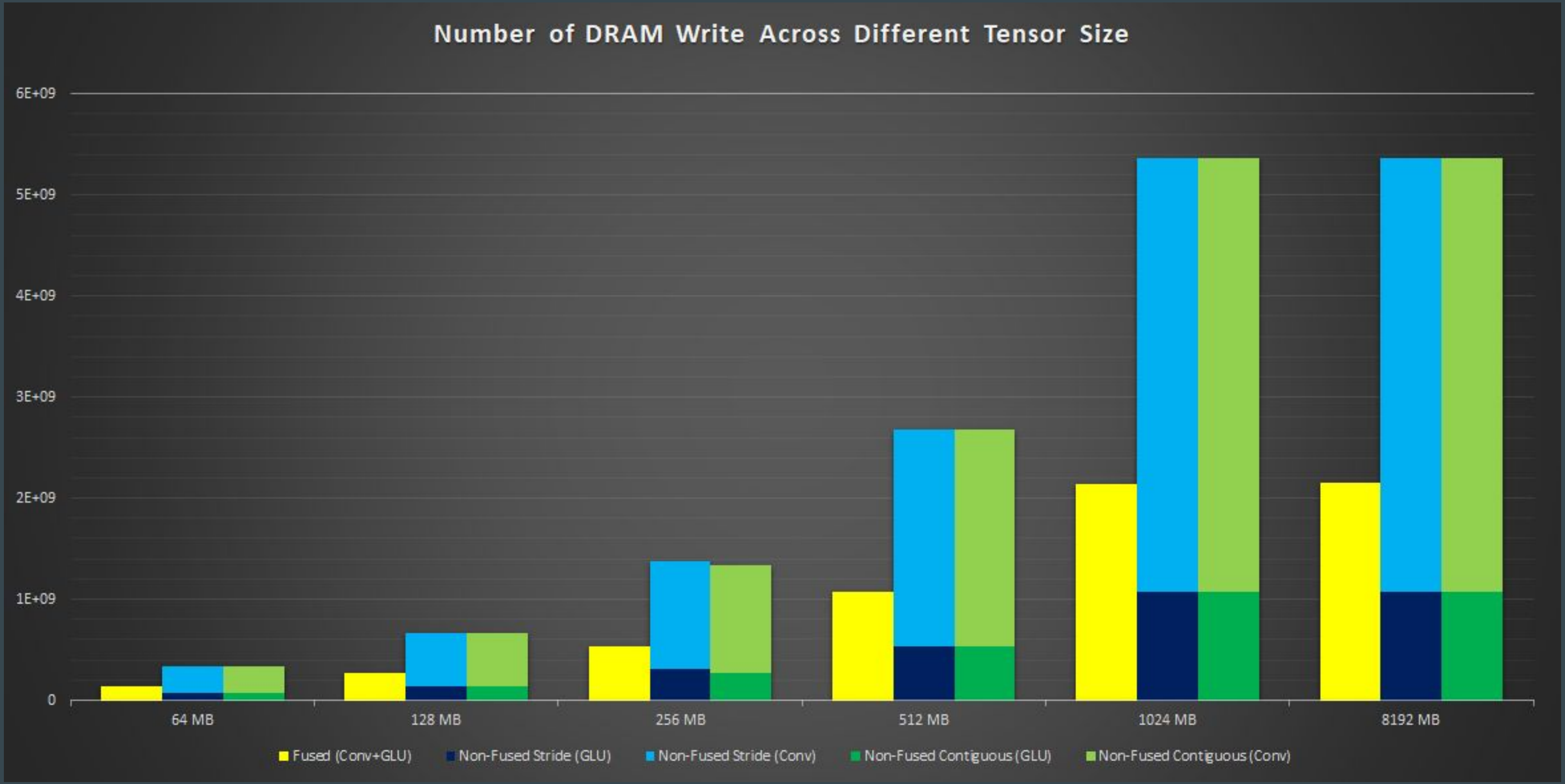
Results - L2 Write



Results - DRAM Read



Results - DRAM Write



Failed attempts - Computational graphs

In order to get a good overview of the neural network model, we tried various tools to generate the computational graph:

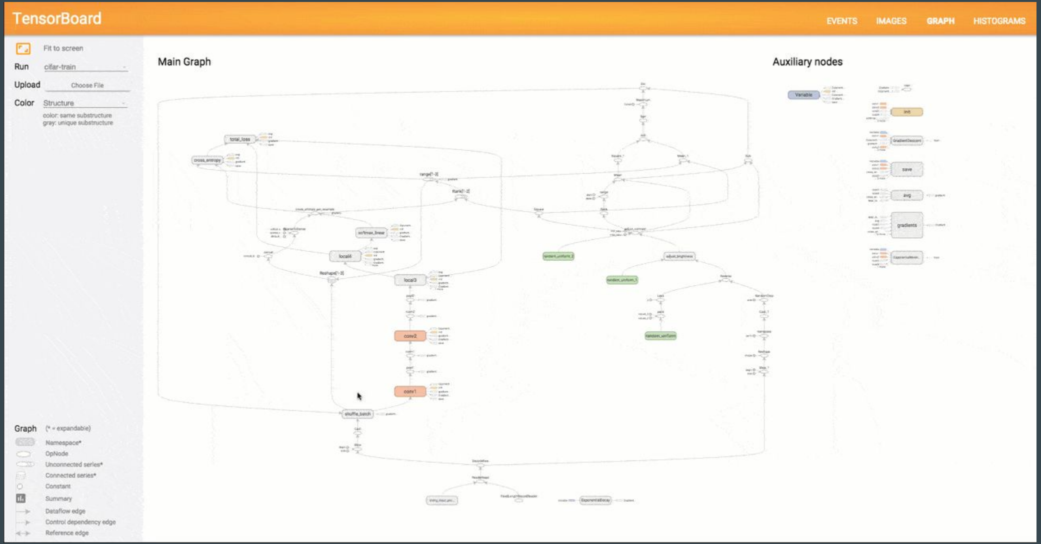
- TensorBoardX (Generalized version of TensorBoard which was custom made for TensorFlow)
  - However, we were only able to generate plots for loss function and scalars.
  - Lack of tutorials/documentation for generating graphs with TensorBoardX.
- HiddenLayer (A lightweight library by Waleed Abdulla and Phil Ferriere)
  - As far as we understand this tool works well for models included within the framework.
  - Lack of tutorials/documentation for a custom model.

Hence, we followed the codebase to understand the working of the model.

TensorBoard

TensorBoardX

https://www.tensorflow.org/images/graph_vis_animation.gif

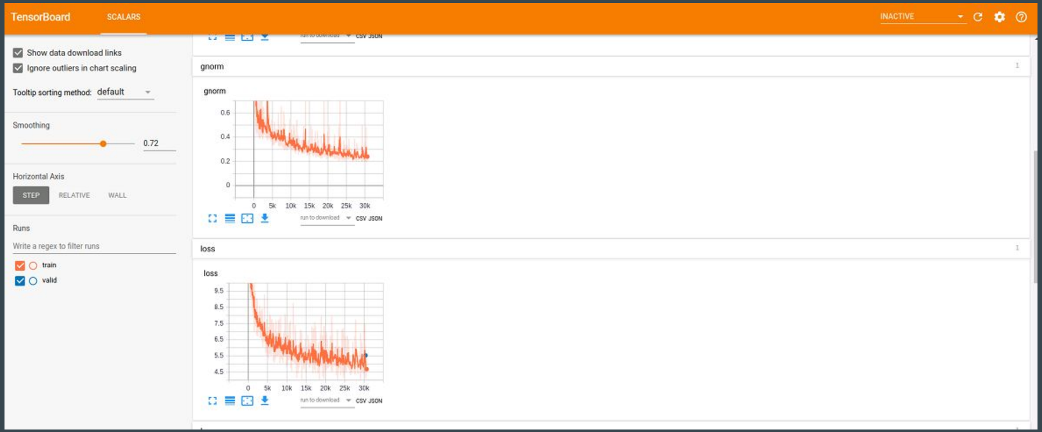


Run: cifar-train

Upload: Choose File

Color: Structure

color: same substructure
gray: unique substructure



☑ Show data download links

☑ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing: 0.72

Horizontal Axis: STEP RELATIVE WALL

Runs

Write a regex to filter runs

☑ ○ train

☑ ○ valid

Failed attempts - Matrix Multiplication

1. We attempted to optimize matrix multiplication ourselves for the V100 GPU.
2. We followed CUTLASS's algorithm for multi-level blocking of the matrix.
3. Since fairseq's convolution is done by a series of matrix multiplications with the same input but different kernels, we thought there could be reuse by fusing the multiplication together. However, due to the issues stated below, the locality would have been minimal as we would have launched many kernels.
4. We ran into the following issues:
    a. At each level there are so many variations of the size of the blocking that finding the optimal blocking numbers specific to the V100 was a great task.
    b. The tiling scheme appears to use the same parts of the data in many different blocks - however, because each block uses separate shared memory, the amount of shared memory taken up per kernel increases proportionally to the number of blocks launched. This means we needed to further block into smaller sizes to be able to fit on the GPU.

Failed attempts - Matrix Multiplication continued

- We suspected that the most optimal matrix multiplication algorithm would need to make use of the tensor cores - we learned that we could assign matrix multiply and accumulates to the tensor cores as long as the dimensions were multiples of 4 and had the correct data type.
- Because the tensor cores require half precision, and the fairseq implementation is in single precision, we considered switching to half precision just to be able to use the tensor cores for the fastest matrix multiply.
- However, we did not implement a matrix multiply with tensor cores - instead, only a blocking scheme at a kernel/block/warp/thread level was implemented.

Failed attempts - nvprof profiling

- We aimed to compare the fused vs unfused implementations over various metrics and explain the results using the nvprof metrics. However, it took a lot of time for us to profile the baseline fairseq implementation. We have results for the same on https://github.com/UT-LCA/FusedConvGLU/tree/master/nvprof , however, we didn't have enough resources to profile our own implementations.
- We profiled the following metrics: `inst_per_warp`, `branch_efficiency`, `shared_store_transactions`, `shared_load_transactions`, `local_store_transactions`, `gld_transactions`, `gst_transactions`, `sysmem_read_transactions`, `sysmem_write_transactions`, `l2_write_transactions`, `dram_write_transactions`, `global_hit_rate` .
- Additionally, we profiled the following events: `shared_ld_transactions`, `shared_st_transactions`, `generic_load`, `generic_store`, `global_load`, `global_store`, `local_load`, `local_store`, `shared_load`, `shared_store` .

What did we learn?

1. Closely understood the working of CNNs specifically concerning Translation (encoders, decoders, and attention layer).
2. A decent understanding of the working of PyTorch and its interface with the C++ and CUDA libraries.
3. Working with open source template library - CUTLASS
4. Working with cuDNN.
5. Data locality optimizations in CUDA by kernel fusion.
6. Extending PyTorch with custom C++ and CUDA functions.
7. One main thing we learnt is that we should have planned the timeline appropriately. We tried to cover a wide breadth but we weren't able to finish everything in time which lead to poor evaluation.

Future goals

1. The CUDA code can be optimized further.
2. As mentioned earlier, the CUTLASS and cuDNN implementations were limited due to time constraints. We aim to complete these approaches.
3. The evaluation is weak. We plan on strengthening it by comparing various approaches (fused vs. unfused) among each other and with the reference fairseq code, that makes use of cuBLAS library, over various metrics (including BLEU score).
4. Since the benefit obtained only by fusing convolution layer with GLU layer is limited, fusing convolution layer with GLU and Attention layers would show significant performance improvements.

Acknowledgements

We would like to thank

- Qinzhe Wu
- Sangkug Lym
- Dr. Ramesh Radhakrishnan
- TAs: Yongkee Kwon and Kyushick Lee
- Prof. Mattan Erez

for their continuous advice and for making the hardware available.

Thank You!

Some of the code can be found @ https://github.com/UT-LCA/FusedConvGLU

Questions?