CSCI 6461 Final Project - Profiling GPU Efficiency on Jetsons

By Karl Simon

Code: https://github.com/KarlSim24/csci6461_cuda.git

Overview:
The goal of this project was to explore CUDA acceleration for common deep learning operations such as matrix multiplication and model inference, and to evaluate which types of applications benefit most from GPU acceleration. I used an NVIDIA Jetson Orin Nano as my deployment board, which will eventually be used for real-time deep learning tasks in my masters research. The project was divided into two parts: the first focused on low-level CUDA kernel performance compared to PyTorch's high-level operations, and the second evaluated deep learning inference speed using PyTorch and TensorRT.

Part 1: CUDA Kernel Performance vs PyTorch Matrix Operations

In the first part of this project, I explored how raw CUDA kernel implementations compare to PyTorch's high-level matrix operations in terms of runtime efficiency. The experiment focused on matrix multiplication, which is a common operation used in forward passes through fully connected neural networks. The goal was to understand how low-level memory control and thread management in CUDA work and to get a tangible sense of how optimized the high-level libraries like PyTorch really are.

a) On an NVIDIA A100 GPU server, I ran a simple benchmark CUDA kernel (**Part1/naive_matmul.cu**) for square matrix multiplication, and benchmarked its runtime on matrices of increasing sizes 128×128 up to 2048×2048. Additionally, I tested an equivalent PyTorch-based implementation (**Part1/pytorch_matmul.py**) which ran on both CPU and GPU backends.
b) I repeat part a), but now on the Jetson Orin Nano's GPU for speed comparison. I ran them in their MAXN Super power mode, which allows for maximum clock speeds and all available CPU cores to be used.

Results:

After averaging many runs of these two scripts, it was clear that the naive CUDA kernel was significantly slower than PyTorch's CUDA backend. These results showed that PyTorch's libraries which implement their own optimized CUDA kernels such as cuBLAS outperformed my hand-written kernels. Even when I tested a shared memory custom kernel which used tiling (see **Part1/shmem_matmul.cu**), performance was significantly slower than PyTorch's

matmul function. This test showed me that unless you need a very particular function to be run on a GPU, for most cases the PyTorch libraries are actually surprisingly efficient and should be used if possible, especially on Jetsons where inference time is crucial for supporting real-time operation.

i)

```
(cuda_project) karlsimon@fdcl1:~/csci6461/final/csci6461_cuda$ ./matmul
Matrix size: 128x128 | Time: 283.782 ms
Matrix size: 512x512 | Time: 1.14644 ms
Matrix size: 1024x1024 | Time: 3.72943 ms
Matrix size: 2048x2048 | Time: 16.6523 ms

(cuda_project) karlsimon@fdcl1:~/csci6461/final/csci6461_cuda$ python pytorch_matmul.py
Benchmarking PyTorch matmul on cpu
Size: 128×128 | Avg Time: 0.060 ms
Size: 512×512 | Avg Time: 0.199 ms
Size: 1024×1024 | Avg Time: 1.061 ms
Size: 2048×2048 | Avg Time: 8.819 ms
Benchmarking PyTorch matmul on cuda
Size: 128×128 | Avg Time: 0.030 ms
Size: 512×512 | Avg Time: 0.037 ms
Size: 1024×1024 | Avg Time: 0.139 ms
Size: 2048×2048 | Avg Time: 0.993 ms
```

1) GPU Server results

ii)

```
root@ubuntu:/workspace# nvcc naive_matmul.cu -o matmul
root@ubuntu:/workspace# ./matmul
Matrix size: 128x128 | Time: 155.183 ms
Matrix size: 512x512 | Time: 11.5607 ms
Matrix size: 1024x1024 | Time: 66.6798 ms
Matrix size: 2048x2048 | Time: 199.22 ms

root@ubuntu:/workspace# python3 pytorch_matmul.py
Benchmarking PyTorch matmul on cpu
Size: 128×128 | Avg Time: 0.242 ms
Size: 512×512 | Avg Time: 7.368 ms
Size: 1024×1024 | Avg Time: 36.232 ms
Size: 2048×2048 | Avg Time: 177.358 ms
Benchmarking PyTorch matmul on cuda
Size: 128×128 | Avg Time: 0.191 ms
Size: 512×512 | Avg Time: 0.873 ms
Size: 1024×1024 | Avg Time: 8.737 ms
Size: 2048×2048 | Avg Time: 20.938 ms
```

1) Jetson Results

Part 2: Jetson Orin Nano: Simple MLP in pytorch vs TensorRT+cuDNN

The second part of the project shifted from low-level matrix operations to higher-level deep learning inference. Here, the goal was to evaluate how model structure and computation affect GPU performance, specifically using PyTorch and TensorRT for optimization. Two different benchmark models were trained and tested on the Jetson Orin Nano's GPU:

a) a simple MLP designed to predict the largest eigenvalue of a symmetric 4x4 matrix (to mimic SE(3) transformation matrices I use in my research)
b) simple convolutional neural network trained for MNIST digit classification.

After training these, I used **tensor2trt** library to optimize the trained model. This simply takes various blocks of the PyTorch model and converts certain operations into TensorRT API calls that run even more optimized. For instance, one of the main benefits to TensorRT is its ability to quantize model weights to different floating point levels, allowing for speed improvements at slight precision tradeoffs. For the sake of this project, those tradeoffs were negligible.

Results:

a) Eigenvalue Model:
    i) The model that predicts the largest eigenvalue of a symmetric matrix didn't have a significant difference in inference time between the optimized TensorRT vs regular PyTorch models. PyTorch inference averaged 0.0014 seconds per batch, while TensorRT inference dropped only slightly to 0.0012 seconds. The reason for this was likely that the model was too "small" to take advantage of RT's optimizations. This task I chose was likely of low arithmetic intensity, meaning delays were less due to computation overhead and more due to memory constraints.

```
root@ubuntu:/workspace/Part2# python3 mlp_eigval.py
Using device: cuda
Training MLP to learn largest eigenvalue of symmetric matrices...
Epoch 000 - Avg Loss: 0.293829
...
Epoch 099 - Avg Loss: 0.000976
Training completed in 77.92 seconds.
Model saved to mlp_eigenvalue_model.pth
True largest eigenvalue: 1.1654
Predicted:               1.1697

root@ubuntu:/workspace/Part2# python3 convert_trt.py
/workspace/Part2/convert_trt.py:13: FutureWarning: You are using `torch.load` with `weights_
  model.load_state_dict(torch.load("mlp_eigenvalue_model.pth"))
TensorRT model saved.
[04/17/2025-16:54:41] [TRT] [W] Using default stream in enqueueV3() may lead to performance
PyTorch inference time: 0.0014 sec
TensorRT inference time: 0.0012 sec
```

    i)

b) Conv Model
    i) This model was simple, but consisted of convolutional layers on image (i.e. matrix) data. Due to common code reuse and much more information needing to be captured within the model layers, TensorRT showed a significant speedup. Specifically, convolutional layers (unlike the linear fully connected layers in a)) have high data reuse and can be easily

parallelized. Further, the reduced precision on the model weights (FP16) played a role since there were now millions of params rather than thousands as before. All together, this was a clear demonstration to me that the these optimizations are workload-dependent and certain tasks benefit more than others from hardware-specific optimizations.

ii)

```
root@ubuntu:/workspace/Part2# python3 conv_model.py
Using device: cuda
Training ConvNet on MNIST...
Epoch 001 - Avg Loss: 0.125675
Epoch 002 - Avg Loss: 0.040868
Epoch 003 - Avg Loss: 0.028934
Epoch 004 - Avg Loss: 0.020561
Epoch 005 - Avg Loss: 0.016395
Training completed in 191.32 seconds.
Model saved to mnist_conv_model.pth
Test Accuracy: 98.74%
root@ubuntu:/workspace/Part2# python3 convert_trt_conv.py
/workspace/Part2/convert_trt_conv.py:12: FutureWarning: You are using `torch.load` with `wei
  model.load_state_dict(torch.load("mnist_conv_model.pth"))
TensorRT model saved to mnist_conv_model_trt.pth
[04/17/2025-17:09:31] [TRT] [W] Using default stream in enqueueV3() may lead to performance
Avg PyTorch inference time: 0.003392 sec
Avg TensorRT inference time: 0.000550 sec
```

Conclusion:

This exploration into how CUDA is used to take advantage of GPUs, and how “simple” model optimizations provided by libraries like TensorRT can significantly increase speedup in inference and prediction tasks. However, it’s not a magic bullet, and understanding what the application is doing, specifically its architecture (if it's a deep learning model) or its mathematical operations (matrix multiplication vs simple addition), is crucial. I also realized that writing custom CUDA kernels that can *outperform* (in terms of speed) PyTorch is very difficult. But by understanding in which data structures and operations PyTorch + CUDA are applied in my code, I may be able to reduce some more latency by parallelizing operations at a higher level, such as by restructuring batch processing or reducing unnecessary CPU-GPU memory transfers which occur quite frequently. Tying it back to my masters research, this was the first time I saw how PyTorch uses CUDA behind the Python API to achieve its performance, and gave me a sense of how feasible optimizations really are. Since my work deals with real-time point cloud processing, this is likely a very useful domain to test TensorRT optimizations with.