1 Background and Experimental Methodology

GPUs have been central to accelerating ML training over the past decade. However, given the varied nature of training workloads across different application models, obtaining optimal performance from ML systems is still a challenge. In this study, we focus on the case study of DeepSpeech to understand the memory and compute facets of training on a Nvidia Pascal P100 GPU. This study was carried out using a combination of profiling tools to characterize training performance.

1.1.1 Workload

DeepSpeech[1] is an open-source speech-to-text engine implemented using optimized Recurrent neural networks that play a key role in automatic speech recognition pipelines. To exploit data parallelism available on GPUs, multiple examples from the dataset are processed at once during training. This form of training is referred to as batch mode. The training is carried out using multiple batches, iterations, and epochs. For our measurements we consider 10 batch-iterations in an epoch. In this context, these terms have the following meaning:

- Batch size: Number of training examples present in a single batch
- Iterations: Number of batches needed to complete a single epoch
- Epoch: Number of times the entire training data set is passed through the training model. Multiple passes (forward & backward) are needed on the same dataset to reduce training error.

We used a PyTorch implementation [4] of DeepSpeech2 built using the PyTorch Lightning framework for analysis. For the training dataset, we used the LibriSpeech ASR corpus[3] that contains 100 hours of speech and spans 6.3GB.

1.1.2 Hardware

We use a single Nvidia Pascal P100 GPU [5] for training. The device has the following configuration:

- Compute: 56 Streaming Multiprocessors @ 1.3 GHz, 250W TDP, 32 Thread Blocks per SM, 2048 Threads per SM
- Memory: Configurable 64KB shared/L1, 4096KB L2, 8x512b memory controllers, 16GB HBM2 DRAM
- Support for precision arithmetic: single FP16 (21.2 TFLOPS), half FP32 (10.6 TFLOPS), double FP64 (5.3TFLOPS)

The P100 is particularly known to be the first GPU with HBM2 package that significantly improves the DRAM bandwidth. One thing to note is the lack of support for Tensor Cores that was first introduced on the succeeding Volta architecture. Tensor Cores on Volta are claimed to deliver 3x faster training compared to Pascal through higher compute efficiency.

1.1.3 Profiling tools

We use a combination of profiling tools listed below for the analysis presented in the next section:

- **NVProf**: The "metrics" mode was used to collect stats about *achieved_occupancy*. The "gpu-trace" mode was useful in finding the launch parameters of kernels.
- **cProfile**: Since frameworks like PyTorch lightning abstract away the steps in training, we found the call trace from cProfile to be useful in understanding training steps and the underlying calls to library functions.
- **PyTorch SimpleProfiler [6]**: Useful in quickly capturing the forward vs backward distribution of runtime since PyTorch has information about the model and each step in the epoch.
- **PyTorchProfiler [9]:** In our experiments, NVProf was found to be very slow, especially with memory measurements. This slowdown is likely due to instrumenting every CUDA API call which grows exponentially on large models. PyTorch on the other hand did not slow down even with verbose metric collection and visualization. As detailed in Section 2.1 PyTorch is sampling based and does not profile every call, but most metrics needed for optimizing ML training were available.

We have captured the profiling commands, code annotation and experiential learning from profiling experiments in a git repository [8]. In the rest of this document, we describe the trends in utilization and bottlenecks observed.

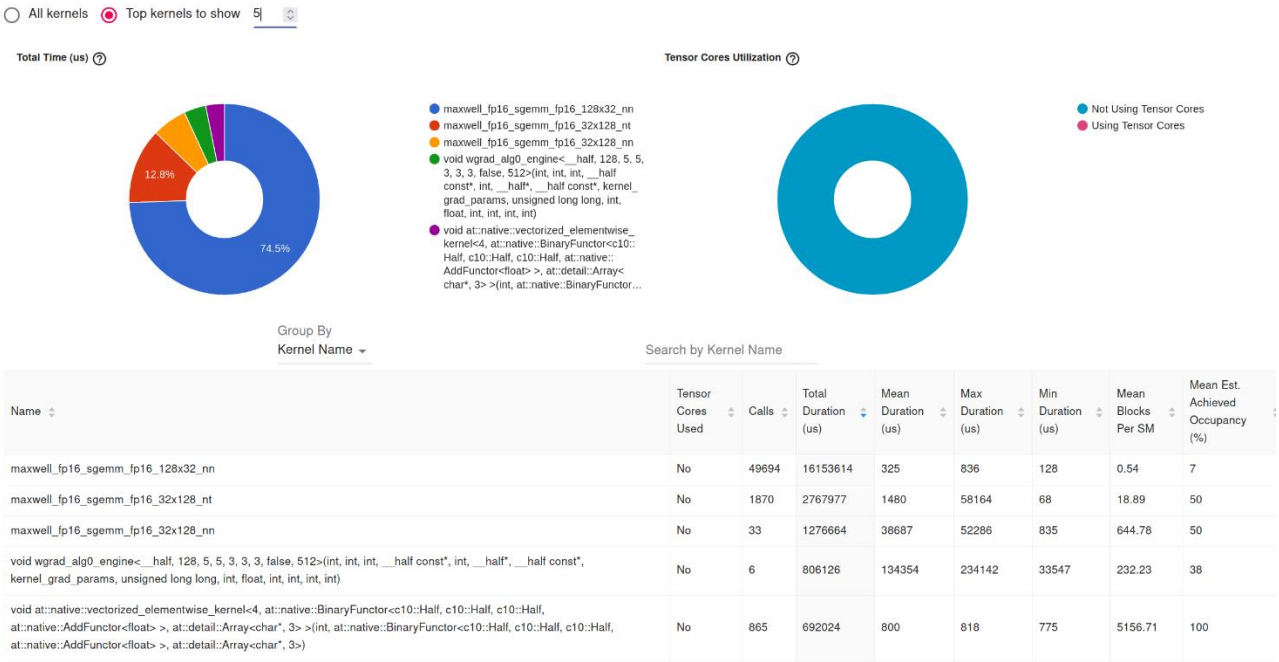
2 Profiling and analysis

2.1 Task 1: Long running kernels in the model

We determined the following kernels to occupy the longest fraction of time by measuring the total time (that includes every invocation of the kernel):

Kernel	Number of calls	Fraction of time	Total runtime
sgemm_128x32_nn	186316	62.43%	56.74s
sgemm_32x128_nt (nt:transpose)	6220	10.17%	9.24s
sgemm_32x128_nn (nn:non-transpose)	130	5.25%	4.77s
wgrad_engine (convolution)	20	2.94%	2.67s
vectorized_elementwise_kernel	2920	2.57%	2.33s

The above metrics were obtained from NVProf. We found that PyTorchProfiler produced nearly the same results with much less runtime overhead. We attribute the difference in estimated statistics such as calls and average time between NVProf and PyTorchProfiler to the fact that the latter likely relies on sampling to avoid significant profiling overheads.



Name	Tensor Cores Used	Calls	Total Duration (us)	Mean Duration (us)	Max Duration (us)	Min Duration (us)	Mean Blocks Per SM	Mean Est. Achieved Occupancy (%)
maxwell_fp16_sgemm_fp16_128x32_nn	No	49694	16153614	325	836	128	0.54	7
maxwell_fp16_sgemm_fp16_32x128_nt	No	1870	2767977	1480	58164	68	18.89	50
maxwell_fp16_sgemm_fp16_32x128_nn	No	33	1276664	38687	52286	835	644.78	50
void wgrad_alg0_engine<__half, 128, 5, 5, 3, 3, 3, false, 512>(int, int, int, __half const*, int, __half*, __half const*, kernel_grad_params, unsigned long long, int, float, int, int, int, int)	No	6	806126	134354	234142	33547	232.23	38
void at::native::vectorized_elementwise_kernel<4, at::native::BinaryFunctor<c10::Half, c10::Half, c10::Half, at::native::AddFunctor<float> >, at::detail::Array<char*, 3> >(int, at::native::BinaryFunctor<c10::Half, c10::Half, c10::Half, at::native::AddFunctor<float> >, at::detail::Array<char*, 3>)	No	865	692024	800	818	775	5156.71	100

Figure 1: Distribution of top 5 kernels using PyTorch Profiler

The *wgrad_alg0_engine* kernel in the gradient normalization layer took the longest per call. While in terms of total fraction of time, sgemm in the 5 RNN layers took the longest.

Achieved Occupancy indicates how many warps can be active at once per SM. However, increasing this beyond a point can have diminishing returns due to resource contention such as memory-bound conflicts [7]. We observe that out of the top 5 kernels, only sgemm_128_32_nn shows lower than acceptable occupancy. The following observations were made regarding this kernel:

- Most invocations of this kernel happen with a small number of thread blocks (4) with few threads (256). The count was obtained using the *gpu-trace* option in NVProf that indicates kernel launch parameters. This observation confirmed that there is unlikely to be a memory bottleneck. Also, from the PyTorchProfiler memory trace, it was evident that the memory utilization never reached the limit.
- The poor occupancy is likely due to (i) Large number of control-flow instructions (Fig. 3) reducing the IPC and thereby occupancy (ii) Small matrix multiplication calls that indicates a serial component (from the trace visualization) in the RNN workload that limits parallelism. But generally, the GPU resources were not observed to be the source of the bottleneck here.

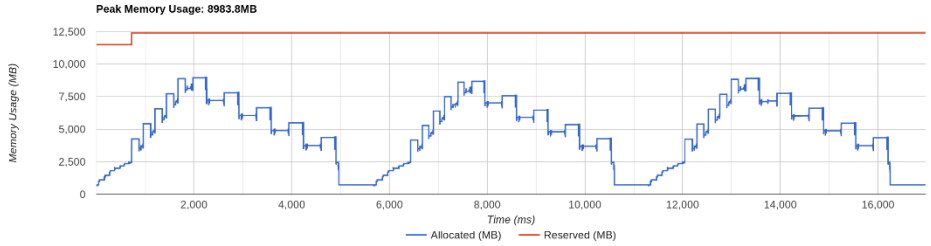


Figure 2: Memory utilization trends for 3 steps with batch-size 64

```
Invocations                              Metric Name                      Metric Description         Min         Max         Avg
Device "Tesla P100-PCIE-16GB (0)"
    Kernel: maxwell_fp16_sgemm_fp16_128x32_nn
          4476                    achieved_occupancy                 Achieved Occupancy    0.124969    0.146554    0.140943
          4476                        sm_efficiency            Multiprocessor Activity      51.38%      88.89%      77.31%
          4476                        cf_executed  Executed Control-Flow Instructions       40704       81408       73114
          4476                                ipc                        Executed IPC    1.152589    1.532222    1.379661
```

Figure 3: NVProf metrics for sgemm_fp16_128x32 kernel

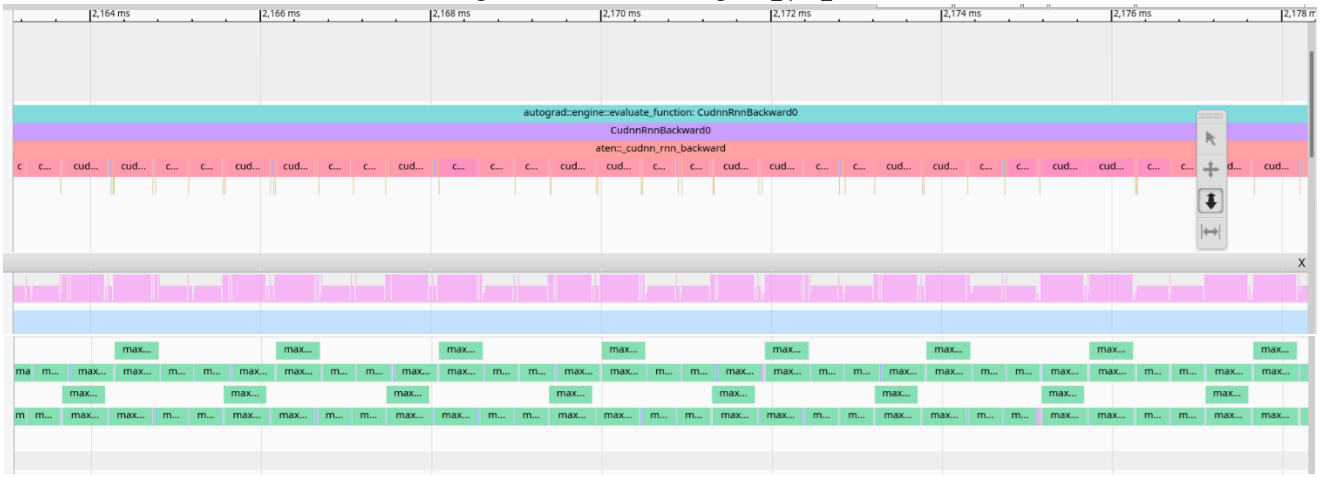


Figure 4: Visualization trace for sgemm_fp16_128x32 kernel calls (indicated by green bars)

2.2 Task 2: Role of batch size in training optimization

The maximum batch size supported on P100 for DeepSpeech was found to be **64**. Beyond this, the training fails to launch since the memory reservation required to fit all the samples from the batches is unsuccessful. Also, as seen in the table for BS 64, we have ~8GB peak memory usage. For 128, this would roughly be double this value that does not fit in the 16GB VRAM device. The trends on varying batch size are illustrated in the table below:

- We observed that the SGEMM kernels generally continue to dominate but for lower batch sizes, the matrix to vector multiplication (*gemm2N_kernel*) that dominate.
- We also note that the average step time does not linearly increase with batch sizes that indicates exploiting more parallelism by trading latency for throughput.
- Batch size improves achieved occupancy, but the correlation between batch size and GPU utilization is murky. This could be due to (i) speech-to-text training data set contains varying lengths of utterances so it may be harder to find/group samples with same length (ii) all batch-size runs had the same learning rate in the implementation, whereas previous work suggests an inverse root relationship is necessary between scaling batch size and learning rate (iii) Larger batch sizes may also cause resource contention at L2 hurting utilization.

	A	B	C	D	E
1	**Batch Size**	**Achieved Occupancy**	**GPU utilization**	**Average step time (ms)**	**Peak memory usage (MB)**
2	1	10.25%	74.97%	676	1048
3	2	5.19%	91.25%	1820	1189
4	4	5.53%	92.37%	2247	1430
5	8	7.12%	91.08%	2406	1894
6	16	9.34%	89.79%	2902	2975
7	32	13.75%	86.30%	3668	5028
8	64	23.35%	82.79%	5671	**8984**

2.3 Task 3: Role of precision in training optimization

Changing precision seemed to have had little effect on execution other than a very minor difference (~5%) in total runtime. We studied the trends in GPU utilization, memory usage, and execution time across three models where the "precision" configuration parameter was set to 16, 32, and 64, but unfortunately noted no major differences, any variations were probably due to noise. We believe that there could possibly be two reasons for this:

1. It may have been a bug in the implementation of the DeepSpeech model used by us – since we used an unofficial PyTorch implementation, mixed/lower precision computation may not have been supported. Added to the fact that the model had recently undergone some major revisions to bring it up to date with more recent versions of PyTorch, this is likely the cause, the authors may have missed some functionality, because the project description mentions half precision support (and mentions a speed memory benefit from using half/mixed precision.). This was also confirmed by the fact that the NVProf/PyTorchProfiler kernel call list continued to present invocations of only the "*sgemm_**fp16***" kernel even when using a single (32) or double (64) precision.
2. The GPU used by us, GP100, introduced low precision floating point computation as per Nvidia. However, it does not possess tensor cores, which are now ubiquitous when exploiting low/mixed precision computing, and this might have resulted in some mismatch between the driver and library API.

2.4 Task 4: Understanding Forward and Backward passes

Forward vs Backward: The ratio of time spent performing forward pass to backward pass was about 3:5. We noticed that forward pass had two distinct phases in the profiler trace: there was an initial phase where data was copied from host to device and processed. In this phase, GPU utilization was low, at about 35% and SM Efficiency was 0 for large periods. This was followed by actual forward propagation through the neural network. We saw extremely high GPU utilization, at 95-100% and an SM Efficiency consistently around 1 for this phase. This phase was dominated by sgemm128*32 calls, followed by gemm32*128 calls.

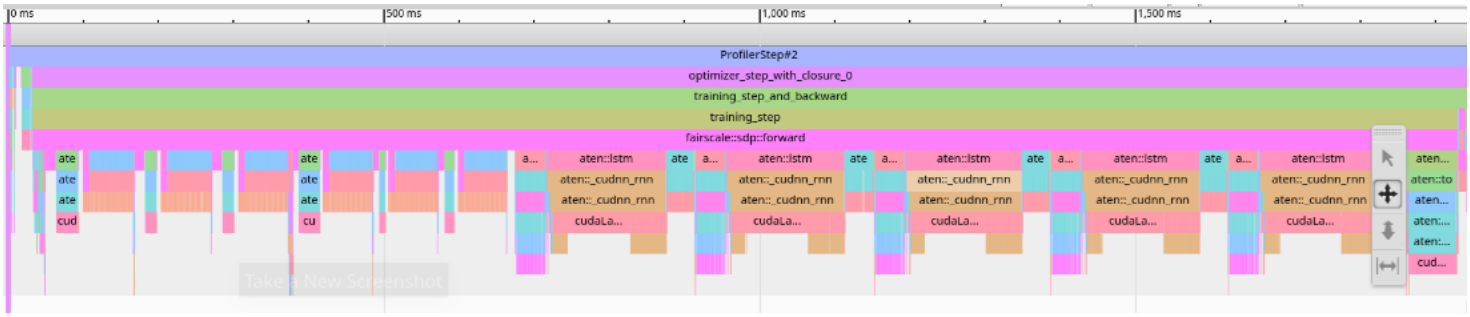


Figure 5: Visualization trace of forward pass operations

In the backward pass, we consistently see high GPU utilization of about 90%. We similarly saw a series of sgemm calls, however, these calls were not called as frequently as in the forward pass. Thus, the SM Efficiency was lower for the backward pass as compared to the forward pass. We found that GPU Utilization statistics were quite coarse grained, while the SM Efficiency seemed to be updated with each function call. The backward pass repeatedly invoked autograd, which called 3 functions sequentially; it first called cuDNN's RNN backward function, followed by a brief function call to pack a padded sequence, followed a CopySlices call. The cuDNN backward pass would begin with a D-to-D memcpy, following which sgemm kernels were called. The CopySlices function also initially involved a cudaMemcpyAsync. In both cases, SM Efficiency was lower on average during the memcpy operations, averaging 40-50%.

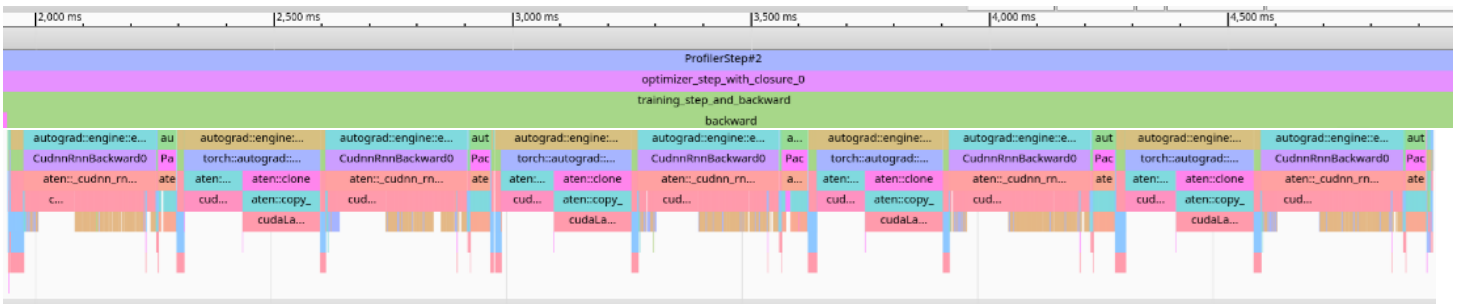


Figure 6: Visualization trace of backward pass operations

Action	Mean duration (s)	Num calls	Total time (s)	Percentage %
Total	-	_	71.67	100 %
run_training_epoch	54.829	1	54.829	76.502
run_training_batch	5.0794	10	50.794	70.872
optimizer_step_with_closure_0	5.0782	10	50.782	70.856
training_step_and_backward	4.3529	10	43.529	60.735
backward	2.9386	10	29.386	41.002
model_forward	1.4136	10	14.136	19.724
training_step	1.4133	10	14.133	19.72
evaluation_step_and_end	4.6363	2	9.2726	12.938
validation_step	4.636	2	9.272	12.937

Figure 7: Forward + Backward distribution of runtime using SimpleProfiler

3 References

[1] Deep Speech: Scaling up end-to-end speech recognition https://arxiv.org/abs/1412.5567

[2] Deep Speech 2: End-to-End Speech Recognition in English and Mandarin https://arxiv.org/pdf/1512.02595v1.pdf

[3] Open Speech and Language Resources: LibriSpeech ASR corpus https://www.openslr.org/12

[4] Speech Recognition using DeepSpeech2. https://github.com/SeanNaren/deepspeech.pytorch

[5] Nvidia Tesla P100 Whitepaper https://tinyurl.com/4chfrmhs

[6] PyTorch Profiling https://pytorch-lightning.readthedocs.io/en/stable/advanced/profiler.html

[7] Achieved Occupancy https://tinyurl.com/2p8c2dv8

[8] Git repository of experiments https://github.com/rajesh-s/deepspeech-mlsys-gpu-profiling

[9] PyTorchProfiler https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html

4 Contributions

Ashwin Poduval

- Bring up of TensorFlow implementation of DeepSpeech
- Profiling and data collection: cProfile, PyTorchProfiler

Rajesh Shashi Kumar

- Bring up of PyTorch implementation of DeepSpeech
- Profiling and data collection: NVProf, SimpleProfiler, PyTorchProfiler (memory trace)

Shared contributions

- Task-wise analysis of profiled results and documentation