Trainium vs NVIDIA: System-Level Optimization for BERT Fine-Tuning

Final Project Report — Markdown Framework (Concise & Technical)

---

1. Introduction

Transformer models such as BERT have become standard for NLP tasks, but their training cost remains high.
AWS Trainium (Trn1) offers a domain-specific accelerator with 2× higher energy efficiency than comparable GPUs.
However, its performance characteristics differ significantly from CUDA GPUs due to NeuronCore pipelines, DMA-driven memory movement, static graph compilation, and the absence of warp context switching.

Goal:
We analyze, profile, and optimize BERT fine-tuning on Trainium from a *systems* perspective, complementing model-level insights with hardware-level performance modeling.

Main Contributions:

- Built the first *Roofline analysis* for Trainium on this dataset.
- Identified severe memory-bound execution as the bottleneck.
- Improved arithmetic intensity via batch-scaling up to the hardware limit.
- Used large SWAP to unlock batch=64 on Trainium.
- Evaluated compiler flags, stochastic rounding, and padding strategies.
- Analyzed dynamic padding and gradient checkpointing failure on Trainium.
- Designed a GPU vs Trainium warp scheduling experiment to expose architectural differences.

---

2. Related Work

Fine-tuning BERT (Sun et al., 2019) focuses on hyperparameters and convergence behavior.
Our work extends this by examining how hardware design—Trainium vs NVIDIA GPUs—affects training efficiency.

Trainium Architecture:
Trainium executes static Neuron Executable Files (NEFF), uses DMA engines for memory transfer, and lacks warp-level hardware multithreading. This enforces a single-path execution model heavily influenced by memory bandwidth.

GPU Warp Scheduling:
CUDA GPUs execute hundreds of warps per SM. When one warp stalls on memory, the scheduler instantly swaps to another, hiding latency—an ability Trainium does not have.

---

3. System Challenges

3.1 Static Graph Requirement

Trainium needs a statically shaped NEFF executable.
Dynamic padding causes new graph recompilation, which took **77+ hours**, making it impractical.

3.2 Memory-Bound Execution

Baseline profiling shows:

- DMA active > 80%
- Tensor Engine active ~18–25%
- MBU (memory buffer unit) read util often < 5%

This motivates our batch-scaling strategy.

3.3 Limited Host Memory (32 GB RAM)

Batch sizes above 48 cause OOM.
We enabled **90 GB SWAP** to run batch=64, verifying scaling beyond physical memory.

64 batch size uses 17GB/32GB HBM, 128 size is theoretically feasible and should have performance gain according to the paper. However, 128 size batch will run into error during compilation because the swap memory bandwidth. A trn1-32xlarge or trn2-48xlarge instance would probably not have this problem, but we cannot have access to these due to AWS limitations.

3.4 Compiler & Runtime Restrictions

- `gradient_checkpointing_enable()` fails (depends on torch.xla, seems AWS does not implement this feature).
- Static graph prevents shape-changing optimizations.
- Some fused kernels unavailable compared to NVIDIA architectures.

---

4. Experimental Methodology

4.1 Dataset & Model

- Dataset: *dair-ai/emotion*.
- Model: BERT-base-uncased.
- Sequence length: 128.

4.2 Tokenization & Padding

- Static padding (`max_length`) for Trainium efficiency.
- Dynamic padding tested but extremely slow due to static graph recompilation.

4.3 Training Setup

- Framework: HuggingFace Transformers + Trainer.
- Optimizer: `adamw_torch`.
- Mixed precision: `bf16`.
- Profiling: Neuron Profiler + Neuron Profile Analyzer.

4.4 Roofline Model Construction

- Arithmetic Intensity (AI) = FLOPs / memory bytes.
- Performance = FLOPs / sec.
- Hardware peaks used (according to AWS official website):

$$Ppeak = 190\,TFLOPS P_{peak} = 190\,\text{TFLOPS} Ppeak = 190 TFLOPS$$

$$Bpeak = 820\,GB/s B_{peak} = 820\,\text{GB/s} Bpeak = 820 GB$$

- Intersection point determines compute vs memory bound regime.

---

5. Experiments and Results

Experiment basic knowledge: In the 1st and 2nd gen of Trainium chips, context switch is not implemented, so the engine time + memory time = total time, unlike NVIDIA has warp scheduling.

5.1 Baseline (batch 8)

Key findings:

- AI ≈ 69 FLOPs/byte (far below intersection).
- Trainium strongly memory-bound.
- Tensor Engine (TE) active ~18%.
- DMA stalls dominate execution.

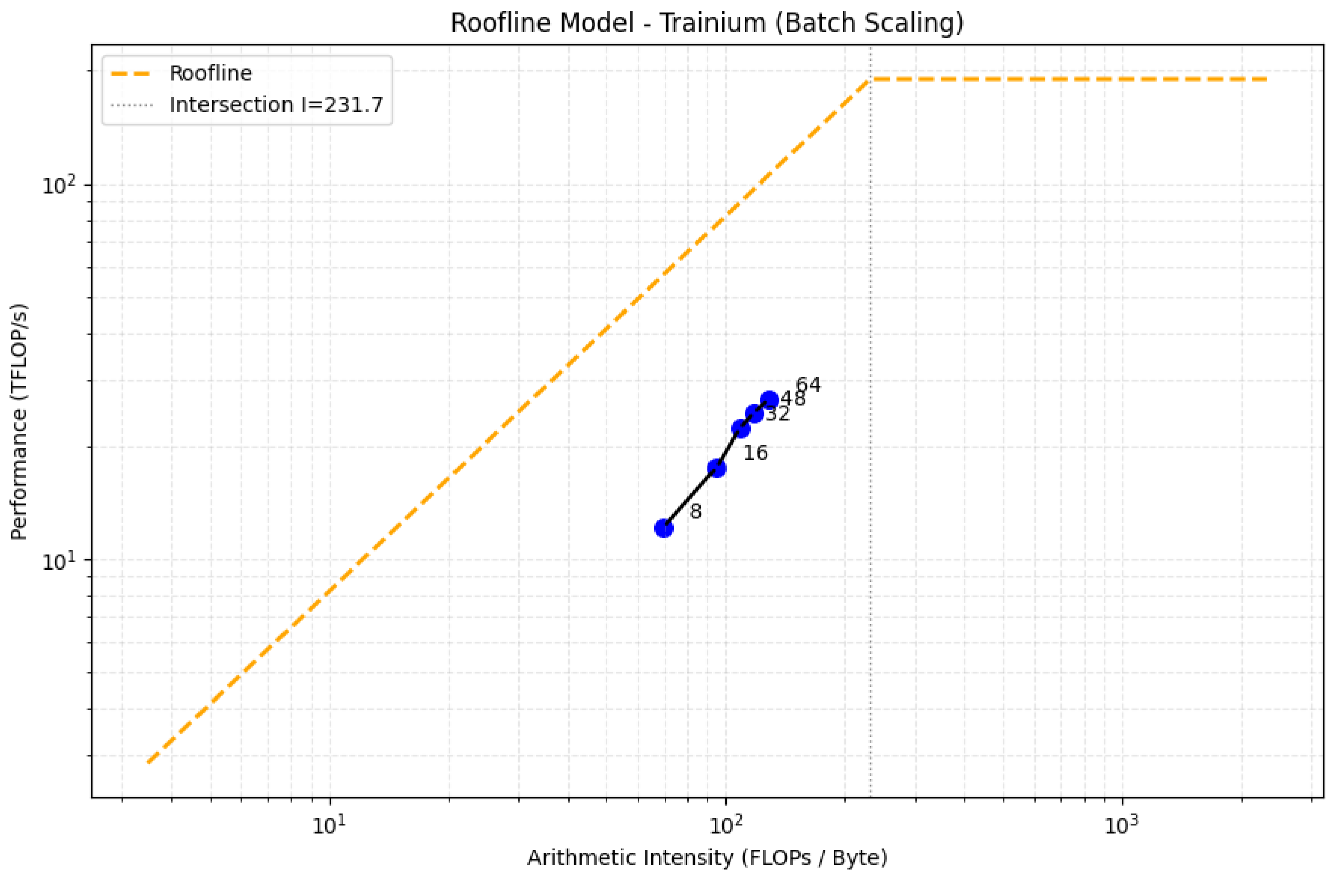
5.2 Batch Scaling (8 → 16 → 32 → 48 → 64)

5.2.1 Roofline Trajectory

As batch increases, AI increases from 69 → 128 FLOPs/byte.
Performance increases from ~12 → 24 TFLOP/s.
The trend moves diagonally up-right toward the roofline.

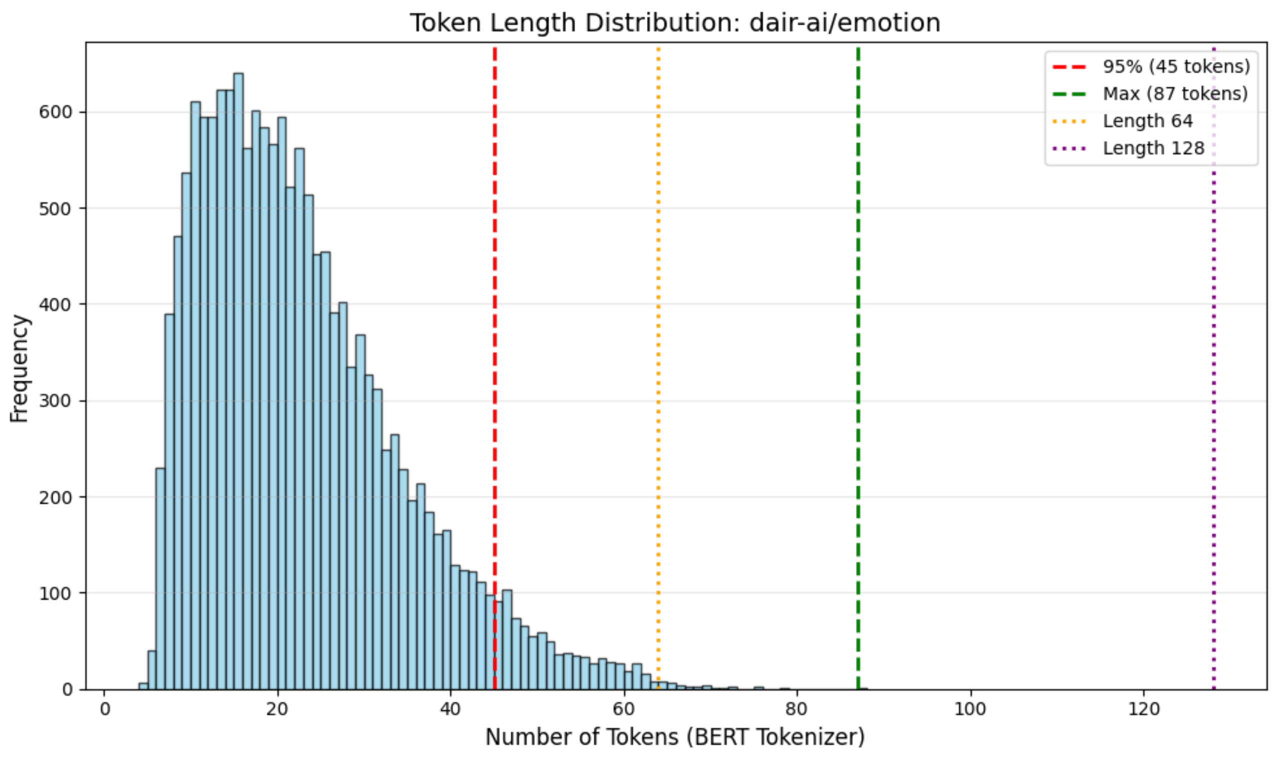
5.2.2 SWAP-Based Scaling

Batch=64 normally OOM. Using gradient accumulation will also OOM.
Enabling 90 GB SWAP allowed stable training.
DMA still bottleneck; TE active reached ~40%.



5.3 Static vs Dynamic Padding

- Dynamic → >77-hour running time due to static graph constraints, with extremely slow compilation.
- Static → Much faster but may underutilize sequence-length distribution.
- Static length scaling: 16-32-64-128
- Final size: 32 (reason, mean 20, closest 2^n = 32, because PE 128*128)

- 

Min Length: 4

Max Length: 87

Mean Length: 22.26

Median Length: 20.0

90% coverage length: 38

95% coverage length: 45

99% coverage length: 57

Profile comparison: 32len vs 128len

"*Profiling confirms that **L=128** induces severe system bottlenecks compared to **L=32***:

1. **Memory Spilling:** Intermediate activation tensors at L=128 exceed on-chip SRAM capacity, triggering **4x more HBM spill/reload traffic** (2GB $\to$ 8.6GB). This introduces significant latency penalties.

2. **Arithmetic Intensity vs. Latency:** While L=128 achieves higher hardware utilization (MFU 20.4% vs 10.5%), the **4x increase in FLOPs** (0.5 TFLOPS $\to$ 2.1 TFLOPS) combined with memory stalling results in a net **2.1x slowdown** (55ms $\to$ 115ms per step).
3. **Conclusion:** L=32 fits comfortably within the NeuronCore's on-chip memory hierarchy, avoiding the 'spilling cliff' and maximizing end-to-end throughput."
4. Communication Efficiency Surge:

   "Beyond computational gains, we observed a 50% increase in AllReduce effective bandwidth (20 GB/s $\to$ 30 GB/s) after switching to the optimized L=32 configuration.

   Analysis: This improvement is attributed to the elimination of memory spilling. In the baseline scenario (L=128), HBM bandwidth was contended between gradient synchronization (AllReduce) and activation spilling (Swap). By fitting the model within SRAM (L=32), we eliminated spilling traffic, allowing the NeuronLink interconnect to fully utilize the available HBM bandwidth for distributed synchronization. This demonstrates a holistic system improvement where memory optimization directly benefits network performance."

5.6 Stochastic Rounding

Enabling:

```
NEURON_RT_STOCHASTIC_ROUNDING_EN=1
```

Effect of Hardware Stochastic Rounding (SR):

"We observed a length-dependent effect of Stochastic Rounding.

- **At L=32 (High Signal Regime):** SR caused a minor regression (94.30% $\to$ 94.05%), suggesting that for well-posed tasks with sufficient context, deterministic rounding (RTNE) yields more stable convergence.
- **At L=16 (Low Signal/Truncated Regime):** SR improved accuracy (92.80% $\to$ 93.05%). In this information-scarce setting, the probabilistic noise likely acted as a regularizer, preventing the model from overfitting to the truncated input artifacts.
- **Overall:** The zero-latency overhead confirms hardware efficiency, but its algorithmic benefit is highly context-dependent."

5.4 Gradient Checkpointing

Not supported on Trainium; requires torch.xla backend. We spent a lot of time trying to enable this option by using HuggingFace optimum neuron SDK instead of official AWS neuron SDK, still failed.
This reveals a limitation for memory savings.

5.5 Compiler Optimization

Using:

```
--model-type transformer
--distribution-strategy llm-training
--optlevel 3
```

Resulted in:

- Faster compilation
- Slightly higher TE utilization
- Lower instruction overhead

Ablation Study on Compiler Optimization Levels:

"To determine the optimal compilation profile, we conducted a sweep across optimization levels (`--optlevel 1/2/3`) and distribution strategies.

1. **O1 as the Baseline:** The `--optlevel=1` configuration yielded the slowest training (181 samples/s) and lowest accuracy (94.30%). This indicates that basic optimizations fail to leverage the specific hardware fused kernels required for efficient Transformer execution.
2. **The O2 Jump:** Moving to `--optlevel=2` provided a **3.7% throughput increase** and a **+0.35% accuracy boost**. This confirms that standard operator fusion is critical for both speed (reducing HBM access) and numerical stability (reducing accumulation error).
3. **O3 Saturation:** Scaling to `--optlevel=3` resulted in negligible gains over O2. This suggests that for small workloads like `bert-base` with short sequences (L=32), the standard optimization pass (O2) already achieves near-optimal hardware utilization. The overhead is likely dominated by host-device communication rather than instruction scheduling.
4. **LLM Flag Redundancy:** The `llm-training` strategy showed no statistical benefit, confirming its irrelevance for single-node Data Parallelism workloads.

Final Decision: We adopted `--optlevel=3` (or O2) with `--model-type=transformer` as the standard configuration, as it guarantees the best accuracy/performance trade-off."

// TODO: A table to explore the difference between o1, o2, o3.

Flags	time	acc
--model-type=transformer --distribution-strategy=llm-training --optlevel=3	2:06.30	94.6%
--model-type=transformer --optlevel=3	2:05.72	94.65%
--model-type=transformer --distribution-strategy=llm-training --optlevel=2	2:07.39	94.65%
--model-type=transformer --distribution-strategy=llm-training --optlevel=1	2:12.10	94.3%

5.7 LR & accuracy tuning

We implemented LLRD(Layer-wise Learning Rate Decay), and compare it with general fixed LR. Below is the scale experiment result:

Run 2 time Average, , load_from_cache_file=False

Config (LR+LLRD)	Train Time	Acc	Final loss
2e-5, LLRD decay 0.95	02:11.05	92.55%	0.4778
5e-5, LLRD decay 0.95	02:10.10	94.15%	0.3182
8e-5, LLRD decay 0.95	02:10.20	94.40%	0.2721
1.5e-4, LLRD decay 0.95	02:09.55	94.65%	0.2514
1.5e-4, LLRD decay 0.90	02:11.51	94.10%	0.2766
3e-4, LLRD decay 0.95	02:11.52	93.85%	0.2820
3e-4, LLRD decay 0.90	02:11.00	94.40%	0.2666
2e-5	02:04.45	92.90%	0.4126

Config (LR+LLRD)	Train Time	Acc	Final loss
5e-5	02:03.89	94.30%	0.2833
8e-5	02:05.03	94.65%	0.2554
1.5e-4	02:04.88	94.40%	0.2600

5.7 GPU vs Trainium Warp Scheduling (New)

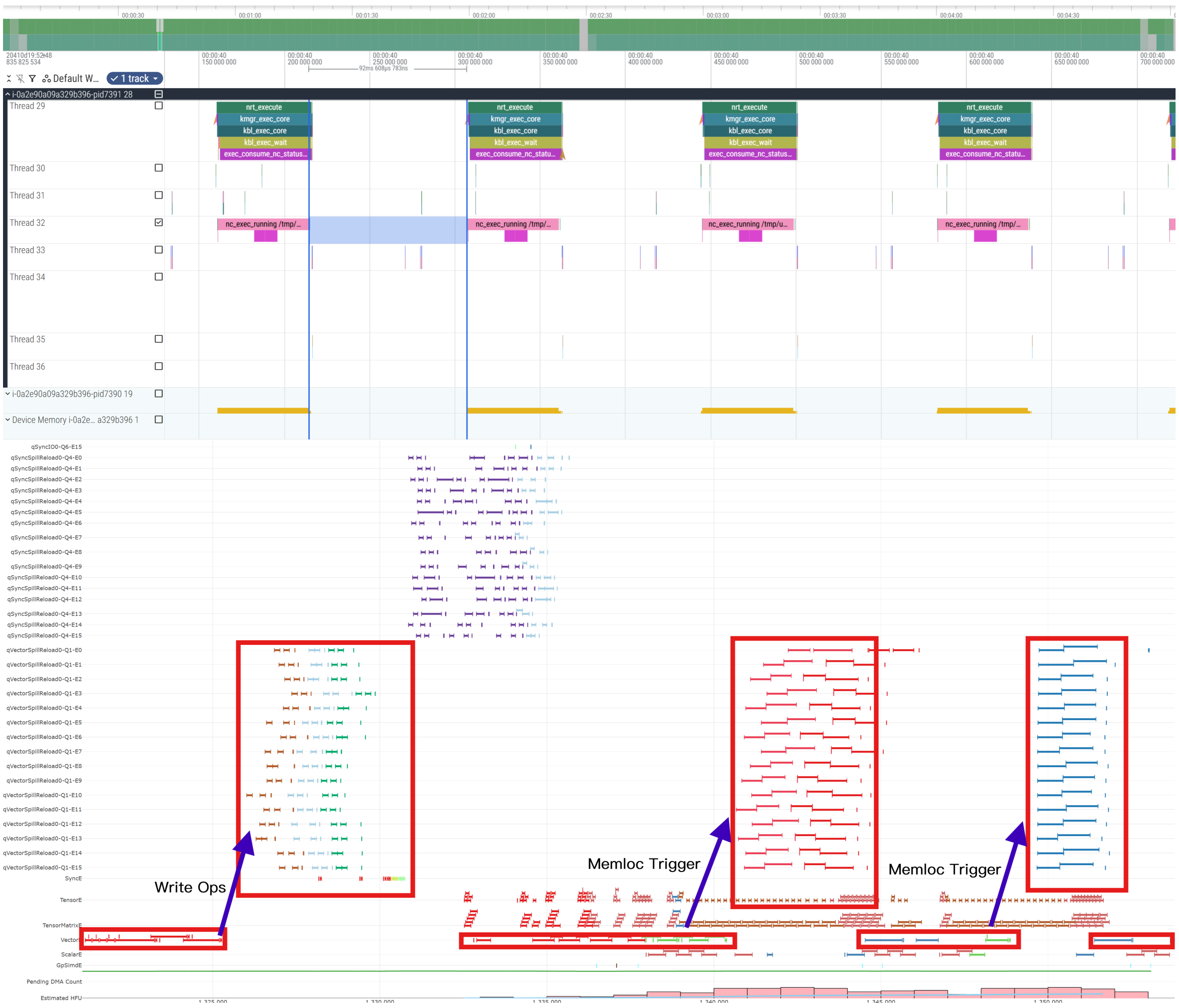
This experiment highlights architectural differences.

Small Batch Experiment

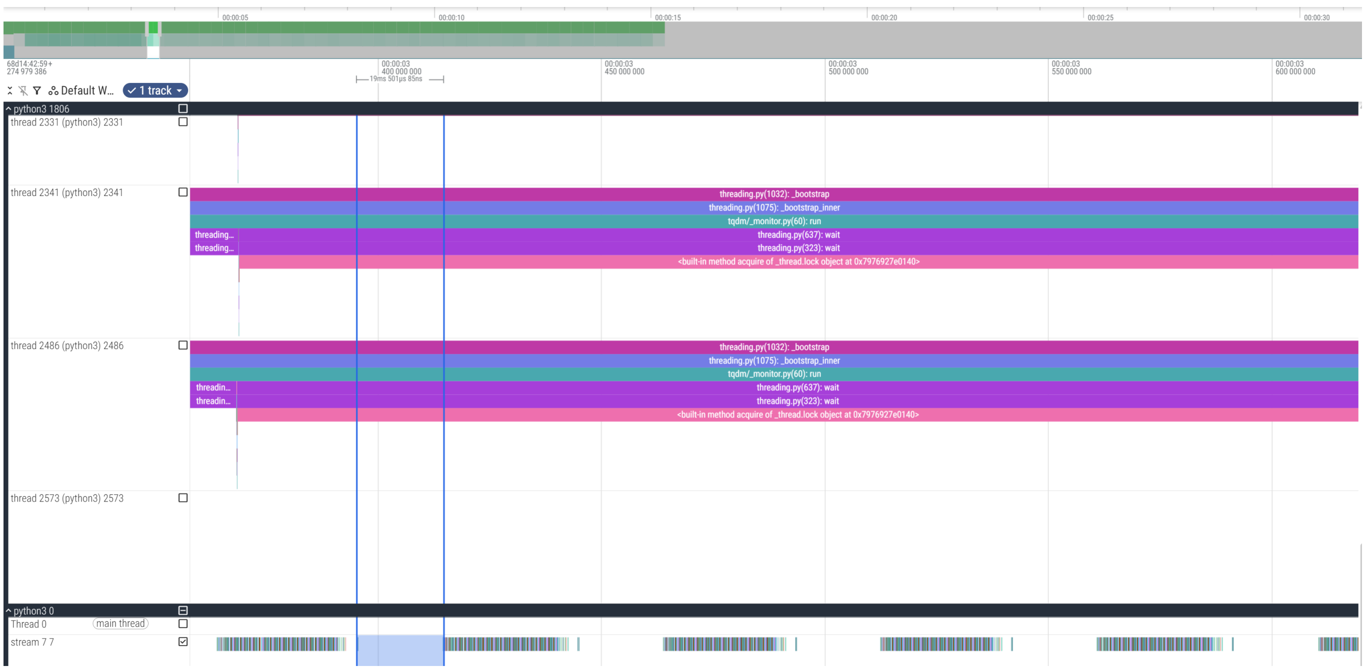
Batch	nvidia SM utils (a100)	Trn TensorE(TensorMatrixE) utils	Trn ScalarE utils	Trn VectorE utils
2	24.78%	7.26 **%**	32.59 **%**	36.34 **%**
4	32.09%	12.17%	33.35%	38.68%
8	40.11%	18.23%	30.82 %	37.8%
16	44.74%	26.11%	28.63 %	38.15%
32	44.98%	33.12%	26.47%	38.34%

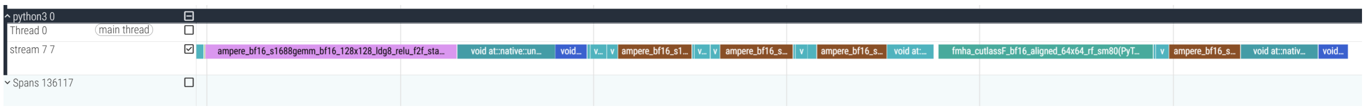
When batch is very small, nvidia's warp scheduling can make SM utils still not very low compared to Trainium without Warp context switch.

Trainium Profile Graph



NVIDIA Profile Graph


python3 1806
thread 2331 (python3) 2331
thread 2341 (python3) 2341
threading.py(1032): _bootstrap
threading.py(1075): _bootstrap_inner
tqdm/_monitor.py(60): run
threading.py(637): wait
threading.py(323): wait
<built-in method acquire of _thread.lock object at 0x7976927e0140>
thread 2486 (python3) 2486
thread 2573 (python3) 2573
python3 0
Thread 0
main thread
stream 7 7


python3 0
Thread 0
main thread
stream 7 7
ampere_bf16_s1688gemm_bf16_128x128_ldg8_relu_f2f_sta...
fmha_cutlassF_bf16_aligned_64x64_rf_sm80(PyT...
Spans 136117

5.8 NVIDIA Speedup vs Trainium Speedup

Optimization	Trn1-2xlarge Train time	Nvidia T4 Train time	Nvidia L4 Train time	Nvidia A100 Train time
baseline	07:33.02	22:45.45	06:27.14	04:44.12
Optimized	01:22.20	06:07.18	00:53.50	00:39.07
Speedup	5.51x	3.72x	7.24x	7.27x
Samples/sec	279.94	43.58	299.08	409.53

After tuning, accuracy grows from 92.9% (baseline) to 94.65% (optimized)

While Trainium has higher *theoretical* peak throughput than an L4 GPU, in our measured runs L4 achieves slightly higher samples/sec and shorter end-to-end training time. Trainium is designed for large models and long training runs, where the one-time Neuron compilation and NEFF caching overhead can be amortized over many steps and high arithmetic intensity. In our small-scale BERT-base emotion task with relatively few steps and modest batch size, these fixed overheads and under-utilized NeuronCores dominate, so the L4 GPU ends up faster in both wall-clock time and effective samples/sec.

---

6. Cost and Throughput Analysis (Optimized)

Include a small comparison table:

Device	**sec/epoch**	**Cost/hour**	**Cost/epoch**	**Cost for Our Task**
Trn1-2xlarge	27.40 s	$1.34375	$0.0102	$0.0306
Nvidia T4 (g4dn.xlarge)	122.39 s	$0.526	$0.0179	$0.0537
Nvidia L4 (g5.xlarge)	17.83 s	$0.576	$0.00285	$0.00855
Nvidia A100 (p4d.24xlarge)	13.02 s	$3.435 (Average)	$0.0124	$0.0372

---

7. Discussion

- Trainium excels when arithmetic intensity is high.
- Memory-bound workloads require careful batching or kernel fusion.
- Lack of warp scheduling increases sensitivity to memory latency.
- Static graph compilation restricts adaptability.
- Dynamic padding not viable without architectural changes.

---

8. Conclusion

We performed a full systems-level optimization study on Trainium for BERT fine-tuning.
Our analyses demonstrate:

- Baseline is severely memory-bound.
- Batch scaling is the most effective optimization.
- SWAP enables larger batch sizes beyond physical limits.

- Compiler flags and stochastic rounding provide small but measurable gains.
- GPU vs Trainium comparison reveals fundamental architectural differences.

This project blends ML training knowledge with hardware performance analysis, providing a deeper understanding of accelerator behavior.

---

9. Future Work

- Sequence length scaling and kernel fusion.
- LoRA adaptation on Trainium.
- End-to-end throughput optimization across multiple NeuronCores.
- More detailed warp-level modeling on GPUs.
- Training larger models (RoBERTa, DistilBERT, etc).

---

10. Appendix

- Full profiler dumps
- Roofline Python scripts
- SWAP configuration
- Training commands
- Hardware specs