APAC-HPC-AI-2024 GPU Task

Workload Profile

- **Workload**: Llama-2-7b finetune-full
- **Max Sequence Length**: 512
- **Number of Epochs**: 1
- **Dataset**

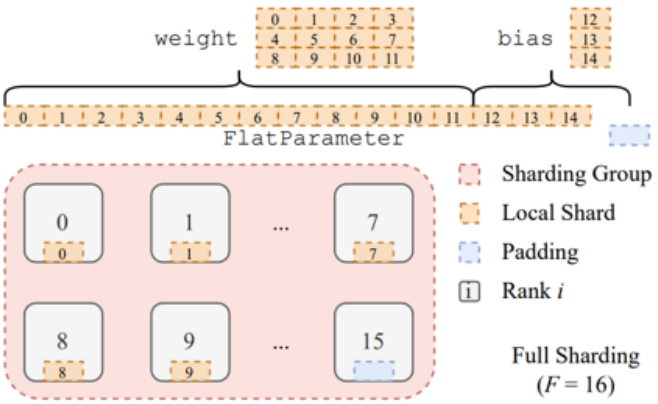
Supercomputer	NCI Gadi iterations	Gadi Baseline	NSCC SG Aspire-2A iterations	Aspire-2A Baseline
Dataset	`Alpaca1024/train.json`	421.66s / 422.16s	`Alpaca1024/train.json`	41.42s / 41.51s

Litgpt Llama2-7B Full-param Finetune

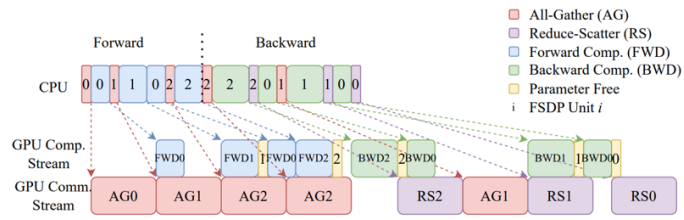
LitGPT offers high efficiency, simplicity, and extensibility, especially for deploying and optimizing LLMs. It uses Pytorch Fully Sharded Data Parallel(**FSDP**) which is particularly suited for full-parameter finetuning, as it shards model parameters across GPUs, significantly reducing memory usage and improving multi-GPU training efficiency. LitGPT's from-scratch model implementations and lack of abstraction layers allow deep debugging and control, ideal for highly customized use cases.

FSDP

PyTorch FSDP is a powerful distributed training strategy designed for large-scale model training. FSDP achieves efficient parallelism by dividing and distributing model parameters across multiple GPUs or nodes, significantly reducing memory usage and improving computation speed. The core technique involves sharding the model's weights across devices so that only the relevant shard is loaded into memory at any given time. This method optimizes both memory consumption and communication costs.



Sharding Params Opt Grade: Splits model weights and biases into multiple shards, each assigned to different GPUs to reduce memory overhead.

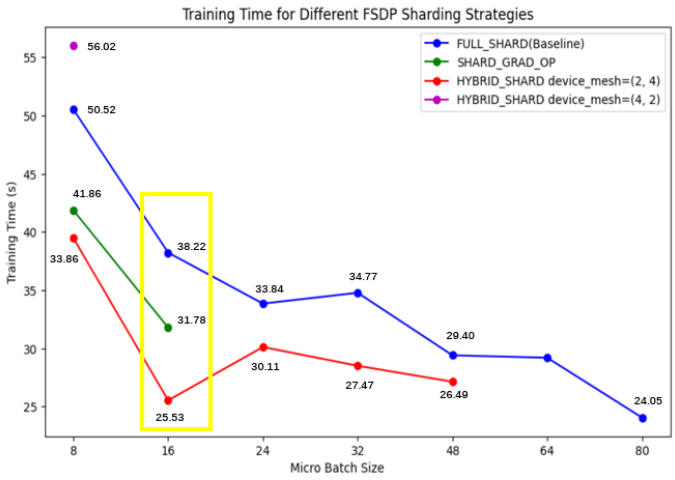


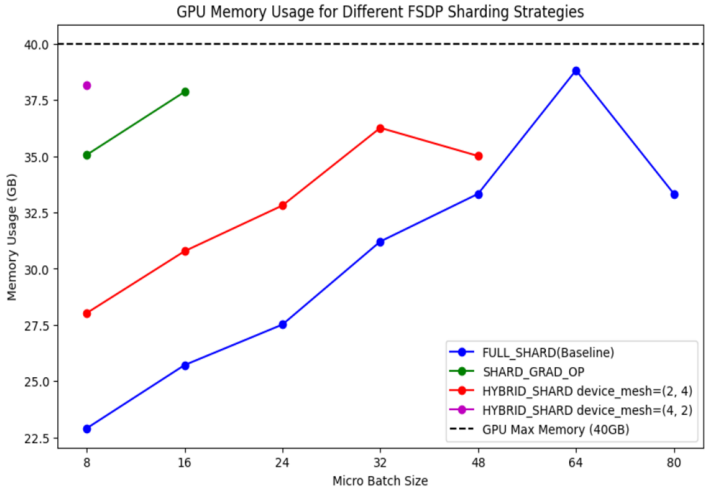
Overlap Communication and Computation: Utilizes overlapping of All-Gather, Reduce-Scatter, forward, and backward computations to enhance efficiency.

In FSDP training, there are several sharding strategies to balance memory utilization and communication cost:

- **FULL_SHARD** -> Shards parameters, gradients, optimizer states
- **SHARD_GRAD_OP** -> Shards gradients, optimizer states
- **NO_SHARD** -> No sharding, similar to regular DDP
- **HYBRID_SHARD** -> Shards within node, replicates across nodes

The FSDP sharding strategies offer various trade-offs between memory usage and communication overhead. **FULL_SHARD** maximizes memory efficiency by fully sharding parameters, gradients, and optimizer states, but incurs high communication costs due to frequent all-gather and reduce-scatter operations. **SHARD_GRAD_OP** reduces some communication by sharding only gradients and optimizer states, while **NO_SHARD** requires the least communication but has the highest memory consumption as all data is fully replicated. **HYBRID_SHARD** combines the benefits of FULL_SHARD within each node and replicates across nodes, significantly reducing inter-node communication and achieving better performance for multi-node setups. This is particularly effective for two-node, eight-GPU configurations, as it confines costly communication operations within nodes, balancing memory usage and reducing communication time.





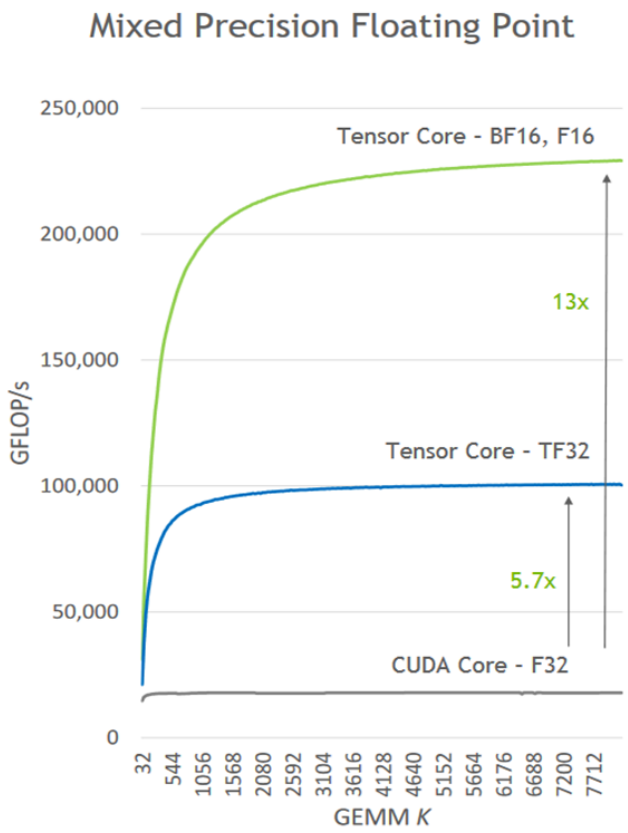
Nvidia A100

The NVIDIA A100 Tensor Core GPU is a high-performance computing GPU designed specifically for AI and scientific computing tasks.

The key feature of the A100 is **Tensor Core Optimization**: The A100 Tensor Cores are optimized for matrix operations (like matrix multiplication), which are foundational in AI model training. This optimization allows for higher throughput in tasks such as large-scale matrix computations.

NVIDIA A100 TENSOR CORE GPU SPECIFICATIONS (SXM4 AND PCIE FORM FACTORS)

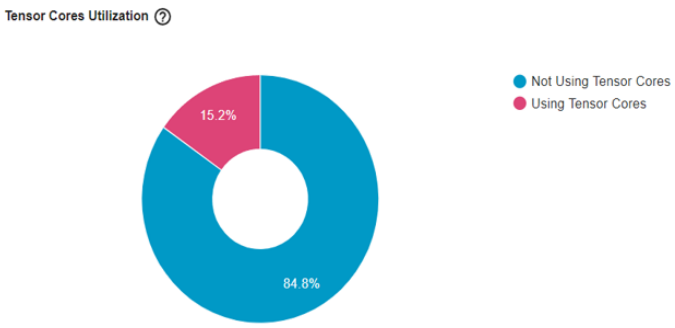
	A100 40GB PCIe	A100 80GB PCIe	A100 40GB SXM	A100 80GB SXM
FP64	9.7 TFLOPS			
FP64 Tensor Core	19.5 TFLOPS			
FP32	19.5 TFLOPS			
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*			
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
INT8 Tensor Core	624 TOPS | 1248 TOPS*			



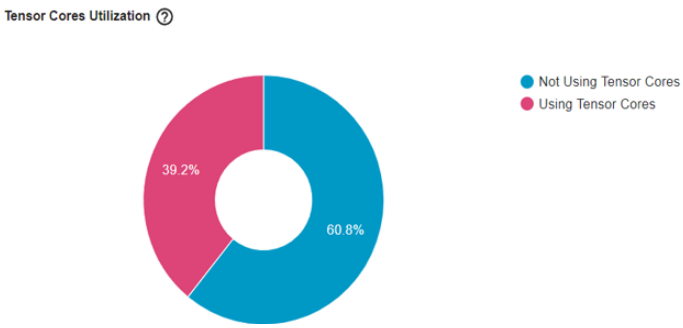
Using BF16 or FP16 mixed precision on Tensor Cores achieves up to a **13x** performance boost in GFLOP/s compared to standard FP32 on CUDA Cores, while TF32 on Tensor Cores provides a 5.7x improvement.

Increasing batch size is an effective way to improve the utilization of Tensor Cores by optimizing memory bandwidth usage. A larger batch size means that more data can be processed simultaneously in each computation step, allowing the Tensor Cores to perform operations on a larger dataset at once. This reduces the need for frequent memory access, thus alleviating the memory bandwidth bottleneck. On high-bandwidth GPUs like the A100, a larger batch size ensures a steady flow of data, enabling Tensor Cores to operate at peak efficiency without waiting on memory transfers. This approach not only speeds up training but also maximizes the computational power of Tensor Cores.

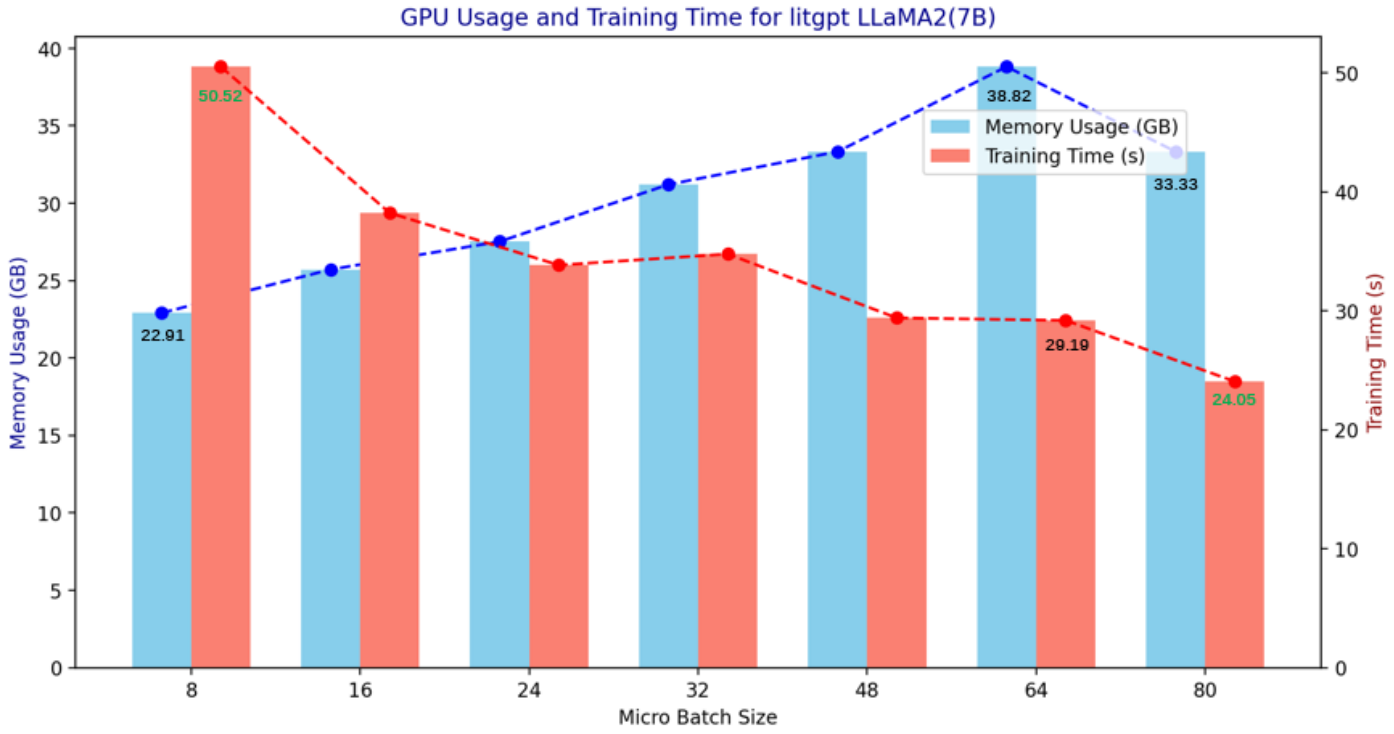
CUTLASS provides us with high-performance GEMM operators by maximizing A100 hardware capabilities, such as *vectorized memory access* and *asynchronous memcpy*. Libraries like FlashAttention further optimize the GEMM utilization and the overlap between computation and memory access. Therefore, we "**just give it enough data to compute**," which can be achieved by increasing the batch size.



micro batch size = 8

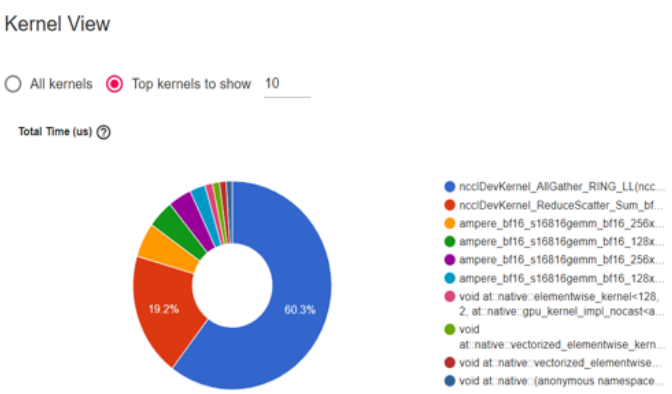


micro batch size = 64

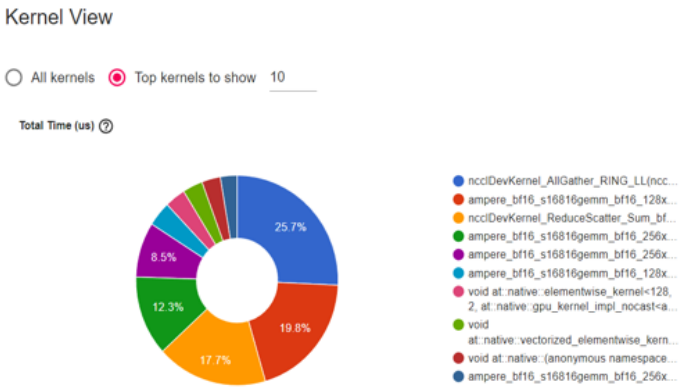


Training Time speedup about **2.10x** for 2 nodes 8 GPUs

Collective Communication



micro batch size = 8



micro batch size = 64

Collective Communication operations occupy a substantial portion of the total computation time, with communication (e.g., All-Gather and Reduce-Scatter) making up about 80% of the time. Additionally, the data size communicated is large, with the backward pass requiring about 3.5 GB and the forward pass needing about 1.75 GB. This suggests that communication overhead is a bottleneck in the training process, consuming a significant amount of time and bandwidth, which impacts overall training efficiency.

Increasing the **batch size** can **reduce the proportion of time spent on communication relative to computation**. With a larger batch size, each computation step processes more data,

which improves GPU utilization and reduces the frequency of communication calls. This shifts the balance towards computation, effectively lowering the communication overhead as a percentage of the total training time, thereby enhancing overall efficiency.

We attempted to use MSCCL, a state-of-the-art distributed backend faster than NCCL, but saw almost no improvement in performance. The analysis revealed several reasons:

1. **PyTorch’ s Default Bucket Size**: The default bucket size in PyTorch is 25MB. MSCCL optimization shows minimal speedup (around 1.0x) for bucket sizes between 16MB and 32MB, indicating that performance improvement is limited in this range.
2. **FSDP Overlap of Computation and Communication**: FSDP already provides overlapping of computation and communication to minimize idle time, which reduces the potential gains from switching to MSCCL.
3. **Communication Efficiency on Two Nodes with Eight GPUs**: Communication efficiency is inherently lower on a two-node, eight-GPU setup compared to a single-node, eight-GPU setup. *The bandwidth difference between intra-node and inter-node communication is an order of magnitude*. Therefore, for optimal training throughput, we recommend using **a single node with eight GPUs**.

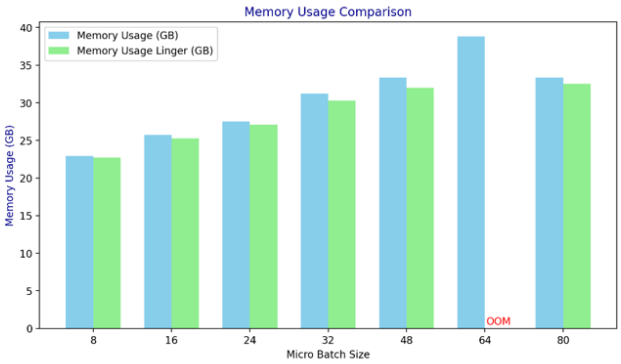
Computational Operators

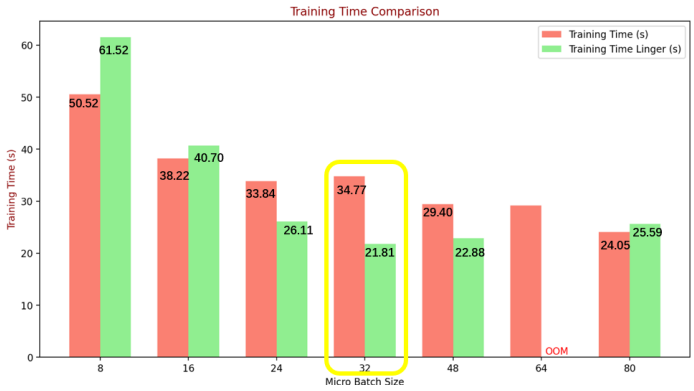
In Transformer models, the core computational operators are limited to five key components:

- Attention
- MLP
- Norm
- Rotary Position Embedding
- Embedding

In our case, **Flash Attention** is already the default in litgpt. We only replaced **RMSNorm** with the **Liger Kernel Implementation**, which is a collection of Triton-implemented kernels.

Unlike other frameworks that use the transformers library, litgpt does not directly rely on transformers and instead has its own blocks for loading weights. To fully integrate all operator replacements, we may need to participate in the development of the litgpt trainer to ensure compatibility with all components.





Although we only replaced the Norm with the Liger Kernel, the performance improvement is still significant. For a micro-batch size of 32, the training time speedup is approximately **1.594x**. Liger Kernel RMSNorm not only reduces training time but also performs well in terms of memory usage for larger batch sizes, enhancing overall resource efficiency.

Conclution

In our LitGPT-based Llama2-7B full-parameter finetuning, we focused on three main optimizations:

1. **Batch Size Optimization** on Dual-node, 8-GPU (A100 80GB): Leveraging the high bandwidth of A100 GPUs, we increased the batch size to maximize Tensor Core utilization. A larger batch size reduced communication frequency, improved GPU utilization, and effectively lowered communication overhead, enhancing overall training efficiency.
2. **FSDP Sharding Strategy Tuning**: By selecting an appropriate FSDP sharding strategy, we balanced memory usage and communication cost. While FULL_SHARD minimizes memory usage, it incurs high communication overhead; HYBRID_SHARD, on the other hand, shards within nodes and replicates across nodes, optimizing multi-node performance, especially in dual-node, 8-GPU setups.
3. **Computation Operator (Norm) Replacement**: We replaced RMSNorm with the Liger Kernel implementation of RMSNorm, based on Triton. This optimization brought significant performance gains. The Liger Kernel RMSNorm not only reduced training time but also improved memory efficiency for larger batch sizes.