LLMQ: Efficient Lower-Precision LLM Training for Consumer GPUs

Erik Schultheis[1], Dan Alistarh[1]
[1]IST Austria
first.last@ist.ac.at

We present LLMQ, an end-to-end CUDA/C++ implementation for medium-sized language-model training, e.g. 3B to 32B parameters, on affordable, commodity GPUs. These devices are characterized by low memory availability and slow communication compared to datacentre-grade GPUs. Consequently, we showcase a range of optimizations that target these bottlenecks, including activation checkpointing, offloading, and copy-engine based collectives. LLMQ is able to train or fine-tune a 7B model on a single 16GB mid-range gaming card, or a 32B model on a workstation equipped with 4 RTX 4090s. Parallel training is achieved while executing a standard 8-bit training pipeline, without additional algorithmic approximations, and maintaining FLOP utilization of around 50%. As such, the efficiency of LLMQ rivals that of production-scale systems on much more expensive cloud-grade GPUs. We also present first results for training on the novel HP ZGX Spark device with a Blackwell-architecture GPU and unified memory. Code is available at `https://github.com/IST-DASLab/llmq`.

1. Introduction

Large language models are extremely popular across many application areas, from code completion to personal assistants. While training generalist models that excel in a wide variety of tasks requires compute resources out of reach for anyone but large companies, many tasks can also be solved with smaller, highly specialized models. However, even such smaller models, e.g. in the 0.5 to 32B parameter range, are typically trained on large datacenter accelerators, which can be undesirable as it requires uploading the training data to a third party, and can have high monetary cost.

The alternative to this is *local* training on *user-owned hardware*, using more affordable gaming or even workstation GPUs. These accelerators can be significantly slower than their datacenter counterparts, but can be very competitive in terms of FLOPs per dollar. From the systems perspective, these devices have two key drawbacks: 1) such GPUs have reduced device memory, and 2) they also have much lower available bandwidth for communication (see Table 4).

Contributions In this paper, we present the design and implementation of LLMQ, and end-to-end C++ framework for LLM (continued) pretraining and fine-tuning on commodity GPUs located in a single server node. The basic version of LLMQ supports efficient bfloat16 training, and thus can be used across all recent GPU families, starting from the NVIDIA Ampere line (e.g. RTX 30xx). However, its key feature is in enabling highly-efficient and accurate FP8 training. For this, it uses dynamic tensor-level scaling, supported on both Ada (RTX 40xx) and Blackwell (RTX 50xx) architectures. Our key technical contribution is the implementation of several strategies to alleviate memory bottlenecks:

1. To limit the cost of activations, selective recomputation can be used, starting from only recomputing the non-matrix-multiplication layers, all the way to recomputing full transformer blocks.
2. Further peak memory reductions can be achieved by offloading the remaining residual to CPU RAM. Similarly, we show that optimizer states can be offloaded efficiently.

Second Conference on Parsimony and Learning (CPAL 2025).

3. In a multi-GPU setup, LLMQ always shards optimizer states (ZeRO-1), and allows for sharding of model weights and gradients, independently.
4. As recent generation gaming GPUs are unable to communicate directly with each other over PCIe, we found that offloading sharded parameters fully to the CPU does not increase the communication required during forward/backward passes while reducing GPU memory usage further.

By combining these optimizations, LLMQ achieves high throughput and utilization across consumer hardware configurations. On a workstation equipped with four RTX 4090s, the system attains a throughput of 7,800 tokens/second for a 14B parameter model, corresponding to 54% Model FLOPs Utilization (MFU), leveraging LLMQ's custom communication backend. The framework successfully scales to a 32B parameter model on the same hardware, achieving 3,400 tokens/second at 51% MFU , an efficiency rate significantly higher than that observed on professional L40S GPUs (29% MFU). On a single RTX 4090, LLMQ can train models up to 14B parameters , while a 7B model runs at 4,300 tokens/second with 61% MFU. Even on a constrained 16GB RTX 5060Ti, the system enables 7B model pretraining with a remarkable 70% MFU. We also present results on the recent NVIDIA DGX Spark hardware.

2. Background

A number of system improvements have been devised to mitigate the large memory costs of LLM training. Four main strategies are used: Sharding, recomputation, offloading and quantization.

Sharding, that is, distributing large tensors across multiple devices, is the key optimization in the well-established ZeRO [1] optimization, in which optimizer states, and potentially also gradients and weights, are spread out evenly across the devices. While sharding of optimizer states can be done for free, gradient sharding requires additional communication if used together with gradient accumulation, and weight sharding always introduces additional transfers. Instead of having each worker process every layer of the network, it is also possible to distribute network layers across workers, in a *pipeline-parallel* setup [2]. Such a setup trivially avoids having to communicate model weights or gradients, but instead has to employ sophisticated scheduling to prevent idle workers due to pipeline bubbles [3]. For very long sequences, it might even be necessary to split the sequences across nodes in *context parallism* [4]. For large scale trainings, these different forms of paralellism are combined into multidimensional parallelism [5, 6].

To address the memory consumption of activation memory in particular, it is possible to recalculate a subset of activations during the backward pass instead of keeping them in memory [5, 7]. A famous example of this technique is the avoidance of squared memory cost in attention layers [8].

As a last resort, one can also move large allocations down in the memory hierarchy into slower but more abundant storage, from GPU to CPU memory [9] or even to NVME [10].

Finally, memory can also be saved by switching to lower-precision datatypes. For example, by compressing the moment buffers of Adam [11] to use only 8 bits per element [12]. By using mixed-precision training [13], the activations can be compressed down to 16 bit or even 8 bit [14] floating-point numbers. However, while the reduced dynamic range of 16 bit can be compensated for by either switching to a BF16 representation with more exponent bits [15], or by introducing *loss scaling*, the further reduction to eight bits requires more sophisticated techniques. First, a combination of two different FP8 types, E5M2 and E4M3, with 5 and 4 exponent bits, respectively, can be used [16], depending on whether accuracy or range are more important for a specific tensor. Second, additional scale factors can be introduced that scale values into the representable range before quantization. Such scale factors can be applied at the level of an entire tensor, either using its current values (just-in-time scaling) or based on previous steps' values (delayed scaling). More fine-grained scaling is also possible, giving one scale per token or channel, or even small sub-blocks of the tensor [17]. Scaling could either be handled such to ensure that no overflows occur (abs-max scaling), or accept a small amount of clipping for better fidelity [18].

In addition to memory savings, for low-precision formats that are natively supported in hardware, there are also computational advantages. Smaller data formats mean less data movement, and fewer transistors and energy expenditure required to handle fused multiply-add operations [19]. This gets amplified by specialized hardware units, such as tensor cores [20], which operate on these formats. To benefit, any scaling that gets applied along the inner dimension of the matrix multiplication either needs native hardware support, only available on recent Blackwell GPUs [21], or requires software emulation with overheads [22].

3. Implementation

In this section, we describe the implementation of LLMQ, starting from overall design decisions, to the optimizations necessary for efficiently running larger models on a single and on multiple GPUs.

Overview LLMQ is implemented in C++ and CUDA. For matrix multiplications and attention, we use the available implementations in cuBLAS and cuDNN, respectively; all other kernels are implemented by us, as part of the project, and are provided as open-source. All memory allocations in LLMQ happen at program startup. This means that if the program does not run out of memory before the first step, it will never run out of memory. (In extreme circumstances (<50MiB free), it is possible to get OOM when the GPU runs out of memory to load the kernels during the first step.)

We provide both a pure BF16 implementation, and mixed BF16-FP8 training. In order to support older GPU cards, we use just-in-time tensor-level absmax-scaling for conversion from BF16 to FP8, that is, we first determine the largest absolute value in each tensor, and then rescale so that this is mapped to the largest representable value. Compared to delayed scaling, this requires an additional kernel call (due to the global reduction), but it guarantees that no value will be clipped even if tensor statistics change rapidly. Main transformer matmuls in forward and backward are calculated in FP8, whereas nonlinearities, SDPA, the embeddings and LM-head, as well as gradient accumulation remain in BF16. The latter ensures that many steps of gradient accumulation can be performed without catastrophic cancellation in the summation.

Reproducibility To achieve reproducible (on the same hardware, with the same software) training runs, LLMQ only uses bitwise-deterministic kernels. This is not a problem during the forward pass, but it means that for reductions required in the backward pass, instead of atomic additions, an intermediate buffer to store local result, and a second kernel that handles the global reduction, are needed. Particularly challenging in that regard is the backward pass for the embedding layer: If tokens get distributed "randomly" to different thread blocks, then an intermediate buffer of size `embeddings`×`num-blocks` would be needed. Instead, as implemented in `llm.c` (`https://github.com/karpathy/llm.c`), the token indices are first sorted and partitioned on the CPU, so that each thread block can focus on a small subset of tokens. The sorting process can overlap with the regular backward pass, and thus does not introduce additional latency. In cases where random decisions need to be taken inside GPU kernels (e.g., for stochastic rounding), we use counter-based generators to draw deterministic pseudo-random numbers without requiring an internal state.

Multi-GPU Support Within a single node, there are two main approaches for handling multi-gpu support. One is to use one process per GPU, the other to use multiple threads in a single process. Multiprocessing can be beneficial in avoiding the global interpreter lock in python, which is of no concern to use, and scales beyond a sinlge node. On the other hand, in a multi-threading setup, one can exploit the shared address space which allows direct GPU-to-GPU memcpy without having to resort to IPC handles. While we do support both options, the focus is on the multi-threaded setup and its direct communication options.

When running on multiple GPUs, LLMQ *always* shards optimizer states (aka ZeRO-1). This is strictly better than traditional DDP with replicated optimizer states, as this leads to reduced memory consumption without increasing the amount of communication.

To avoid unnecessary round trips to device memory, we fuse all successive operations that are not either a global reduction or involve a matrix multiplication. In particular, this means that all our

non-linearity operators have an additional output parameter that returns the abs-max of its result. Further, RMS-norm and residual-stream addition are handled in a joint kernel, which then also returns the abs-max of the RMS-norm. As in FP8 on consumer cards, gemm only supports the `TN` (transpose:non-transpose) layout, we need to handle the transposes manually, including a fused transpose+quantize kernel. Finally, we fuse the forward and backward pass of the cross-entropy loss into a single kernel [23, 24], avoiding the need to materialize a huge per-token loss tensor.

3.1. Training on a Single Mid-level commodity GPU: RTX 5060Ti

Next, we will consider the optimizations necessary to enable training a 7B model with only 16GB of device memory. On a single 16GB GPU, the implementation described above allows training $0.5B$ parameter models at batch size 6, but runs out of memory for $1.5B$.

Activation checkpointing To alleviate the memory pressure coming from the activations stored for the backward pass, it is possible to only keep a select subset of activations and recompute the others during backward. If only a moderate amount of memory is needed, it might be sufficient to only recompute the non-gemm values, that is, swiglu and rmsnorm, but for drastic reductions the entire transformer block needs to be recomputed, only keeping the residual of the feed-forward part. As memory capacities vary significnatly between commodity cards, it is not possible to decide the optimal trade-off a-priori; therefore, LLMQ allows for selective recomputation that covers the full range.

In addition to preserving the feed-forward residual, we also always keep small statistics tensors from the forward pass. That means that during recomputation, we do not have to determine the absmax again. In particular, with the absmax known, there is no longer a need for a global reduction, and the quantization can be fused into the nonlinearity.

Reduced-precision optimizer states A second large contributor to memory consumption are the optimizer states. By default, these are kept in fp32 precision, which for AdamW corresponds to 8 bytes per parameter, or 12GB in total for a 1.5B model. This shrinks by a factor of two when switching to BF16 to represent momentum and variance. To ensure unbiased values, conversion of fp32 to bf16 is handled by stochastic rounding. Note that we keep master copies of parameters only in bf16, too.

Offloading As the optimizer states still make up a large fraction of total device memory, further memory cost reduction can be achieved by offloading them to host memory. Here we have two options: Either just keep them in page-locked (pinned) memory and rely on the GPUs zero-copy ability to directly read from host, or allocate smaller buffers on the GPU to do explicit double-buffering. Interestingly, we found that zero-copy gave bad performance (i.e., low utilization of PCIe bandwidth) on gaming GPUs (5060Ti, 4090 but worked well on the more high-end cards (L40s), whereas the situation was reversed for double-buffering. Consequently, we recommend testing both options an the system in question and picking the faster one.

With these improvements, we can run 1.5B with batch-size 2 (no offloading) or 12 (offloading both m and v). To enable the 3B model, we additionally need to offload the 16-bit master copies of the matrix parameters, in which case a batch size of 8 can be achieved.

Finally, we can also offload the last remaining residuals to host memory, which increases the maximum batch size to 10. At this point, activation memory is dominated by the cost of a single layer.

Chunking In particular, it turns out that two tensors grow very large as the batch size increases. The first, unsurprisingly. are the logits, due to the large vocabulary dimension. Somewhat unexpectedly, the second is the workspace needed for cuDNN to enable deterministic backward for flash-attention.

Both of these can be solved by chunking the calculation. For the logits, this means that we split the pre-lmhead embeddings into small chunks, and do a full forward-backward over each chunk, writing the derivative of the embeddings into the corresponding slice and accumulating the gradients of the lm-head weights[24]. For attention, it is even simpler, because there are no trainable param-

eters: Simply iterate over slices of input and output, calling the cuDNN kernel multiple times with a smaller workspace.

By fixing these bottlenecks, we can increase the batch size for the 3B model all the way to 24.

Enabling 7B models on 16GB VRAM The preceding optimizations allow increasing the micro-batch size significantly. This, in turn, means that the amount of computation done in each forward/backward pass increases, and consequently, more communication can be hidden without affecting end-to-end latency.

This gives us the opportunity to offload even the gradient buffers to host memory.[1] Thus, the device memory consumption reduces to that of two layers; one for active computations, and one for double-buffering data transfers. As such, even a 7B model can be run with micro-batch size 16. As this point, however, even a high-end gaming PC will reach its limits of available host memory: A 7B model needs approximately 3×14GB for the optimizer state, another 7GB for the quantized weights, and 5GB for offloaded residuals, a total of 54GB of memory. While it might be possible to offload one more level, from host memory to an NVME [10], doing so will change the speed of transfers by another order of magnitude, no longer hidden behind the computations.

To achieve further total memory reductions, one would need to either switch to even lower bit-width optimizers [12, 25, 26], optimizers with smaller buffers [27–29], or parameter-efficient fine-tuning [30]. While possible and interesting, these optimizations perform algorithmic approximations, and are therefore orthogonal to our work.

3.2. Multi-GPU Training on a 4x RTX 4090 Workstation

In this section, we consider a workstation with 4×RTX 4090 GPUs. This allows for sharding of optimizer states, weights, and gradients, before we need to resort to CPU offloading.

Weight caching on host However, on recent consumer devices, the GPUs cannot communicate directly to each other via PCIe, and instead need to go through the host. As such, placing weights in host memory actually reduces the required communication: They need to be sent from GPU to host during the first forward pass after each optimizer step, but the backward pass, and all subsequent forward passes for gradient accumulation, can use the values cached on the host.

This also means that, contrary to traditional ZeRO [1] levels, one should enable sharded model weights *before* enabling sharded gradients. This is especially true in FP8 training, when model weights are gathered in FP8 but gradients are communicated in BF16.

Imbalances of LM-head and embeddings In particular, due to the large vocabulary dimension, both compute and communication cost for the LM-head exceed that of a regular transformer block. For this reason, we forgo sharding the LM-head/embedding, instead replicating them across each worker. Thus, their gradients need only be synchronized at the last gradient accumulation step. By having the LM-head in a separate buffer than the double-buffered transformer block, we further ensure that during that last backward pass, sending its gradient can be overlapped with computing the gradients for the last two transformer blocks, whose weights are still available locally from the preceding forward pass. Additional overlap can be achieved by scheduling the two backward matrices of the LM-head gradient such that the one that calculates the weight gradient is handled first, and the communication can be overlapped with the input gradient calculation. However, the effectiveness of this optimization diminishes with increased chunking, as sending can only commence in the last chunk.

Unfortunately, for the token embeddings, the next required operation is the global norm reduction of the gradients, so there is no way to hide that communication latency.

cudaMemcpy-based communication When running `all-gather` (fetching model weights) and `reduce-scatter` (accumulating gradients) colletives with `nccl`, we observed that the PCIe link

[1] As explained in the next section, we only offload transformer blocks, not lm-head and embeddings

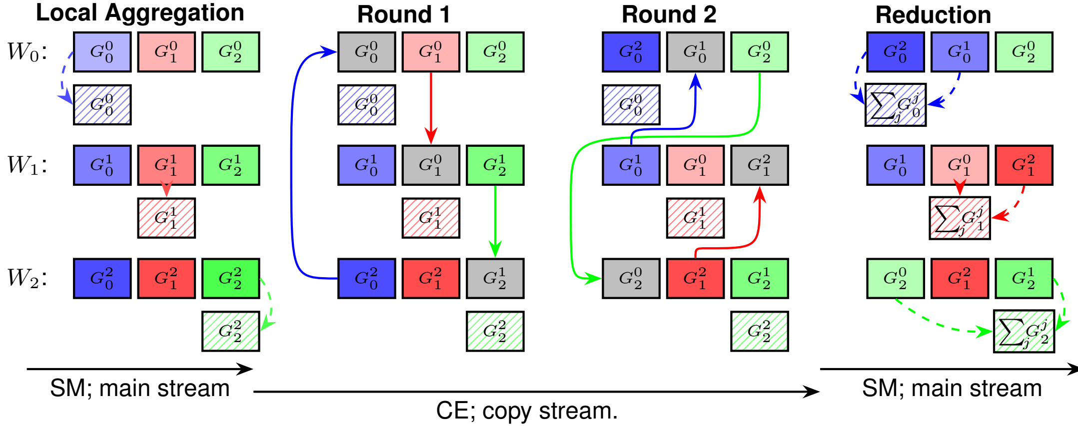

Figure 1: Memcpy-based reduce-scatter. After the local shard has G_i^i of worker W_i has been added to the local accumulator (striped), in each round of communication each worker has exactly one chunk that is not needed anymore, that can be used as a destination buffer for memcpy. At the end, each worker has the corresponding chunks of the other workers, and can accumulate them. The communication phase can be purely handled by the GPU's copy engine (CE), leaving the streaming multiprocessors (SM) available to run the backward pass of the next transformer block.

utilization was quite low. On the other hand, memory transfers handled by the GPUs copy-engines, as initiated by `cudaMemcpy`, do achieve much better utilization, and do not require reserving any SMs for communication kernels.

For `all-gather`, replacing `nccl` kernels with copies is trivial, as gathering only moves bytes around. However, `reduce-scatter` mixes arithmetic and data movement, so it cannot be implemented by copy-engines alone. Therefore, we split the implementation into three phases. First, the local chunk for the current layer is aggregated to the sharded gradient buffer. Then, as this chunk of memory is no longer needed, it can be used as a scratch buffer for receiving chunks from the other workers. This happens in round-robin fashion, that is, after having received chunk and sent one chunk, another part of the memory is unused and can be used as buffer. The copying operations do not need any multiprocessors, and thus can be run perfectly parallel to the backward of the next transformer layer. Only after that transformer layer has finished do we need to synchronize and wait for the transfers to be done. At that stage, the preceding layers gradient buffer contains the gradients for the local shard that have been gathered from the other workers, and we can run the reduction operations, adding them with stochastic rounding. This is illustrated in Figure 1.

Multi-threaded multi-GPU and deadlocks While the multi-threading-based parallelization makes implementing memcpy communication much easier, we found it to have some drawbacks when involving nccl collectives. Specifically, the collective operations involve a global barrier that all GPUs must reach before any can make progress. We observed that, in some cases, this could lead to deadlocks. While we do not know the exact cause, we hypothesize the following: There is some per-process resource that gets filled up as GPU operations are enqueued. If one GPU is faster than another, it could enqueue the collective, and then continue enqueueing further kernels until that resource is exhausted. Because of the barrier in the collective, the GPU cannot execute any more kernels to make space for issuing new operations, and the other GPU cannot issue kernels it needs to reach the collective because there is no space. By placing CPU-side synchronization (that is, the CPU threads are synchronizing among each other, but *not* with the GPU), we can prevent new kernels getting submitted until every worker has issued the collective, which fixes the deadlocks.

Table 1: Training speed and utilization on a single gpu. Sp denotes speed-up of FP8 over BF16.

Size	RTX 5060Ti					RTX 4090					
	FP8		BF16		Sp	FP8		BF16		Sp	LF
	TPS	MFU	TPS	MFU		TPS	MFU	TPS	MFU		TPS
0.5B	16.5k	70%	13.0k	85%	27%	47k	58%	39k	73%	20%	30.4k
1.5B	5.7k	67%	3.9k	78%	46%	20k	69%	14k	81%	42%	9.5k
3B	3.1k	69%	2.0k	80%	55%	10.6k	68%	7.0k	80%	51%	5.4k
7B	1.4k	70%	0.9k	79%	55%	4.3k	61%	2.7k	71%	59%	2.5k
14B	—	—	—	—	—	2.0k	55%	1.3k	59%	54%	1.2k

Table 2: Training speed and utilization on multiple GPUs. Sp denotes speed-up of FP8 over BF16.

Size	4×L40S					4×RTX 4090					
	FP8		BF16		Sp	FP8		BF16		Sp	LF
	TPS	MFU	TPS	MFU		TPS	MFU	TPS	MFU	TPS	
0.5B	191k	27%	185k	37%	3%	181k	55%	154k	71%	18%	123k
1.5B	81k	31%	68k	44%	30%	71k	60%	52k	75%	36%	41k
3B	45k	32%	36k	37%	25%	38k	61%	26k	74%	46%	19k
7B	21k	34%	16k	38%	31%	16.5k	58%	10.9k	69%	51%	6.9k
14B	10k	31%	7.1k	42%	41%	7.8k	54%	5.2k	67%	50%	2.6k
32B	4.2k	29%	3.0k	40%	30%	3.4k	51%	2.2k	65%	54%	OOM

4. Benchmarks

In this section, we provide an overview of the speed achievable in different scenarios. Table 1 shows the tokens per second and model-flops-utilization for differently-sized models, trained at a per-step batch size of 500k, on a single GPU. For each configuration, the combination of offloading/recomputation/micro-batch size that leads to the highest throughput was chosen. Model Flops Utilization (MFU) is calculated taking into account the mixed precision nature of the implementation, that is, we calculate the amount of floating-point operations to be done in each precision, divide by the device's peak rate, and get a lower bound for the achievable duration. The ratio of achievable duration to actual timing is presented in the MFU columns.

Table 3: Training speed and utilization on a DGX Spark. Sp denotes speed-up of FP8 over BF16.

Size	DGX Spark				
	FP8		BF16		Sp
	TPS	MFU	TPS	MFU	
0.5B	17k	28%	17k	44%	0%
1.5B	7.0k	33%	6.5k	53%	8%
3B	4.4k	39%	3.6k	57%	22%
7B	2.4k	47%	1.7k	63%	41%

Table 4: Comparison between datacentre and gaming GPUs

	H100	RTX 4090	Ratio
BF16 [TFLOP/s]	989.4	165.2	6×
Memory [GB]	80	24	3.3×
Bandwidth [TB/s]	3.3	1	3.3×
Cost [$]	30k	2k	15×
Power [W]	700	450	1.5×
Communication	NVLink	PCIe 4.0	–
Bandwidth [GB/s]	900	64	14

Impact of FP8 The relative speed-up due to lower precision is shown in the SP column, and reaches about 50% for sufficiently large models. For very small models, non-gemm operations contribute significantly to the total running time, so speeding up the GEMMs only provides limited improvements. Yet, FP8 comes with additional overheads due to dynamic quantization, and the transposes required in the backward pass. As the model size increases, compute is dominated by matrix math, and FP8 can lead to significant boosts. Due to the fixed batch size, for very large models, the optimizer step starts taking a noticeable fraction of the time.

Notice that part of the computations are still done in 16-bit precision. For the 7B model, the operations break down to 39.2×10^9 ops in FP8 for the linear operations inside the transformer blocks, 3.3×10^9 operations in bf16 for the LM-head, and 0.6×10^9 operations in BF16 for attention. Con-

Table 5: Effect of `nccl` collectives compared to `cudaMemcpy`-based communication of consumer (RTX 4090) and professional (L40S) cards for the 14B model. Comparing `nccl` (None) with enabling `memcpy` only for one of the collectives (Gather, Scatter) and using `memcpy` for all (large) collectives.

GPU	FP8				BF16			
	None	Gather	Scatter	Full	None	Gather	Scatter	Full
$4 \times$ 4090	4.3k	6.3k	5.3k	7.8k	2.9k	4.6k	3.6k	5.2k
$4 \times$ L40S	9.5k	10k	9.9k	9.9k	6.8k	7.0k	7.1k	6.9k

sequently, even if there were no quantization-related overheads, we could expect FP8 to only bring a speed-up of 90%. Despite falling short of the theoretical maximum, the achieved speed-ups are generally in line with existing FP8 implementations [31, 32].

Interestingly, significant benefits from FP8 generally do *not* come from enabling a larger batch size due to lower activation memory requirements. For small models, overheads limit the effectiveness of FP8, and for large models, activation recomputation means that we do not store the activations that would be compressed to FP8 (e.g., residuals remain in BF16); even worse, FP8 requires additional buffers for transposes and quantization, thus actually using *more* memory when entire transformer blocks are recomputed.

Comparing the RTX 5060Ti and the 4090, we generally get slightly better utilization on the smaller card. While this is expected for small models, at first it might be surprising for larger models; however, the 5060Ti has only about 1/3 of the FLOP/s of the 4090, yet provides the same PCIe bandwidth, so even though it requires more offloading, it is also better able to hide the resulting latencies.

Llama-Factory Comparison Finally, the "LF" column show the speed of LLama-Factory (LF)[33] on the 4090. For small models, the llmq implementation is significantly faster, but for large models, the difference almost disappears. Interestingly, the strategies to achieve optimal performance at increasing model size differ between the two frameworks. For llmq, parts of the model are offloaded until at least a moderate batch size can be sustained; for LF, however, we found that, as soon as offloading is required, it is more efficient to do full offloading in order to support a very large batch size, than to do partial offloading at medium batch sizes. We attribute this to the fact that the overheads associated with each forward/backward propagation are much lower in llmq than in LLama-factory.

Scaling up to multiple GPUs, Table 2 shows training speeds on a system with 4 consumer GPUs (4090), compared against 4 professional-grade GPUs (L40s). We can see that, despite their nominally much faster peak FLOP/s, for small models the L40S are only a little faster than the 4090s, especially in FP8, and even for large models, the gap is lower than expected, as the relative utilization on the L40S is considerably lower. This appears to be because the L40S run into thermal and/or power throttling, never achieving actual peak performance, see also appendix A.3.

The comparison against LLama-factory shows the tremendous benefit conferred by efficient `memcpy`-based communication. For small models that do not require parameter sharding, there is only a minor performance gap (52k vs 41k for 1.5B parameters in BF16), but at the largest scale supported by LF, 14B, the llmq implementation is twice as fast.

In Table 5, we show the effect of using `memcpy`-based communication compared to `nccl` collectives, on a system with consumer GPUs without direct peer-to-peer support, and on a professional setup with peer-to-peer capable GPUs. The data shows that in the consumer setup, `memcpy` is essential for good performance, whereas there is only a minor difference on the professional hardware. In that setup, using a combination of `nccl` and `memcpy` communication ends up faster than either option.

DGX Spark Results Finally, we consider performance on the recent NVIDIA DGX Spark, shown in Table 3. This is interesting because it offers 128 GB of *unified* memory shared between GPU and CPU, so models up to a size of 7B can fit on a single device without any offloading. The large memory capacity comes at the cost of low memory bandwidth: at $300\,\mathrm{GB\,s^{-1}}$, it is even slower than

the 5060Ti's $448\,\mathrm{GB\,s^{-1}}$, though still faster than PCIe. However, the lack of faster device memory altogether means that *any* memory-bound kernel with a working set larger than the L2 cache will operate at this reduced speed, whereas for the consumer GPUs we are able to mostly hide the PCIe limitations behind explicit prefetching into DRAM. This also explains why we see little speed-up from switching to FP8 with smaller models: It is precisely these memory-bound kernels that do not benefit from FP8, and the additional transposes and conversions just add more memory transfers. Only at 7B do the matrix multiplications make up enough of the total time for reduced-precision to provide substantial benefits. Surprisingly, even in pure BF16, the Spark achieves less MFU than the 5060Ti, despite the latter having to rely on activation checkpointing. This is probably due to power limits preventing the system from reaching its advertised performance.

5. End-to-end results

Scenario 1: 1.5B pre-training Frist, consider pre-training a 1.5B parameter Qwen-style model, to be trained on 4 × RTX 4090. As training data, we use a retokenized and subsampled version of ClimbMix [34] with 10B tokens and 30B tokens. Due to the long training time involved, we only provide the FP8 version of the 30B run. The results are presented in Figure 2. We can see that the E4M3 (forward and backward) curve closely tracks that of BF16. In contrast, using E5M2 for activation gradients during backward leads to a minor degradation in performance, in contrast to traditional recommendations [16, 35, 36].

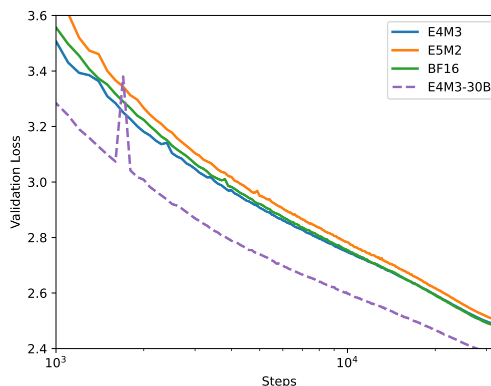

Figure 2: Validation loss vs # optimizer steps for FP8 and BF16 training on 10B tokens (solid lines), as well as 30B tokens (dashed) for FP8. The 30B run uses a 3x larger batch size.

Scenario 2: 0.5B long pretraining run We also perform a much longer training run for a 0.5B model over around 350 billion tokens to verify the stability of the training recipe. While we do observe growing activations in the swiglu layer, at least at this scale they are not strong enough to destabilize the training, which progressed without loss spikes. Details of the run can be found in the appendix A.4.

Scenario 3: GSM8k As a third test, we fine-tune LLama2-7B [37] and Qwen2.5-14B [38] on the GSM8k[39] dataset. As LLama2 has not been specifically pre-trained with math data, this provides a good demonstration that the performance of LLMs can be improved substantially on narrow domains even with modest computational resources. On the other hand, Qwen models have deliberately used large amounts of mathematical reasoning data during their pre-training, so it is already very good in a few-shot setting, but fails in the zero-shot setting, where performance can be mostly recovered through fine-tuning. Both BF16 and FP8 fine-tuning are able to recover performance.

The results can be seen in Table 6. Training in FP8 does not result in noticable performance degradation compared to the BF16 baseline; On the other hand, FP8 QAT does confer consistent benefits when inference is also run in FP8. Most notably, while Qwen2.5-14B has already excellent performance in the direct five-shot setting, and receives no benefit from further fine-tuning, there is a significant gap for FP8 inference in the original model that is close with additional fine-tuning.

Table 6: Performance of LLama2-7B on GSM8K, comparing the original pre-trained model against fine-tuned (T) in BF16 and in FP8, while running inference (I) in either BF16 or FP8. Shown is the average over five independent runs, as well as the standard deviation.

Model↓ T→ I→	n-shot	Pretrained BF16	Pretrained FP8	LLMQ BF16 BF16	LLMQ BF16 FP8	LLMQ FP8 BF16	LLMQ FP8 FP8
LLama2-7B	5	13.5	13.7	34.5 ± 4.3	34.4 ± 4.2	37.6 ± 1.6	37.2 ± 1.1
LLama2-7B	0	6.0	4.9	35.7 ± 1.5	35.4 ± 1.6	36.6 ± 0.9	36.4 ± 2.0
Qwen2.5-14B	5	84.1	80.7	84.5 ± 0.9	83.5 ± 0.7	84.0 ± 0.6	83.9 ± 0.3
Qwen2.5-14B	0	14.0	17.8	80.4 ± 1.1	79.1 ± 0.7	82.7 ± 0.6	83.1 ± 0.5

6. Limitations

It is often observed that low-precision works successfully shorter training runs, but breaks down when scaled up [36]. However, the intended use case for the software described in this paper are heavily compute-constrained environments, where we do not expect training to run far beyond the Chinchilla-optimal [40] ratio; e.g., for the 1.5B model, training 200B tokens in FP8 on 4 RTX 4090 would take about 5 weeks.

Acknowledgments

We would like to thank contacts at NVIDIA (Vartika Singh, Nina Carrejo, Kyla Wilkes, and Tijmen Blankevoort), HP (Curtis Burkhalter), and Datacrunch/Verda (Paul Chang and Antonio Dominguez) for hardware support that was essential to this project. ES was supported in part by ERC Proof-of-Concept grant FastML.

References

[1] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[2] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL `https://proceedings.neurips.cc/paper_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf`.

[3] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble (almost) pipeline parallelism. In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=tuzTN0eIO5`.

[4] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ringattention with blockwise transformers for near-infinite context. In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=WsRHpHH4s0`.

[5] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5:341–353, 2023.

[6] Nouamane Tazi, Ferdinand Mom, Haojun Zhao, Phuc Nguyen, Mohamed Mekkouri, Leandro Werra, and Thomas Wolf. The ultra-scale playbook: Training llms on gpu clusters, 2025.

[7] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[8] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[9] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. {Zero-offload}: Democratizing {billion-scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[10] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–14, 2021.

[11] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[12] Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via block-wise quantization. In *International Conference on Learning Representations*, 2022. URL `https://openreview.net/forum?id=shpkpVXzo3h`.

[13] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=r1gs9JgRZ`.

[14] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. Fp8 formats for deep learning, 2022. URL `https://arxiv.org/abs/2209.05433`.

[15] Dhiraj D. Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan M. Srinivasan, Abhisek Kundu, Mikhail Smelyanskiy, Bharat Kaul, and Pradeep K. Dubey. A study of bfloat16 for deep learning training. *ArXiv*, 2019.

[16] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi (Viji) Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL `https://proceedings.neurips.cc/paper_files/paper/2019/file/65fc9fb4897a89789352e211ca2d398f-Paper.pdf`.

[17] Bita Darvish Rouhani, Ritchie Zhao, Ankit More, Mathew Hall, Alireza Khodamoradi, Summer Deng, Dhruv Choudhary, Marius Cornea, Eric Dellinger, Kristof Denolf, et al. Microscaling data formats for deep learning. *arXiv preprint arXiv:2310.10537*, 2023.

[18] Andrei Panferov, Jiale Chen, Soroush Tabesh, Mahdi Nikdan, and Dan Alistarh. QuEST: Stable training of LLMs with 1-bit weights and activations. In *Forty-second International Conference on Machine Learning*, 2025. URL `https://openreview.net/forum?id=I0Ux2nAN6u`.

[19] David A Patterson and John L Hennessy. *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann, 2016.

[20] Jack Choquette and Wish Gandhi. Nvidia a100 gpu: Performance & innovation for gpu computing. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–43. IEEE Computer Society, 2020.

[21] Nvidia blackwell architecture technical brief, 2024. URL https://resources.nvidia.com/en-us-blackwell-architecture.

[22] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report, 2025. URL https://arxiv.org/abs/2412.19437.

[23] Vidit Jain, Jatin Prakash, Deepak Saini, Jian Jiao, Ramachandran Ramjee, and Manik Varma. Renee: End-to-end training of extreme classification models. *Proceedings of Machine Learning and Systems*, 2023.

[24] Pin-Lun Hsu, Yun Dai, Vignesh Kothapalli, Qingquan Song, Shao Tang, Siyu Zhu, Steven Shimizu, Shivam Sahni, Haowen Ning, Yanning Chen, and Zhipeng Wang. Liger-kernel: Efficient triton kernels for LLM training. In *Championing Open-source DEvelopment in ML Workshop @ ICML25*, 2025. URL https://openreview.net/forum?id=36SjAIT42G.

[25] Haocheng Xi, Han Cai, Ligeng Zhu, Yao Lu, Kurt Keutzer, Jianfei Chen, and Song Han. COAT: Compressing optimizer states and activations for memory-efficient FP8 training. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=XfKSDgqIRj.

[26] Tianjin Huang, Haotian Hu, Zhenyu Zhang, Gaojie Jin, Xiang Li, Li Shen, Tianlong Chen, Lu Liu, Qingsong Wen, Zhangyang Wang, et al. Stable-spam: How to train in 4-bit more stably than 16-bit adam. *arXiv preprint arXiv:2502.17055*, 2025.

[27] Ionut-Vlad Modoranu, Mher Safaryan, Grigory Malinovsky, Eldar Kurtić, Thomas Robert, Peter Richtárik, and Dan Alistarh. Microadam: Accurate adaptive optimization with low space overhead and provable convergence. *Advances in Neural Information Processing Systems*, 37:1–43, 2024.

[28] Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. Galore: Memory-efficient LLM training by gradient low-rank projection. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=hYHsrKDiX7.

[29] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pages 4596–4604. PMLR, 2018.

[30] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1 (2):3, 2022.

[31] Houwen Peng, Kan Wu, Yixuan Wei, Guoshuai Zhao, Yuxiang Yang, Ze Liu, Yifan Xiong, Ziyue Yang, Bolin Ni, Jingcheng Hu, Ruihang Li, Miaosen Zhang, Chen Li, Jia Ning, Ruizhe Wang, Zheng Zhang, Shuguang Liu, Joe Chau, Han Hu, and Peng Cheng. Fp8-lm: Training fp8 large language models. 2023.

[32] Alejandro Hernández-Cano, Dhia Garbaya, Imanol Schlag, and Martin Jaggi. Towards fully FP8 GEMM LLM training at scale. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=KYTFXxTJ12.

[33] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics* (*Volume 3: System Demonstrations*), Bangkok, Thailand, 2024. Association for Computational Linguistics. URL http://arxiv.org/abs/2403.13372.

[34] Shizhe Diao, Yu Yang, Yonggan Fu, Xin Dong, Dan Su, Markus Kliegl, Zijia Chen, Peter Belcak, Yoshi Suhara, Hongxu Yin, Mostofa Patwary, Celine Lin, Jan Kautz, and Pavlo Molchanov. Climb: Clustering-based iterative data mixture bootstrapping for language model pre-training. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2504.13161.

[35] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, et al. Fp8 formats for deep learning. *arXiv preprint arXiv:2209.05433*, 2022.

[36] Maxim Fishman, Brian Chmiel, Ron Banner, and Daniel Soudry. Scaling FP8 training to trillion-token LLMs. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=E1EHO0imOb.

[37] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[38] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

[39] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[40] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Oriol Vinyals, Jack W. Rae, and Laurent Sifre. Training compute-optimal large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

[41] Alex Hägele, Elie Bakouch, Atli Kosson, Leandro Von Werra, Martin Jaggi, et al. Scaling laws and compute-optimal training beyond fixed training durations. *Advances in Neural Information Processing Systems*, 37:76232–76264, 2024.

[42] Shengding Hu, Yuge Tu, Xu Han, Chaoqun He, Ganqu Cui, Xiang Long, Zhi Zheng, Yewei Fang, Yuxiang Huang, Weilin Zhao, et al. Minicpm: Unveiling the potential of small language models with scalable training strategies. *arXiv preprint arXiv:2404.06395*, 2024.

[43] Joonhyung Lee, Jeongin Bae, Byeongwook Kim, Se Jung Kwon, and Dongsoo Lee. To fp8 and back again: Quantifying the effects of reducing precision on llm training stability. *CoRR*, 2024.

[44] Mitchell Wortsman, Peter J Liu, Lechao Xiao, Katie E Everett, Alexander A Alemi, Ben Adlam, John D Co-Reyes, Izzeddin Gur, Abhishek Kumar, Roman Novak, Jeffrey Pennington, Jascha Sohl-Dickstein, Kelvin Xu, Jaehoon Lee, Justin Gilmer, and Simon Kornblith. Small-scale proxies for large-scale transformer training instabilities. In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=d8w0pmvXbZ`.

[45] Shizhe Diao, Yu Yang, Yonggan Fu, Xin Dong, Dan SU, Markus Kliegl, ZIJIA CHEN, Peter Belcak, Yoshi Suhara, Hongxu Yin, Mostofa Patwary, Yingyan Celine Lin, Jan Kautz, and Pavlo Molchanov. Nemotron-CLIMB: Clustering-based iterative data mixture bootstrapping for language model pre-training. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025. URL `https://openreview.net/forum?id=aBlqKPkc4a`.

[46] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[47] Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. The language model evaluation harness, 07 2024. URL `https://zenodo.org/records/12608602`.

[48] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *ArXiv*, abs/1803.05457, 2018.

[49] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

[50] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *arXiv preprint arXiv:1907.10641*, 2019.

[51] Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, , and Jason Wei. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*, 2022.

[52] Stephanie Lin, Jacob Hilton, and Owain Evans. TruthfulQA: Measuring how models mimic human falsehoods. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3214–3252, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.229. URL `https://aclanthology.org/2022.acl-long.229`.

[53] Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *EMNLP*, 2018.

[54] Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. Piqa: Reasoning about physical commonsense in natural language. In *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.

[55] Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. Boolq: Exploring the surprising difficulty of natural yes/no questions. In *NAACL*, 2019.

[56] Siva Reddy, Danqi Chen, and Christopher D Manning. Coqa: A conversational question answering challenge. *Transactions of the Association for Computational Linguistics*, 7:249–266, 2019.

[57] Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The lambada dataset, August 2016. URL `https://doi.org/10.5281/zenodo.2630551`.

[58] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

[59] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.

Table 7: Offloading: x - residual, m, v - optimizer moments, θ^*, θ - (master) parameters, g - gradient

GPU	Size	DType	Batch	Recompute	Offload
5060Ti	0.5B	FP8	12	—	—
	0.5B	BF16	10	—	—
	1.5B	FP8	8	Block	x
	1.5B	BF16	8	FFN, Att	—
	3B	FP8	12	Block	m, v, θ^*
	3B	BF16	12	QKV, FFN	m, v, θ
	7B	FP8	32	Block	$x, m, v, g, \theta, \theta^*$
	7B	BF16	32	Block	x, m, v, g, θ
4090	0.5B	FP8	16	—	—
	0.5B	BF16	16	—	—
	1.5B	FP8	4	—	—
	1.5B	BF16	4	—	—
	3B	FP8	4	—	m, v, θ^*
	3B	BF16	4	SwiGLU	m, v
	7B	FP8	16	Block	$m, v, \theta^*, \theta, x$
	7B	BF16	16	Block	m, v, θ, x
	14B	FP8	32	Block	$x, m, v, g, \theta, \theta^*$
	14B	BF16	32	Block	x, m, v, g, θ

Table 8: Configurations for LLama-factory. Activation checkpointing is enabled in all settings.

	1x4090		4x4090	
Size	Batch	Offload	Batch	Offload
0.5B	128	—	128	—
1.5B	16	—	32	—
3B	48	ZeRO-2	64	ZeRO-3
7B	32	ZeRO-3	32	ZeRO-3
14B	20	ZeRO-3	21	ZeRO-3
32B	OOM	OOM	4	ZeRO-3

A. Appendix

A.1. Optimal Configurations

Table 7 shows the configuration used to achieve the throughput reported in Table 1. The configurations selected for the LLama-factory baseline runs are given in Table 8.

A.2. GSM8k hyperparameters

For LLama2-7B, we train for two epochs at a batch size of $32\,768$ tokens per optimizer step with a sequence length of 2048 and learning rate 2×10^{-5} linearly decaying to 25% after an initial warm-up of 10 steps. This training can be completed in about 30 minutes on a single 4090, or 90 minutes on a 5060Ti.

For Qwen2.5-14B, we train for one epoch with a batch size of 96 at sequence length 512 for a single epoch, decaying the learning rate from 6×10^{-6} to 0 after 20 warm-up steps.

A.3. A note on MFU

At first glance, the MFU values for L40S seem disappointingly low. The "bad" performance can be explained with the fact that the MFU was calculated using official spec-sheet peak FLOP/s, which appear to be unattainable in the hardware configuration in question. For example, running a large

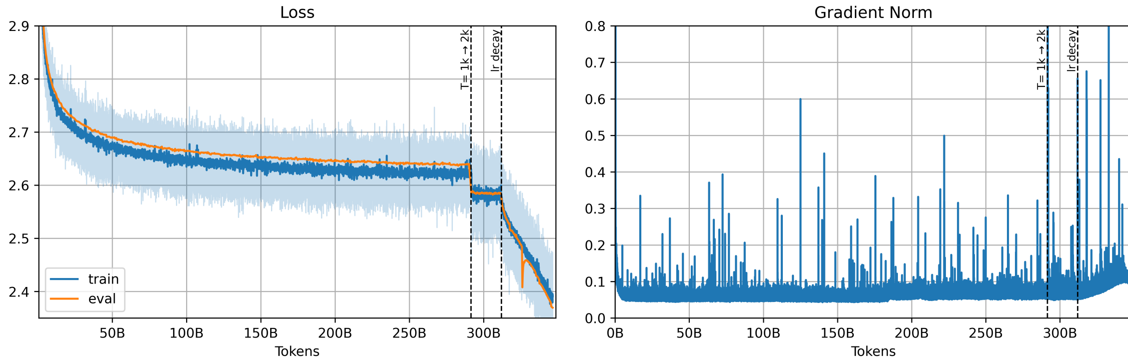

Figure 3: Loss (left) and gradient norm (right) for a 0.5B model over 350B tokens.

matrix multiplication ($16384 \times 16384 \times 16384$) on a single GPU only achieves about $270\,\mathrm{TFLOP\,s^{-1}}$, just $^3/_4$ of the advertised $362\,\mathrm{TFLOP\,s^{-1}}$ peak performance.

For the 5060Ti, we found that a single large matmul managed to achieve up to 108% of the supposed peak FLOP/s, so the MFUs are slight overestimates. In case of the 4090, we similarly benchmark around 103% of peak performance.

For the DGX Spark, similar to the L40S, we see about 70% of peak flops in a matmul micro-benchmark.

A.4. Long training run in FP8

Fishman et al. [36] observed that FP8 training, even though it initially progresses promisingly, can incur sudden instabilities that are not present in corresponding BF16 runs. In particular, they found that after around 200 billion tokens (for a 7B model), the activations inside SwiGLU started to grow to extreme values, making tensor-level scaling ineffective. Based on the reported observation that small FP8 models diverge earlier than large ones [32] and due to computational resource constraints, we train a 500 million parameter model on about 350 billion tokens. To be flexible in the training-run scale, we chose a warump-stable-decay [41, 42] learning rate schedule. We chose a relatively high learning rate of 0.002, as larger learning rates are expected to exacerbate instabilities [43, 44]. Learning-rate decay starts at 312 billion tokens and uses a $1 - \mathrm{sqrt}$ schedule. We performed most of the run with a sequence length of 1024, but extended to 2048 after 290 billion tokens.

The training data is from Nemotron-climb[45]; our number of tokens is lower than the reported number because we use the Qwen tokenizer, which has a much larger vocabulary than the tokenizer used in Diao et al. [45].

Training progressed smoothly in our setup. As shown in Figure 3, neither loss nor norm show large spikes (note that the spikes visible in the norm graph still remain below the gradient clipping threshold of 1.0).

In Figure 4, we show the development of tensor-level abs-max for swiglu activations in selected layers; in particular layer 5 and layer 23 showed huge activations. While we can see large values, larger than could be represented in fp8 without global scaling, they still remain an order of magnitude smaller than the values reported in [36]. We suspected that the large values seen in layer 23, which is the last transformer block before the LM-head, might be related to unrestricted growth of the logits values, a known failure mode of LLMs. Therefore, we implemented z-loss regularization [46] and enabled it with a weight of 1×10^{-4} (taken from Palm [46]) at step 350k.

While we observe a clear effect on the average log-sum-exp of the logits (i.e., the term that z-loss regularization penalizes) as shows in Figure 5, the largest value observed in each batch does not decrease after enabling the regularization. During the decay phase of the learning-rate schedule,

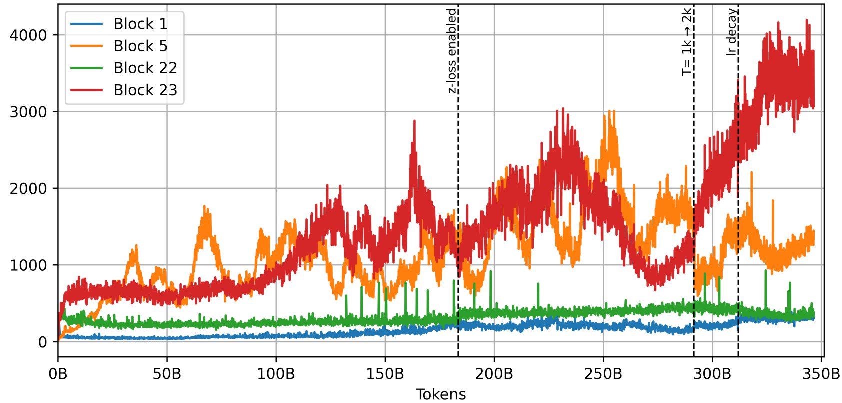

Figure 4: Tensor-level abs-max of SwiGLU activations in layers 1, 5, 22 and 23.

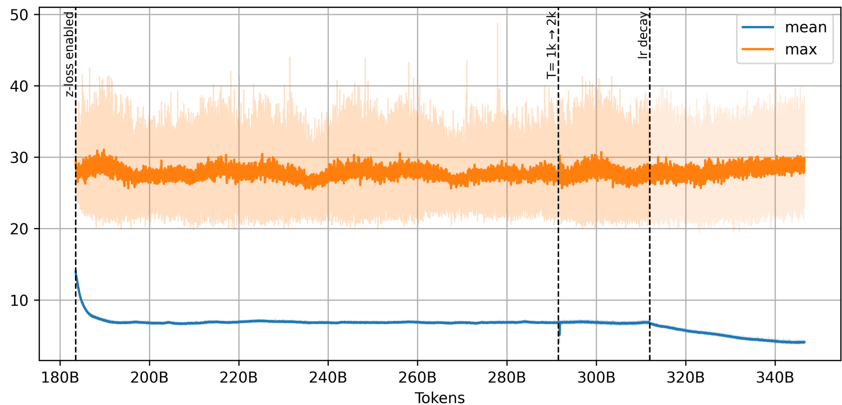

Figure 5: Average and maximum log-sum-exp (i.e., log of softmax normalizer) in loss calculation.

we see a further reduction in mean lse, again without corresponding decreases in maximum lse. There is also no clear effect on the activation magnitude in the last layer.

Finally, we perform downstream evaluations of the created model. We evaluate both the final model, as well as intermediate checkpoints where we perform an additional LR decay. We use `lm_eval`[47] to run evaluations on `arc` [48], `hellaswag` [49], winogrande [50], `bbh` [51], `truthfulqa` [52], `gsm8k` [39], `openbookqa` [53], `piqa` [54], `boolq` [55], `coqa` [56], `lambada` [57] as well as `mmlu` [58] in both multiple-choice and continuation format. Additionally, we evaluate languge modelling on WikiText [59] for 1024 and 2048 context window sizes. The results are presented in Table 9.

The most striking difference is observable in GSM8k: As the pretraining data of Qwen has been augmented with math documents, Qwen can perform well on this task even without additional supervised fine-tuning, whereas our model and TinaLLama fail completely. To a lesser degree, the same applies to `mmlu` in multiple-choice format. On the other hand, on some tasks, our model actually outperforms the others despite being trained on much fewer tokens. It must be cautioned, though, that some tasks such as `hellaswag` and `arc` were used specifically during the data-selection phase for ClimbMix. For knowledge-intensive tasks such as `mmlu`, the models trained for trillions of tokens are much better, unsurprisingly.

We'd like to emphasize that the goal of this section is not to train a state-of-the-art small language model; that would require an even longer training horizon and more careful data engineering. What we show here is that the presented approach can perform stable training in FP8 precision over hun-

Table 9: Evaluation at different training checkpoints, after an additional LR-decay, as well as comparison against models of similar size. These are Qwen2-0.5B (Q2), Qwen2.5-0.5B (Q2.5), and TinyL-Lama (TL).

Task	**Metric**		**100B**	**192B**	**292B**	**350B**	**Q2**	**Q2.5**	**TL**
arc_challenge	acc	↑	36.52	38.99	37.97	**39.33**	24.91	29.18	30.97
	norm	↑	38.82	40.36	40.36	**41.21**	28.75	31.91	32.68
arc_easy	acc	↑	71.13	72.14	72.31	**73.11**	54.67	64.73	61.66
	norm	↑	66.25	66.75	40.36	**69.57**	50.51	58.21	55.47
hellaswag	acc	↑	42.74	43.34	44.29	44.55	38.32	40.59	**46.70**
	norm	↑	55.90	56.86	57.99	58.55	48.92	52.16	**61.47**
winogrande	acc	↑	55.72	56.67	58.64	56.27	57.70	56.35	**59.43**
bbh	exact	↑	19.54	17.77	17.42	22.42	27.75	**31.09**	25.93
truthfulqa_mc1	acc	↑	24.11	24.97	24.60	**26.81**	24.60	25.21	22.64
truthfulqa_mc2	acc	↑	37.75	38.33	36.74	**40.08**	39.75	39.84	35.44
gsm8k	exact	↑	1.82	0.76	1.29	1.29	**36.09**	33.43	1.29
openbookqa	acc	↑	27.00	27.40	29.20	**30.50**	22.60	24.40	25.20
	norm	↑	36.60	37.40	39.40	**41.60**	32.80	34.80	36.80
piqa	acc	↑	74.59	75.03	75.57	**75.79**	69.37	70.46	72.63
	norm	↑	75.35	74.43	75.41	**75.90**	69.04	70.13	73.56
boolq	acc	↑	60.64	54.59	61.19	**62.66**	60.80	62.14	55.99
coqa	em	↑	39.88	43.30	44.97	44.55	45.20	**49.65**	47.45
	f1	↑	52.09	55.08	55.98	57.18	56.95	**61.46**	59.12
lambada_openai	acc	↑	41.28	43.66	43.94	43.86	50.61	**52.88**	49.29
	ppl	↓	20.67	18.64	17.49	17.36	11.59	**10.69**	24.21
lambada_standard	acc	↑	36.08	37.22	38.93	38.60	**44.60**	44.17	6.27
	ppl	↓	37.50	31.90	27.45	30.14	**15.63**	17.09	67000
mmlu (MC)	acc	↑	24.53	26.89	25.95	25.76	43.98	**47.32**	25.38
−humanities	acc	↑	25.44	26.78	26.16	26.29	41.15	**42.27**	25.74
−other	acc	↑	25.01	28.84	26.71	26.78	48.66	**53.07**	25.56
−social sci .	acc	↑	24.34	26.03	25.25	24.54	50.67	**55.48**	24.21
−stem	acc	↑	22.90	25.98	25.56	25.15	37.04	**41.20**	25.82
mmlu (cont.)	acc	↑	30.53	30.64	30.85	31.37	29.85	31.52	**31.91**
−humanities	acc	↑	26.38	26.82	26.21	26.78	26.42	26.23	**28.74**
−other	acc	↑	36.14	33.37	36.50	37.30	33.44	33.79	**37.75**
−social sci .	acc	↑	32.92	33.12	33.60	33.90	32.01	35.10	**35.42**
−stem	acc	↑	28.86	29.27	29.53	29.91	29.31	**31.72**	27.47
wikitext-1k	bpb	↓	0.87	0.86	0.85	0.85	0.83	0.83	**0.73**
	ppl	↓	25.08	24.56	23.69	23.05	22.15	21.70	**15.35**
wikitext-2k	bpb	↓	–	–	–	0.82	0.81	0.80	**0.71**
	ppl	↓	–	–	–	21.10	20.08	19.64	**14.01**

dreds of billions of tokens. As for why we didn't observe the instabilities reported in Fishman et al. [36], we suspect three contributing factors:

- The model is quite small, and thus generally expected to be more stable, even with the large learning rate.
- The training data is of high quality. As we train for only a "small" amount of tokens, we can choose a strongly-filtered dataset. It has been observed that FP8 training stability benefits even more than BF16 from clean training data [43].
- We use just-in-time abs-max scaling for quantization. Thus, a small number of steps with exploding activations have a much more limiting impact on overall training than they would in a delayed-scaling scheme as it is employed in most alternative frameworks.