FLUX: FAST SOFTWARE-BASED COMMUNICATION OVERLAP ON GPUS THROUGH KERNEL FUSION

A PREPRINT

Li-Wen Chang[*1], **Wenlei Bao**[*1], **Qi Hou**[*1], **Chengquan Jiang**[*1], **Ningxin Zheng**[*1], **Yinmin Zhong**[*2], **Xuanrun Zhang**[*1], **Zuquan Song**[1], **Ziheng Jiang**[1], **Haibin Lin**[1], **Xin Jin**[2], **and Xin Liu**[1]
[1]ByteDance Ltd
{liwen.chang, wenlei.bao, houqi.1993, jiangchengquan, zhengningxin, zhangxuanrun, zuquan.song, ziheng.jiang, haibin.lin, liuxin.ai}@bytedance.com
[2]Peking University
{zhongyinmin, xinjinpku}@pku.edu.cn

June 21, 2024

ABSTRACT

Large deep learning models have demonstrated strong ability to solve many tasks across a wide range of applications. Those large models typically require training and inference to be distributed. Tensor parallelism is a common technique partitioning computation of an operation or layer across devices to overcome the memory capacity limitation of a single processor, and/or to accelerate computation to meet a certain latency requirement. However, this kind of parallelism introduces additional communication that might contribute a significant portion of overall runtime. Thus limits scalability of this technique within a group of devices with high speed interconnects, such as GPUs with NVLinks in a node.

This paper proposes a novel method, Flux, to significantly hide communication latencies with dependent computations for GPUs. Flux overdecomposes communication and computation operations into much finer-grained operations and further fuses them into a larger kernel to effectively hide communication without compromising kernel efficiency. Flux can potentially overlap up to 96% of communication given a fused kernel. Overall, it can achieve up to 1.24x speedups for training over Megatron-LM on a cluster of 128 GPUs with various GPU generations and interconnects, and up to 1.66x and 1.30x speedups for prefill and decoding inference over vLLM on a cluster with 8 GPUs with various GPU generations and interconnects.

1 Introduction

In the rapidly evolving field of deep learning, one of the most significant trends recently has been the development of increasingly large models [1]. This progression towards large models is not merely a pursuit of scale for its own sake, but a strategic response to the diverse and complex challenges encountered across various domains. These large models have demonstrated remarkable proficiency in tasks ranging from natural language processing [2, 3, 4], computer vision [5, 6], to speech recognition [7, 8], showcasing their versatility and effectiveness. By leveraging vast amounts of data and computational power, they have been able to unearth intricate patterns and insights that were previously inaccessible, offering unprecedented opportunities in fields as varied as healthcare [9], finance [10], software development [11], and beyond. This growth in model size correlates strongly with enhanced performance, opening new frontiers in artificial intelligence applications and redefining what machines are capable of achieving.

These large deep learning models typically require training and inference to be distributed, due to their parameters well beyond the memory capacity of one single device. Model parallelism, such as tensor and pipeline parallelism, is applied to overcome this limitation. While pipeline parallelism partitions a model across layers into multiple devices, executing

[*]These authors contributed equally to this work

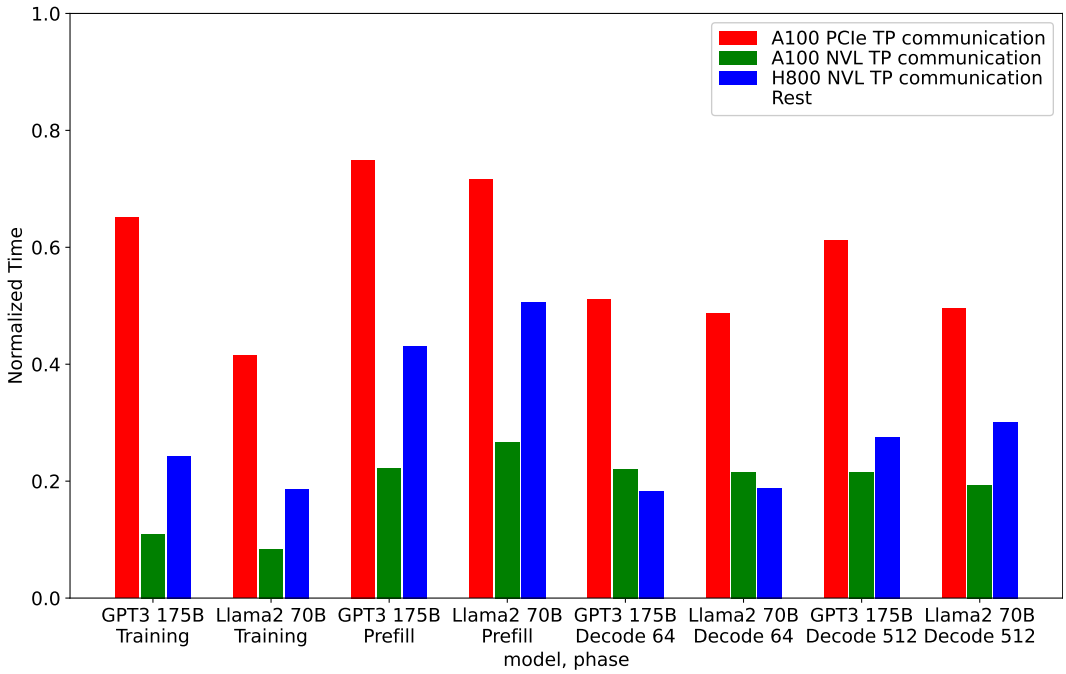

Figure 1: Non-overlapped communication portion within tensor parallelism in common LLM workloads for training with 2-way data, 8-way pipeline, 8-way tensor parallelism on various 128-GPU clusters, and inference with 8-way tensor parallelism on various 8-GPU clusters.

multiple batches in a pipeline fashion, tensor parallelism partitions an individual layer into multiple devices, executing in parallel. Both are important and could be applied together, but they do have different characteristics. Compared to pipeline parallelism improving throughput, tensor parallelism can shrink latency, which is critical for inference. Since tensor parallelism partitions a layer into multiple devices, additional data communication across devices might be required for gathering or (re-)distributing correct data, especially when a consecutive layer applies a different partitioning strategy or consumes data across partitions. Figure 1 shows the substantial portion of communication time over the overall runtime for training and inference specifically for applying tensor parallelism, demonstrating the motivation and strong need to reduce the exposed communication time.

Communication overlapping techniques [12, 13, 14, 15, 16, 17, 18, 19, 20] have become crucial for various kinds of parallelism in training and inferring large deep learning models. The existing overlapping techniques [12, 13, 14] for tensor parallelism decompose a communication operation along with the dependent computation operation into a sequence of chunk, point-to-point operations based on the number of partitions, and carefully execute paired decomposed communication and computation with no data dependence in parallel. These methods might have several limitations on GPUs, such as no precise control of execution timing on GPUs when using streams, and poor GPU utilization for executing multiple smaller kernel instances.

To better overlap communication without compromising GPU utilization, we propose a new overlapping method, Fine-grained Communication Overlapping (Flux), that decomposes the original communication and computation into much finer-grained tiles than the existing methods, and then fuses tiled computation and communication into a single larger kernel. In the fused kernel, each dependent computation and communication tile is mapped into each thread block[1]. Flux optimizes communication together with computation, including kernel fusion, tile coordinate swizzling, GPU instruction selection, communication order selection, etc., and Flux could better adapt the GPU architecture as well as the interconnect over the existing methods. On top of that, Flux is built in a modular way by adopting NVIDIA CUTLASS [21], and can be easily auto-tuned across various combinations of GPU architectures and interconnects. Therefore, Flux can deliver more efficient communication overlapping over the existing methods.

Overall, we make the following contributions in this paper:

- We identify several performance issues when applying the existing communication overlapping techniques for tensor parallelism on GPUs.
- We propose a new novel communication overlapping technique that overcomes the above issues and naturally fits the modern GPU design.

[1]Depending on GEMM implementations, a tile can also be mapped into a warp or a thread block cluster as well.

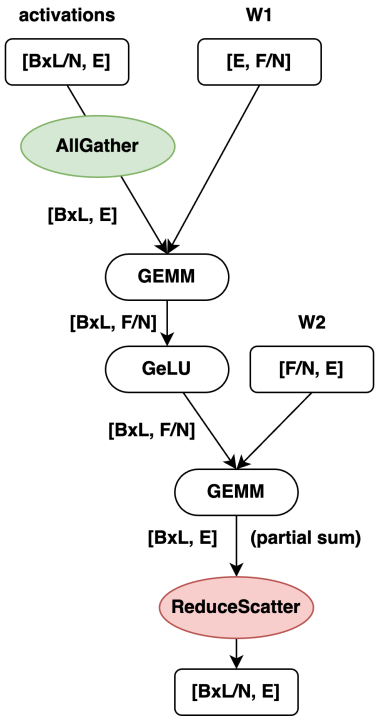

Figure 2: Forward-propagation of the MLP portion with a N-way partitioning across N devices. Here, B and L are flattened to fit the common notation of GEMM.

- We implement the proposed technique using NVIDIA CUTLASS with multiple optimizations for various GPU generations (A100 and H800), and various intra-node interconnects (PCIe and NVLink).
- We evaluate the proposed technique on various 128-GPU clusters for training and 8-GPU clusters for inference. The evaluation results demonstrate up to 1.24x, 1.66x, and 1.30x speedups over the non-overlapping method, such as Megatron-LM [22] and vLLM [23], and 1.38x, 2.06x, and 2.10x speedups over the previous overlapping method, TransformerEngine [14], for training, prefill, and decoding, respectively.

2 Backgrounds on Tensor Parallelism and Overlapping

Tensor parallelism, also called intra-layer model parallelism, is a technique partitioning a layer of a model over multiple devices. This section covers the common state-of-the-art partitioning patterns widely used in large deep learning models, and the corresponding conventional communication overlap strategies.

2.1 Common Partitioning and Communication Patterns

The common partitioning strategy we discuss in the paper is an extended Megatron-LM [24] with sharded activation [13, 25]. For the sake of brevity, we use a multi-layer perception (MLP) portion within a transformer as an example to explain common partitioning and communication patterns. Strategies for other operations, such as multi-head attention, or multi-query attention, can be found in [24, 25, 26].

Figure 2 shows the common communication partitioning pattern in forward-propagation of the MLP example. The first GEMM shards the weight (W1) along the the row direction, and *AllGather*s the sharded input activations along the column direction before GEMM, while the second GEMM shards the weight (W2) along the column direction, and *ReduceScatter*s the output activation along the column direction. In backward-propagation, *AllGather*s and *ReduceScatter*s are interchanged. As the figure shown, the dimensions of these two GEMM operations depends on the degree of tensor parallelism (N). To avoid confusion, in the later sections, when describing problem sizes, we would use a global, original shape, such as $[E, F]$, instead of a local shapes $[E, F/N]$, unless specified otherwise.

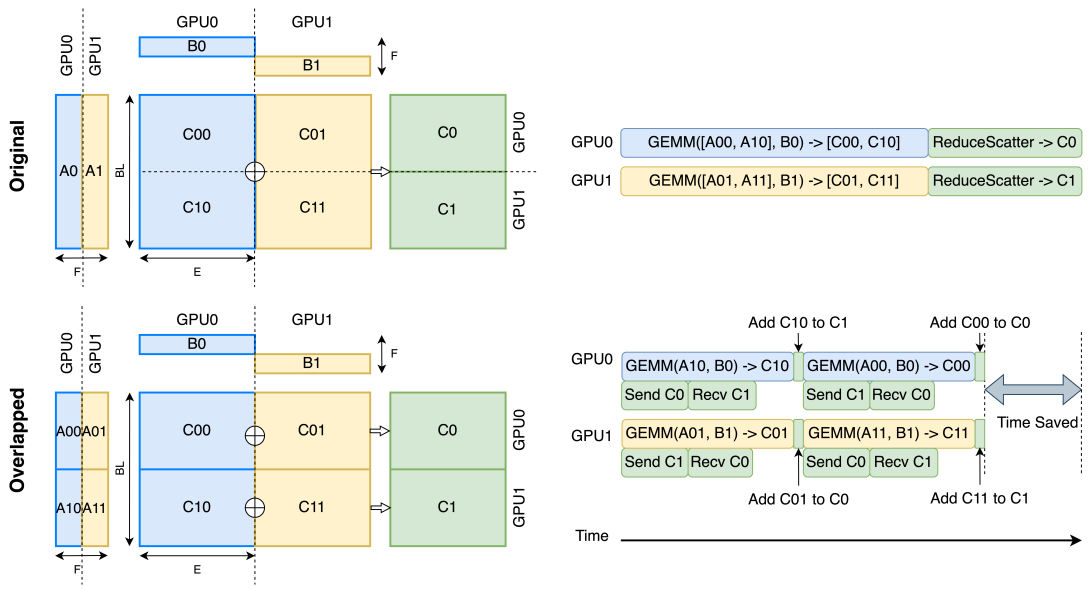

Figure 3: An illustration of the prior GEMM-ReduceScatter overlapping with 2-way tensor parallelism.

Another common partitioning pattern that further shards all weights (W1 and W2) over devices involved in data parallelism, and *AllGather*s weights before GEMM [20, 27, 28, 29, 30]. In this partitioning pattern, since all weights have no data dependence before its consumer GEMM operations, those AllGather operations can be easily prefetched and overlapped with independent operations. Therefore, in the paper, we mainly discuss the first pattern.

2.2 Conventional Communication Overlapping Strategies

现有的 overlapping 方法存在如下问题：(1) tile scheduling 复杂且在实际部署中难以精确控制；(2) 会引入额外的计算操作来合并结果；(3) 切分计算会导致更多的 kernel launch 开销，以及 GPU 的低利用率

Conventional methods [12, 13, 14, 31] decompose the original computation and communication operations into chunks. Then, carefully scheduling operations potentially can overlap communication with computation. The number of partitions in decomposition is aligned with the number of devices in tensor parallelism (or twofold of it to better utilize bidirectional data transfer). Limiting the number of partitions can potentially avoid complicating scheduling and reduce possible scheduling overheads. Figure 3 illustrates a ReduceScatter overlapping scenario[2]. As we can see, ideally communication can be completely hidden by GEMM computation.

These methods might work greatly on TPUs, but not on GPUs, due to different programming models. First, the performance of these methods heavily rely on the execution order, concurrent execution, and execution timing of independent partitions. While the execution order and concurrent execution among GPU kernels can be achieved through streams and events[3], however, the execution timing is not trivially controlled by most GPU programming models. The time variance might be stable and controllable in per-operation evaluation, but it typically becomes unpredictable in the real production environment that involved with many streams and events. Second, ReduceScatter overlapping typically requires performing additional computation operations, such as `add` operations in Figure 3, between GEMM operations, creating data dependence that avoid concurrent execution of multiple GEMM kernels through GPU multiplexing[4]. Although the `add` operations can be further fused with communication [14], they still avoid concurrent execution of multiple GEMM kernels. Last and most importantly, splitting one single large GEMM kernel to multiple smaller GEMM kernels could highly likely underutilize GPU stream processors (SMs) even with a number of partitions as the number of devices, especially when tensor parallelism scales.

2.3 Effective Communication Time and Overlapping Efficiency

It is non-trivial to indicate performance of communication overlapping methods. Overlapping methods typically mix communication with computation, increasing difficulty of directly measuring overlapped time. Moreover, splitting a GEMM kernel into multiple small GEMM ones delivers longer computation time, but longer computation time

[2]Note the existing overlapping method transfers initialized $C0$ and $C1$ in the beginning [13].
[3]Different GPU programming models might have different terminology. In this paper, we mainly use CUDA terminology.
[4]GPU multiplexing can be achieved through multiple CUDA streams.

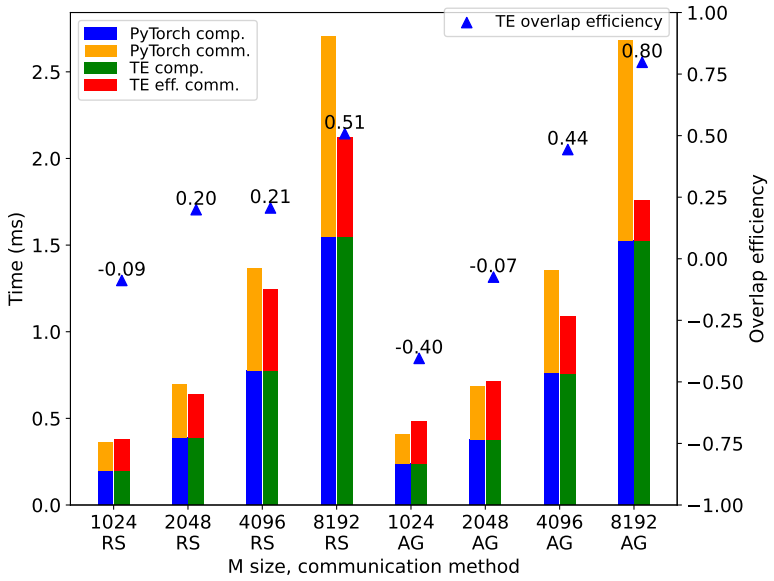

Figure 4: Performance between PyTorch (non-overlapping) and TransformerEngine (prior overlapping method) from m = 1024 to 8192, with (n, k) as (49152, 12288) and (12288, 49152) in AllGather and ReduceScatter on an 8-H800 cluster with NVLink interconnections.

might provide a higher chance to overlap communication. In the end, it may or may not imply shorter overall time. Overall time is a fair performance indicator, but different methods might still use different GEMM algorithm and/or implementation, impacting overall time.

We propose **Effective Communication Time** in Eq. 1 to fairly compare different methods and highlight communication time. Effective communication time (ECT) is defined as overall time ($OverallTime$) minus with **best**, **non-split** GEMM computation time ($GEMM_{non-split}$).

$$ECT = OverallTime - GEMM_{non-split} \tag{1}$$

To minimize impacts of GEMM kernels, we use the *fastest* GEMM kernels from the best of our knowledge in all of the evaluation. Since we use the same, fastest GEMM kernels across methods, given a shape of problem, making $GEMM_{non-split}$ identical across different methods, effective communication time simply has a shift of overall time, but highlights more in communication. Particularly, for a non-overlapping method, effective communication time is equal to regular communication running with the fastest GEMM kernels, while for an overlapping method, slowdown time from any inefficient factor and non-overlapped communication portion all contributes to its effective communication time. It is also worth mentioning that a perfect overlapping method delivers zero effective communication time.

On top of effective communication time, we further define **Overlap Efficiency** ($E_{overlap}$) in Eq 2, as one minus the ratio between effective communication time of an overlapping method ($ECT_{overlap}$) and effective communication time of an non-overlapping baseline ($ECT_{non-overlap}$).

$$E_{overlap} = 1 - \frac{ECT_{overlap}}{ECT_{non-overlap}} \tag{2}$$

To minimize impacts of the non-overlapping baseline, the widely used, standard GPU communication library, NCCL [32], is chosen in all of the evaluation, and NCCL is also the fastest non-overlapping communication from the best of our knowledge. Particularly, overlap efficiency of the non-overlapping baseline is zero, while a perfect overlapping method has a 100% overlap efficiency. An overlapping method with a negative overlap efficiency implies that the method runs slower than the non-overlapping baseline.

Figure 4 shows computation time and effective communication time of PyTorch and the conventional overlapping technique (implemented with TransformerEngine [14]), and the corresponding overlapping efficiency, demonstrating a overlapping technique might deliver poor performance or even worse than the original non-overlapping method, due to the above-mentioned limitations. When m of the matrix is small, the prior overlapping technique delivers worse performance than the non-overlapping baseline (PyTorch), that supports the third reason mentioned earlier. The

Algorithm 1: A simplified GEMM-ReduceScatter (or -AlltoAll) overlapping kernel

Parameters: Input matrix pointers A, B
Parameter: List of output matrix pointers Cs
Parameters: Int scalars $rank\_id$, N_{TP}
$[m, n] \leftarrow$ `TileCoord`$(threadblock\_id, rank\_id, N_{TP})$;
$acc \leftarrow 0$;
standard GEMM prologue(A, B, m, n);
standard GEMM mainloop(A, B, m, n) updating acc;
// epilogue start
$C \leftarrow$ `GetOutput`(Cs, N_{TP}, m, n);
if *fuse reduction* **then**
  `Reduce`(C, acc);
else
  `Write`(C, acc);
end
// epilogue end

prior overlapping technique performs better in AllGather than ReduceScatter. It is mainly because the splited GEMM operations in AllGather might run concurrently through GPU multiplexing, but ReduceScatter cannot, that supports the second reason mentioned earlier.

3 Overview of Fused GEMM with Communication

This paper proposes a more efficient communication overlapping method, Flux, over the conventional methods [12, 14] on GPUs. Different from the existing methods partitioning the computation and communication into the number of devices or twofold of the number, Flux overdecomposes computation and communication into tiles. Here, since the computation operation is GEMM, and most high-performance GEMM kernels on GPUs are written with tiling, such as thread block tiling or warp tiling, our decomposition can naturally map into existing tiling in the kernels. Flux fuses dependent communication and/or wait logic into a GEMM kernel, and launches only one fused kernel, compared to the prior methods launching multiple split GEMM kernels. Considering Flux has much finer-grained than the prior methods, in the rest of the paper, we would refer the prior ones as medium-grained decomposition, and the proposed method as fine-grained decomposition.

3.1 ReduceScatter Overlapping

In Flux, ReduceScatter is implemented as epilogue fusion into a GEMM kernel. More specifically, ReduceScatter communication is fused into the epilogue of the GEMM kernel. Algorithm 1 shows pseudocode of the fused GEMM with ReduceScatter (or AlltoAll) for $C = A \times B$, where A and B are the two input matrices, and Cs are a collection of output matrix pointers on all devices involved in tensor parallelism. Unlike a standard GEMM kernel having only one single output pointer, the number of output pointers (Cs) in the fused GEMM kernel is increased to the number of devices in tensor parallelism (N_{TP}), and can be collected through inter-process communication in the initialization phase of the corresponding PyTorch operation. The output coordinate (m and n) can be computed through a function `TileCoord` with thread block indices and the local rank index($rank\_id$). Selection (`GetOutput`) of an output pointer in the fused GEMM is based on the output coordinate (m and n) and the number of devices in tensor parallelism (N_{TP}). For example, in the two GEMM operations of Figure 2, the selection is based on the row index.

A ReduceScatter operation can be further decoupled to an *AlltoAll* operation and a *reduction* one. Here, AlltoAll refers to only communication across devices, while reduction happens locally on individual devices. Therefore, fusing AlltoAll (`Write` branch) into GEMM epilogue is typically enough to overlap communication, the reduction fusion (`Reduce` branch) only provides marginal performance gain. Section 4.2 would discuss the implementation details of reduction.

This algorithm requires GPUs with peer-to-peer (P2P) supports, which modern NVIDIA GPUs within a node already have, regardless with NVLink or PCIe interconnects. NVSHMEM [33] extends P2P on NVIDIA GPUs across nodes. The detailed implementations of `TileCoord`, `Reduce` and `Write` would be discussed in Section 4.

Algorithm 2: A simplified AllGather-GEMM overlapping kernel

Parameters: Input matrix pointers A_{agg}, B
Parameter: Output matrix pointer C
Parameter: List of scalar $signal\_list$
Parameters: Int scalars $rank\_id$, N_{TP}
$[m, n] \leftarrow$ `TileCoord`$(threadblock\_id, rank\_id, N_{TP})$;
$signal \leftarrow$ `GetSignal`$(signal\_list, N_{TP}, m, n)$;
`WaitSignal`$(signal)$;
standard GEMM(A, B, C, m, n);

Algorithm 3: A host function for AllGather-GEMM overlapping

Parameter: List of input matrix pointers A_list
Parameter: List of output matrix pointer A_{agg}_list
Parameter: List of scalar $signal\_list$
Parameter: Int scalar $rank\_id$, N_{TP}
Parameter: List of tile info $tiles_{comm}$
for $tile$ *from* $tiles_{comm}$ **do**
  if *pull* **then**
    // pull-based
    $A_{remote} \leftarrow$ `GetRemotePtr`$(A\_list, tile)$;
    $A_{local} \leftarrow$ `GetLocalPtr`$(A_{agg}\_list, tile)$;
    `DataTransfer`$(A_{remote}, A_{local}, tile.size)$;
  else
    // push-based
    $A_{remote} \leftarrow$ `GetRemotePtr`$(A_{agg}\_list, tile)$;
    $A_{local} \leftarrow$ `GetLocalPtr`$(A\_list, tile)$;
    `DataTransfer`$(A_{local}, A_{remote}, tile.size)$;
  end
  $signal \leftarrow$ `GetSignalHost`$(signal\_list, tile)$;
  `SetSignal`$(signal)$;
end

3.2 AllGather Overlapping

Different from ReduceScatter, AllGather is implemented as prologue fusion into a GEMM kernel. More specifically, AllGather signal checking is fused into the prologue of the GEMM kernel. Algorithm 2 shows pseudocode of the fused GEMM with AllGather for $C = A_{agg} \times B$, where A_{agg} is the aggregated matrix buffer for input A matrices through AllGather, B is the other input matrix, and C is the output matrix, and Algorithm 3 shows the corresponding communication happening on the host side.

On the kernel side, GEMM tile computation is blocked by the function `WaitSignal` until the value contained in the $signal$ is set to true. Here, the $signal$ is chosen by `GetSignal` from a collection of signals ($signal\_list$) based on the output coordinate (m and n), and the number of devices in tensor parallelism (N_{TP}). For example, in the MLP of Figure 2, the selection is based on the row index. The $signal$ for each communication is only set to true on the host side when the corresponding portion (communication tile) of the input tensor becomes ready, meaning the portion is received on the device running the fused kernel.

The host side (either pull- or push-based) performs tiled communication operations (`DataTransfer`) and set the corresponding signals (`SetSignal`) to true asynchronously. Particularly, the pull-based method transfers tiles by pulling tiles from remote devices through `GetRemotePtr` function and `GetLocalPtr` function choosing the right pointers from a list of the sharded A matrices, A_list, and a list of aggregated matrix buffers, A_{agg}_list, and then setting *local* signals. The $signal$ is chosen by `GetSignalHost` from a collection of signals ($signal\_list$) based on the communication $tile$ index. On the other hand, the push-based one transfers tiles by pushing tiles to remote devices and then setting *remote* signals. Note $signal\_list$ in the pull-based version contains only local signals, while $signal\_list$ in the push-based version contains signals in remote devices. Selection between these two variants is considered as a tuning knob, and is discussed in Section 4.3.

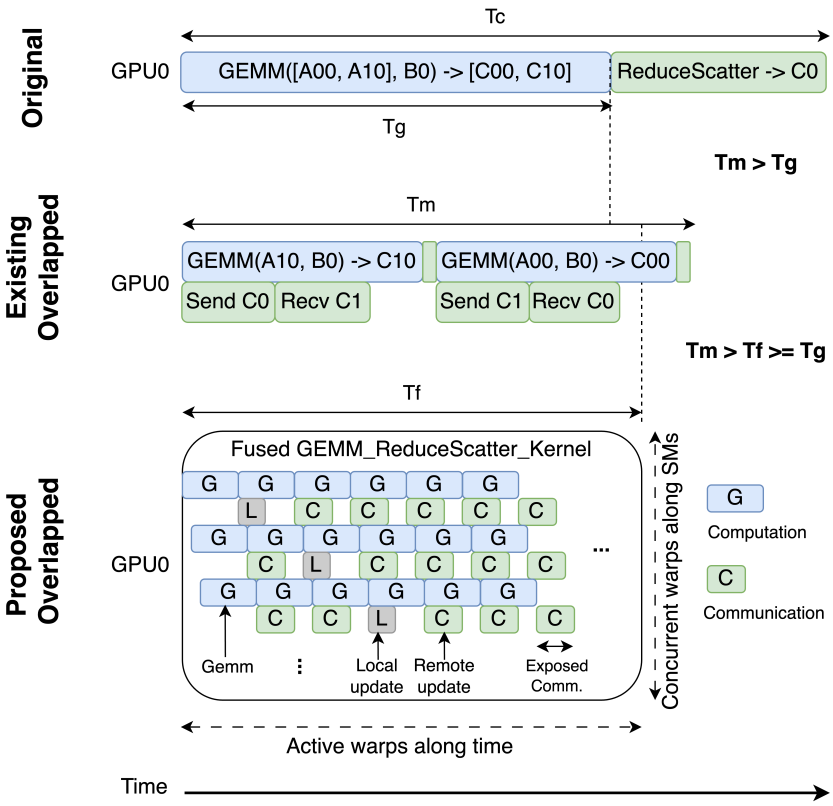

Figure 5: An illustration of differences among the non-overlapping and different overlapping methods in a GEMM-ReduceScatter pattern with 2-way tensor parallelism.

It is worth mentioning that in AllGather our method fuses only the wait logic of communication into the GEMM kernel, instead of entire communication operations. Therefore, AllGather does not necessarily require P2P. Meanwhile, in AllGather, the tiling strategy of communication ($tiles_{comm}$) is decoupled from the tiling strategy of GEMM computation. This design provides a flexible way to choose a trade off between overlapping opportunity and communication efficiency without compromising the GEMM efficiency. Section 4 would discuss all optimizations, and implementation details in the functions `TileCoord`, `WaitSignal`, `SetSignal`, and `DataTransfer`.

3.3 Comparison among Decomposition Strategies

Figure 5 illustrates the major differences among the overlapping techniques in ReduceScatter. Although the existing overlapping (T_m) can potentially perform faster than the original coarse-grained method (T_c), the existing method (T_m) is typically still slower than the original GEMM time (T_g). One major reason is that GPU GEMM efficiency decreases by splitting a GEMM kernel into a sequence of multiple smaller GEMM kernels. GEMM typically requires reasonably large matrices to fully utilize GPU compute power. The sequence of smaller GEMM operations with data dependence further blocks those GEMM kernels from concurrently running through GPU multiplexing, and consequently, the more way of tensor parallelism, the worse GEMM efficiency on GPUs. Compared to the existing method, our proposed technique does not have the above limitation. Our new overlapping technique (T_f) can perform as fast as the original GEMM operation (T_g) with a very small overhead. Its fine-grained decomposition strategy perfectly fits the nature of the modern GPU design, latency hiding among context-switching warps and hundreds of concurrent active warps among SMs, illustrated in the bottom zoom-in view. In the end, our method only exposes a small portion of communication in the tail of execution without compromising GEMM computation efficiency.

Figure 6 illustrates the major differences among the overlapping techniques in AllGather. Similarly, the existing overlapping (T_m) could be faster than the original coarse-grained method (T_c), but is still slower than the original GEMM time (T_g), due to lower GPU GEMM efficiency, and our new overlapping technique (T_f) can deliver a similar performance as the original GEMM operation (T_g). The long latency instruction in AllGather is from waiting signals,

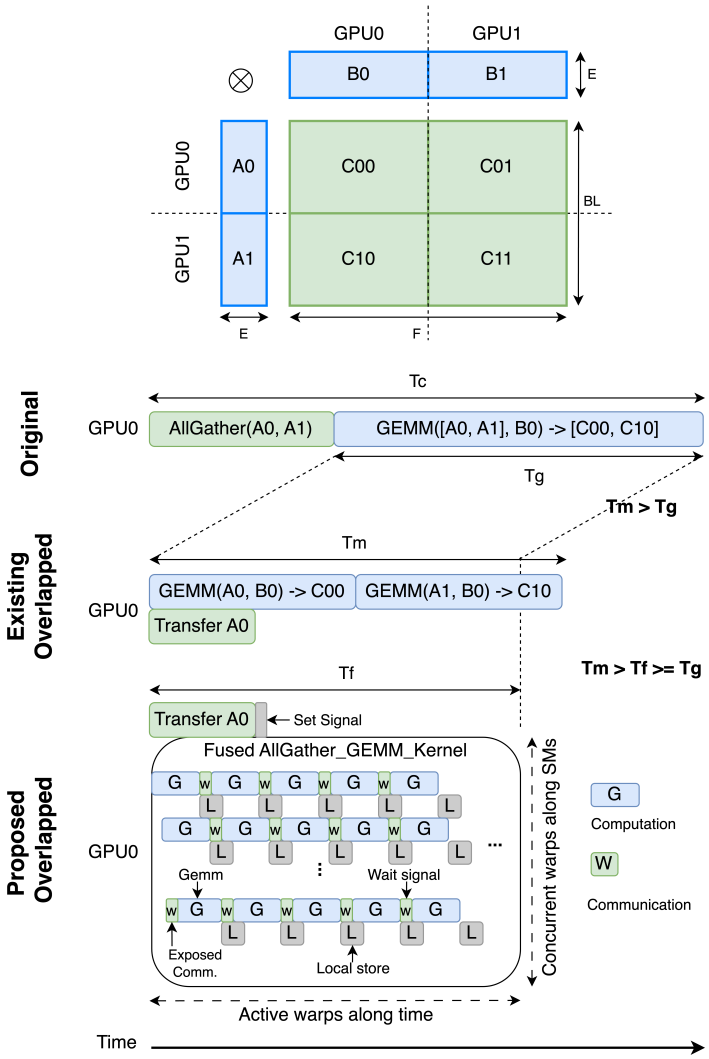

Figure 6: An illustration of differences among the non-overlapping and different overlapping methods in an AllGather-GEMM pattern with 2-way tensor parallelism.

happening in the beginning of each warp since `WaitSignal` is fused in the prologue. Its latency varies based on arrival time of corresponding data transfers. For the tile with data already arrived, the latency is close to zero. For the tile with data not ready, context switching among warps can hide the latency. It is worth mentioning that signals for local tiles are always preset to true, so there is always some warps not needing to wait signals. In the end, our method only exposes a small portion of communication in the head of execution without compromising GEMM computation efficiency. Section 4 further discusses optimization reducing the waiting latency.

4 Optimizations and Implementation Details

As mentioned in Section 3.3, our algorithms fit the nature of the modern GPU design. Therefore, direct implementations of Algorithm 1, 2, and 3 can already outperform the prior methods by delivering better communication overlapping and GEMM efficiency. This section covers advanced optimizations that push the performance to the limit, and the implementation details.

4.1 Tile Coordinate Swizzling

An efficient GPU kernel relies on tiling to exploit parallelism and locality. Therefore, the kernel has a tile mapping logic, such as `TileCoord` in Algorithm 1 and 2, from a thread block index to a tile coordinate. Inspired from a well-tuned GEMM typically swizzling the mapping logic for maximizing memory efficiency [21], we explore tile coordinate swizzling to further improve the efficiency of our fused kernels.

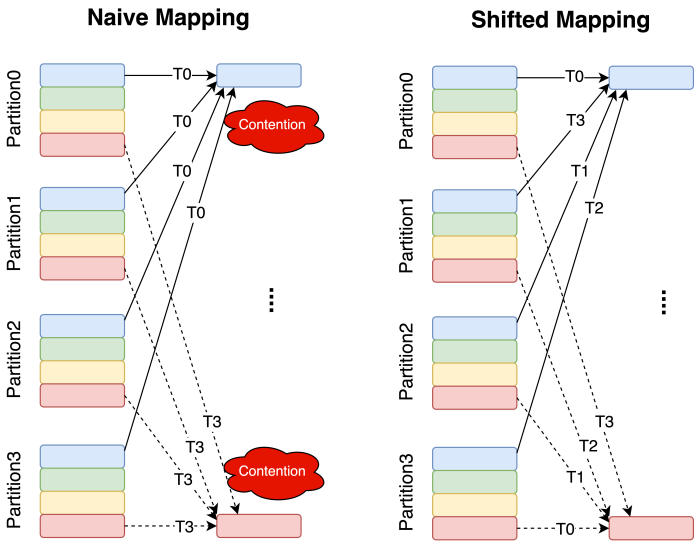

Figure 7: An illustration of memory contention happening at time-step T_i in the naive tile coordinate mapping, and the proposed solution in GEMM-ReduceScatter overlapping with 4-way tensor parallelism.

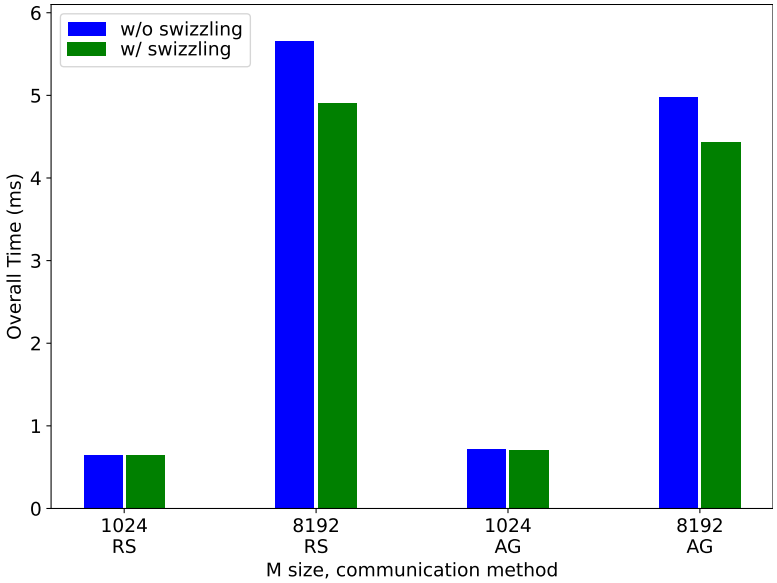

Figure 8: Performance with or without applying tile coordinate swizzling for small (1024) and large (8192) m with (n,k) as (49152, 12288) and (12288, 49152) in AllGather and ReduceScatter, respectively, on an 8-A100 NVLink cluster.

In the fused GEMM-ReduceScatter, tile coordinate is shifted with the device rank index to avoid write request conflicts from the kernels running on different devices, minimizing possible contention in the memory controller on each individual device. Figure 7 illustrates the possible memory contention in a naive mapping, and how a shifted mapping avoids the possible memory contention.

A similar strategy is applied in the fused AllGather-GEMM as well to minimize thread blocks waiting, in the end minimizing overall delay. The fused AllGather-GEMM requires tile coordinate swizzling (`TileCoord`) to align with the order of the signal arrival order, which is determined by the communication order on the host side (controlled by $tiles_{comm}$ in Algorithm 3). In the implementation, these two orders are chosen together based on the network topology to minimize the overall delay, and the more detailed implementation is discussed in Section 4.3.

Figure 8 shows performance impacts before and after applying the tile coordinate swizzling technique on an 8-A100 cluster with NVLink interconnects. The adjusted mapping with tile coordinate swizzling always outperforms the naive mapping. We can also observe that the performance impact increases when the matrix size increases. It is mainly because

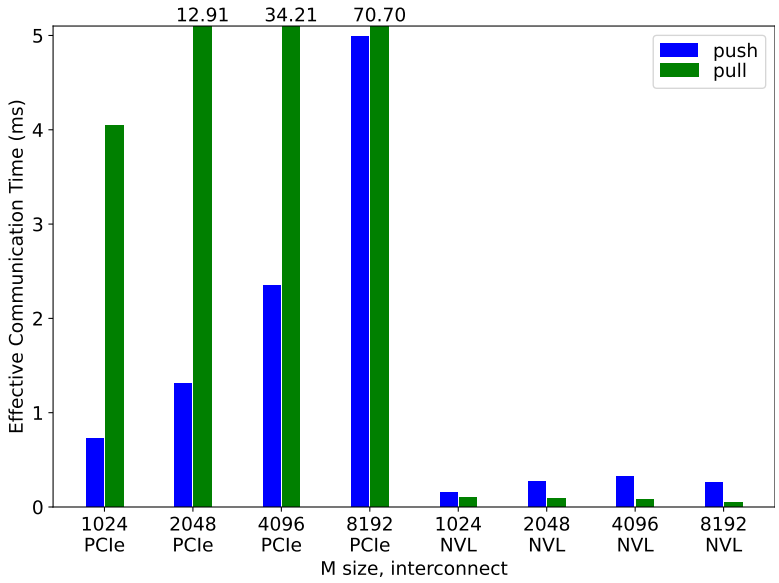

Figure 9: Performance comparison between pull- and push-based data transfers with different m, and (n,k) as (49152, 12288) in AllGather on an 8-A100 cluster with PCIe or NVLink interconnects.

the memory contention of the naive mapping in GEMM-ReduceScatter and the waiting time of AllGather-GEMM also increase when the matrix size increase.

It is worth mentioning that existing method [13] also applied a similar swizzling idea by changing execution order of split GEMM operations. Since our method does not split a GEMM operation, their idea cannot be directly applied in our algorithms.

4.2 Implementation Details of ReduceScatter

Write. Writing data on a local GPU or remote intra-node P2P GPUs is implemented through 1) storing data from registers to global memory using all variants of `st` instructions (including vector versions), 2) storing data from scratchpad to global memory using all variants of `cp.async.bulk` instructions or all variants of Tensor Memory Accelerator (TMA) instructions `cp.async.bulk.tensor` on Hopper GPUs. On the other hand, for writing data on remote inter-node GPUs, NVSHMEM is applied and those writes are implemented through all variants of `put` APIs. All methods are implemented using CUTLASS EVT [34] with templates, and template parameters are chosen during auto-tuning.

Reduce. As mentioned in Section 3.1, reduction can be potentially fused into the GEMM kernel as well. In this case, 1) `red` or atomic instructions can be used to directly implement reduction on device memory without changing code structure or introducing too much overhead if GPUs enable P2P memory access. These kinds of instructions are useful, but might not support all data types or all kinds of GPUs[5]. Therefore, we only apply these instructions for selected data types with capable GPUs. On Hopper GPUs, 2) warp or thread block *specialization*[6] [21] is applied to implement reduction by each GPU writing partial results to its local memory and a specialized warp or thread block pulling ready remote data to perform a local reduction on the destination GPU. These kinds of warp or thread block specialized reduction methods specifically perform well with warp or thread block specialized GEMM kernels on Hopper. For remote inter-node GPUs, we fuse only AlltoAll in the kernel, and perform discrete reduction. All methods are also implemented using CUTLASS EVT with templates, and template parameters are chosen during auto-tuning.

[5]BF16 atomic operations are not supported on A100 and H800.

[6]Warp or thread block specialization is a CUDA programming method on Hopper GPUs allowing a warp or thread block in a kernel to perform a specific task, such as load/store/wgmma and synchronizing warps or thread blocks performing different tasks within a single kernel in a producer-consumer fashion.

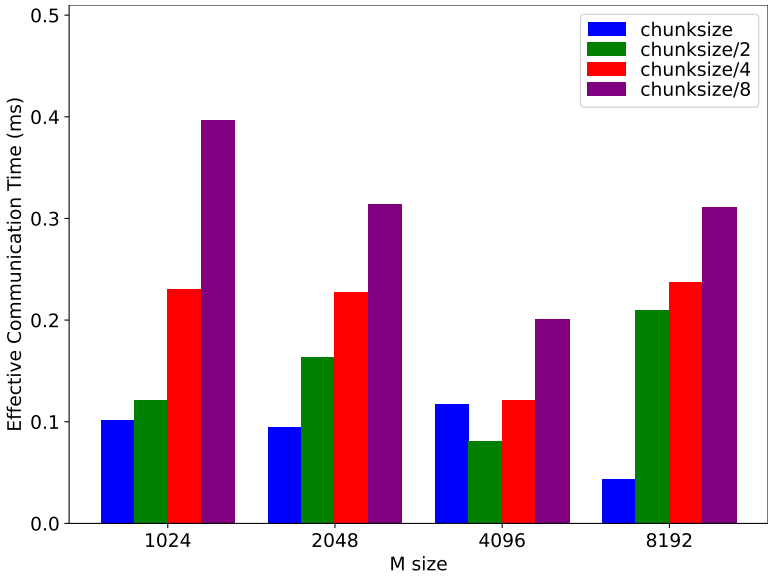

Figure 10: Performance results among different communication tile sizes with different m, and (n,k) as (49152, 12288) in AllGather on an 8-A100 NVLink cluster.

4.3 Implementation Details of AllGather

DataTransfer. Although the proposed AllGather algorithm does not necessarily require P2P, we still separate implementations with and without P2P. For GPUs with P2P memory access, either pull-based or push-based transfers can be implemented with `cudaMemcpy` APIs. The only differences are the pointers. Pull-based uses a local destination pointer and a remote source pointer, while push-based uses the opposite way. Figure 9 shows performance difference between two transfer methods on 8-A100 PCIe and 8-A100 NVLink clusters. As the results shown, different interconnects might have different preference. Therefore, autotuning is applied to select proper transfer methods. On the other hand, for GPUs without P2P access, NCCL [32] send/recv are used. Since NCCL send/recv are paired, there is not difference between pull or push. All methods are implemented in C++ with templates, and template parameters are chosen during auto-tuning.

Signals. We use a regular 32-bit GPU memory to implement a signal. All signals are allocated contiguously for easy preset and reset, and they are preset in the corresponding PyTorch operator's initialization or reset after the corresponding GEMM computation with a stream and an event avoiding data race. On the host side, a signal is set through a `cuStreamWriteValue` API with a stream, while on the kernel, `WaitSignal` is implemented through *spinning*.

Communication tile size. In AllGather, communication tiling is decoupled from GEMM computation tiling for avoiding interfering GEMM tiling, considering GEMM performance is sensitive to GEMM tiling. Tuning communication tiling independently allows us to find a best trade off between overlapping opportunity and communication efficiency, minimizing effective communication time. During tuning, we start from the tiling size of medium-grained partitioning (denoted as chunksize in Figure 10), meaning the tiling size equal m divided by the number of tensor parallelism, and then keep dividing by two until it equal to the GEMM tile size. Figure 10 shows the communication tile size does impact the overall performance. However, since there is no clear trend that one size always outperforming the other, autotuning is applied to select a best tiling factor.

Communication order among tiles. As discussed in Section 4.1, the communication order on the host side is aligned with tile coordinate swizzling, and chosen based on the network topology to minimize the overall delay. Intra-node NVLink interconnects apply direct communication with a ring order starting after the local rank. For example, given a local rank index 5 of 8-way tensor parallelism, the communication order of this rank is 6, 7, 0, 1, 2, 3, 4. Intra-node PCIe interconnects use ring-based communication to efficiently utilize PCIe bandwidth for single-node tensor parallelism. In multi-node tensor parallelism, for example, 16-way tensor parallelism, inter-node communication *potentially* can overlap with intra-node communication as well. Therefore, in the intra-node NVLink interconnects, inter-node direct communication is issued with local intra-node communication, and then after each communication tile received from inter-node communication, corresponding new intra-node communication would be issued. In the intra-node PCIe

interconnects, communication is much tricky, since some parts of PCIe interconnects are shared between inter-node and intra-node communication. In the evaluated PCIe cluster (the A100 PCIe cluster in Section 5), 4 GPUs and 1 NIC connect to one CPU core, and there are 2 CPU cores per node. In this cluster, inter-numa (still intra-node) communication and inter-node communication should not be scheduled at the same time for avoiding possible traffic. Therefore, inter-numa communication is issued first, and then intra-numa and inter-node communication is issued together.

4.4 GEMM Implementation and Auto-Tuning

Flux is generally applicable for almost all kinds of GEMM kernels. Considering GEMM performance is critical for overall performance, workload-balanced GEMM [35] is typically preferred on Ampere GPUs, and warp or thread block specialized GEMM [21] is preferred on Hopper GPUs. Also, since regular tiling of GEMM in Flux is not bond to the number of tensor parallelism, tiling sizes can be adjusted without impacting correctness. Flux is implemented using CUTLASS [21] to fully control GEMM tiling and corresponding prologue or epilogue fusion. Similarly to traditional GEMM libraries tuning and selecting GEMM kernels based on matrix shapes, data types, and GPU architecture, all prologues, epilogues, GEMM algorithms, and all tuning knobs, are written in templates, allowing us to autotune kernels by selecting proper template parameters.

5 Evaluation

Flux is implemented with CUTLASS 3.4.1 [21] and NVSHMEM 2.10.1 [33], with compiled with NVCC 11.8 for NVIDIA A100 GPUs and NVCC 12.2 for H800 GPUs. The results are evaluated with bfloat16 on three different clusters, 1) an A100 PCIe (80GB) cluster (denoted as A100 PCIe) with 8 GPUs per node, PCIe intra-node interconnects, and 2 100Gbs inter-node interconnects (4 GPUs and 1 NIC per CPU core), 2) an A100 SXM4 (80GB) cluster (denoted as A100 NVLink) with 8 GPUs per node, NVLink intra-node interconnects, and 4 200Gbs inter-node interconnects (2 GPUs sharing 1 200Gbs inter-node interconnect), and 3) an H800 SXM5 cluster (denoted as H800 NVLink) with 8 GPUs per node, NVLink intra-node interconnects, and 8 400Gbs inter-node interconnects (each GPU having its own dedicated 400Gbs inter-node interconnect to its corresponding GPUs on other nodes).

For the existing medium-grained overlapping method, we use TransformerEngine 1.4.0 [14] with UserBuffer, since the rest [12, 13] are not available[7] for A100 and H800 GPUs. TransformerEngine has multiple configurations for communication overlapping, and the reported numbers are from the best of all configurations. In operation-level evaluation, we evaluate GEMM with ReduceScatter, and AllGather patterns, and report both computation time and effective communication time, as well as overlap efficiency (defined in Section 2.3), while in model-level evaluation, we evaluate the entire model and only report overall time.

5.1 Operation-level Performance Evaluation

The GEMM dimensions for evaluation are selected from GPT-3 175B [2]. Therefore, (n, k) is determined as (49152, 12288) and (12288, 49152) in AllGather and ReduceScatter, respectively. Note we use (n, k) is the original shape before applying tensor parallelism. We evaluated GEMM for m from 1024 to 8192, simulating different workloads on training and prefill phases, and much smaller m as 64 and 512 for workloads on decoding phases.

Figure 11, 12, and 13 show performance results for ReduceScatter and AllGather overlapping. Given these evaluated sizes, Flux can deliver 1.20x to 3.25x speedups on A100 PCIe, 1.01x to 1.33x speedups on A100 NVLink, and 1.10x to 1.51x speedups on H800 NVLink over TransformerEngine. In terms of overlap efficiency, Flux can deliver 41% to 57% on A100 PCIe, 36% to 96% on A100 NVLink, and 37% to 93% on H800 NVLink, while TransformerEngine has -125% to 36% on A100 PCIe, -99% to 74% on A100 NVLink, and -40% to 80% on H800 NVLink. Note a negative overlap efficiency implies worse performance than the non-overlapping baseline.

Figure 14 shows performance comparison for much smaller m sizes. Given these evaluated sizes, Flux can deliver 1.45x to 3.21x speedups on A100 PCIe, 1.33x to 4.68x speedups on A100 NVLink, and a 0.95x slowdown to a 1.03 speedup on H800 NVLink over TransformerEngine. In terms of overlap efficiency, Flux delivers -2% to 41% on A100 PCIe, 14% to 88% on A100 NVLink, and -165% to -82% on H800 NVLink, while TransformerEngine has -213% to -36% on A100 PCIe, -325% to -49% on A100 NVLink, and -142% to -93% on H800 NVLink.

Figure 15 shows performance comparison for 16-way tensor parallelism on 16-GPU cluster (8 GPUs per node and two nodes), with (m, n, k) as (8192, 49152, 12288) and (8192, 12288, 49152) in AllGather and ReduceScatter. We only compare Flux with the PyTorch baseline, since TransformerEngine does not support multi-node overlapping. Flux

[7] [12] does not support well new GPUs, like A100 or H800, while [13] only supports TPUs.

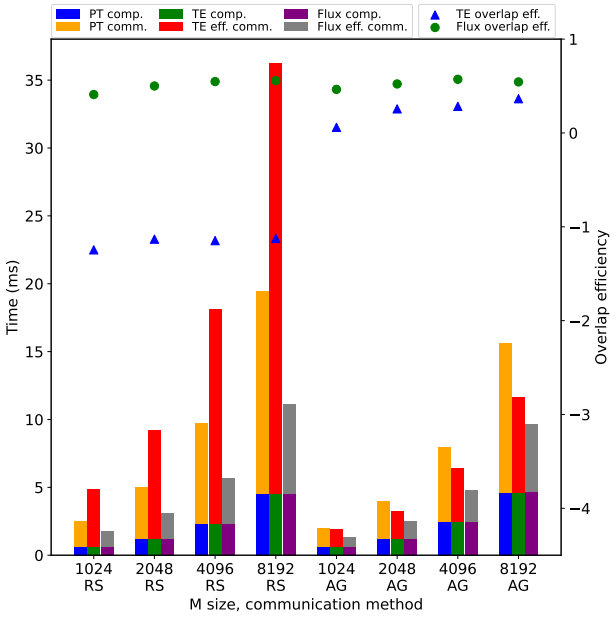

Figure 11: Performance results on an 8-A100 PCIe cluster

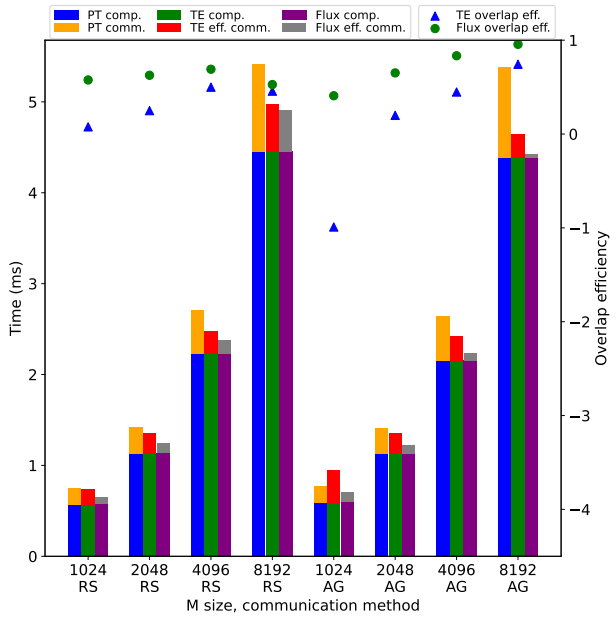

Figure 12: Performance results on an 8-A100 NVLink cluster

can deliver up to 1.32x speedups and 18% overlap efficiency on A100 PCIe, up to 1.57x speedups and 74% overlap efficiency on A100 NVLink, and up to 1.55x speedups and 56% overlap efficiency on H800 NVLink over PyTorch with fastest GEMM and NCCL.

5.2 Model-level Performance Evaluation

The evaluated models are GPT-3 175B and Llama-2 70B for both training and inference. In training, we use Megatron-LM core r0.4.0[8] [22] for GPT-3 175B and Megatron-LLaMA [36] for Llama-2 70B [37] on 128-GPU clusters with 2-way data, 8-way pipeline, and 8-way tensor parallelism. The entire training time, including gradient and optimizer

[8] commit 27cbe46

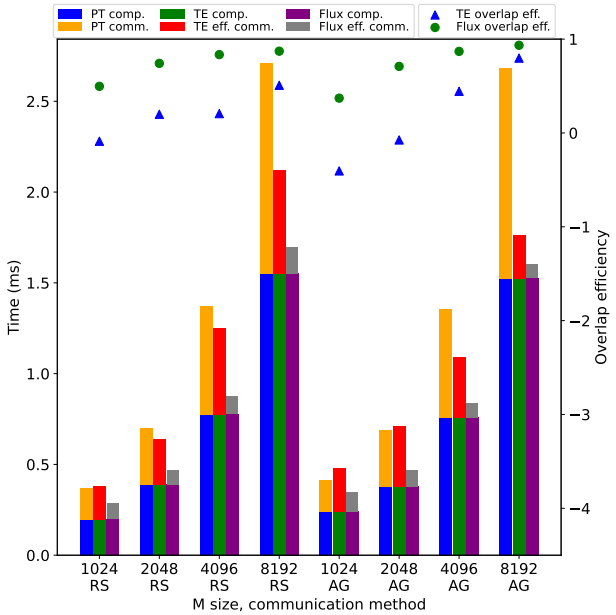

Figure 13: Performance results on an 8-H800 NVLink cluster

phases, are reported. In inference, we use vLLM 0.2.1 [23] for both models, with batch size as 8 and sequence length as 2048 in the prefill phase, and batch sizes as 64 or 512 in the decoding phase.

Figure 16 and 17 show performance results for training, prefill, and decoding phases. Flux can deliver up to 1.37x training, 2.06x prefill, and 1.69x decoding speedups on A100 PCIe, 1.04x training, 1.14x prefill, and 2.10x decoding speedups on A100 NVLink, and 1.05x training, 1.18x prefill, and 1.76x decoding speedups on H800 NVLink over TransformerEngine, while Flux can deliver up to 1.24x training, 1.46x prefill, and 1.28x speedups on A100 PCIe, 1.05x training, 1.45x prefill, and 1.30x decoding speedups on A100 NVLink, and 1.10x training, 1.66x prefill, and *no* decoding speedups on H800 NVLink over Megatron-LM and vLLM baselines.

6 Discussion

Small m sizes and decoding. When m is small (less than or equal to 1024), Flux outperforms TransformerEngine significantly by 1.03x to 4.68x speedups, except a 0.95x slowdown case of ReduceScatter with m as 64 on H800 NVLink. Figure 14 echoes our earlier statement that the existing methods could underutilize GPU compute power by splitting GEMM kernels. The effect becomes more apparent when the problem size getting small. The figure also shows that Flux could perform worse than the non-overlapping baseline in a few extremely small m cases, while TransformerEngine performs even much worse in all small cases. That is mainly because when m is extremely small, the GEMM kernels typically have fewer warps, making latency hiding less efficient. Moreover, when m as 64 on H800 GPU, after 8-way tensor parallelism, Flux ReduceScatter further makes TMA instruction less efficient by reducing TMA store size to 8 along m, causing the only data point worse than TransformerEngine. Similar effects can be observed from the decoding results in Figure 17, especially for the results with the batch size as 64. As the figure shown, Flux outperforms TransformerEngine by 1.21x to 2.10x speedups, but still has 5 cases slower than the non-overlapping vLLM baseline. The results of the batch size 512 are better than ones of the batch size 64, aligning with the above mentioned reason.

Overlap efficiency and speedups. The model-level performance is highly correlated to 2 factors, 1) tensor parallel communication portion shown in Figure 1, and 2) overlap efficiency. When the communication portions are about 8% to 11% on A100 NVLink training, the overall speedups of Flux are only 1.04x to 1.05x over Megatron-LM and only 1.01x to 1.04x over TransformerEngine, regardless of Flux having 63% average overlap efficiency on A100 NVLink cluster. For the workloads with much higher communication portions, such as 40% to 75% on A100 PCIe training and prefill, Flux can easily deliver 1.16x to 1.46x speedups over Megatron-LM, and 1.32x to 2.06x speedups over TransformerEngine. As the results of the operation-level evaluation shown, Flux can deliver 40%, 63% and 72% average overlap efficiency on A100 PCIe, A100 NVLink, and H800 NVLink clusters, respectively, while TransformerEngine only delivers -67%, -61%, and 20% average overlap efficiency, respectively. With that high overlap efficiency, Flux has a great potential to improve performance of tensor parallelism.

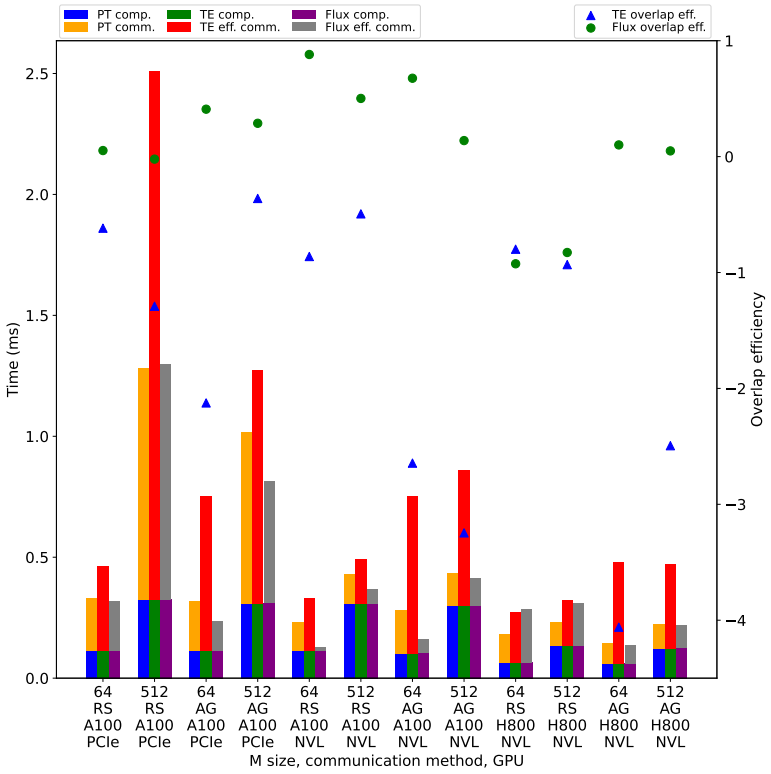

Figure 14: Performance results for small m sizes

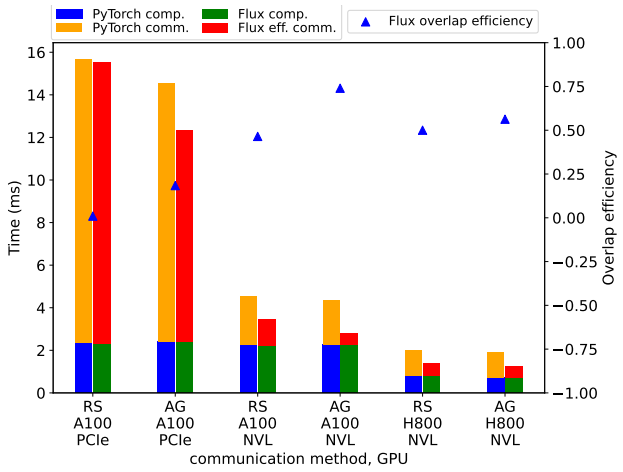

Figure 15: Performance results for 16-way tensor parallelism

High communication proportion. As just mentioned, the communication proportion is a key factor to take benefits from Flux. With high communication proportion on A100 PCIe and H800 NVLink clusters, Flux outperforms TransformerEngine by noticeable speedups, up to 3.25x on A100 PCIe and 1.51x on H800 NVLink, in the operation level. These two clusters have high communication proportion for different reasons, A100 PCIe due to slow interconnects and H800 NVLink due to fast computation. No matter the reason, Flux can still deliver high efficient communication overlapping, demonstrating its robustness, and the advantages fundamentally from the algorithm, optimization, and implementation sides. On the A100 PCIe cluster, we even observe that Flux sometimes can run faster than the non-overlapping communication only time, especially in large m sizes. Although we tend to use the best communication library as the baseline in the evaluation from the best of our knowledge, the widely used, standard GPU communication library, NCCL, can still underperform in some problem sizes. This also demonstrates the performance of Flux and its auto-tuning mechanism adapting the interconnection well.

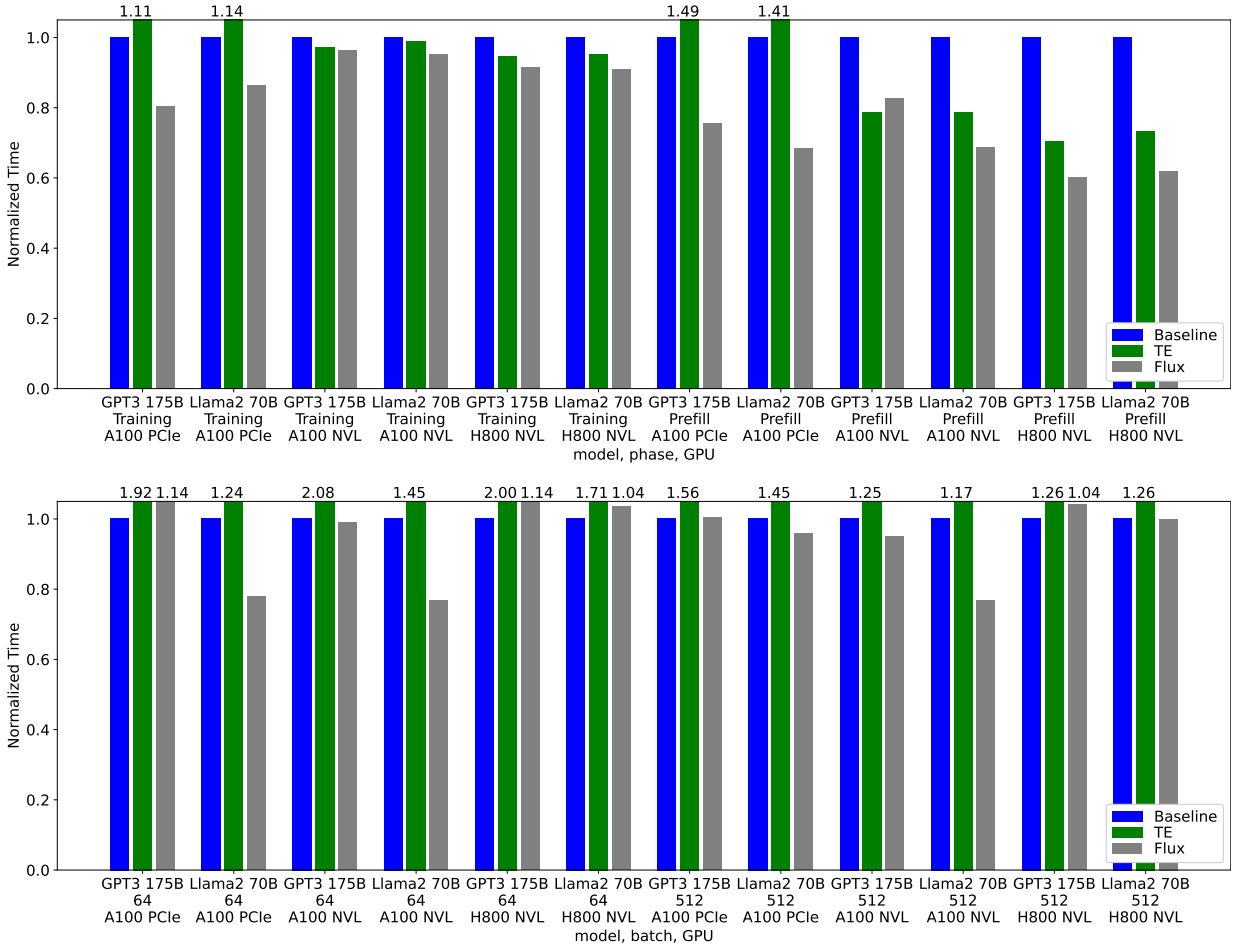

Figure 17: End-to-End results for decoding of GPT-3 175B and Llama-2 70B on various 8-GPU clusters.

7 Related Work

Communication overlapping techniques are widely applied on distributed systems for various applications, using various techniques [38, 39, 40, 41]. Flux uses kernel fusion and mainly targets at training and inference of large deep learning models.

With tensor parallelism for large deep learning models, prior works [12, 13, 14] decompose coarse-grained operations into a sequence of medium-grained ones, and carefully schedule them to overlap communication with computation for TPUs and GPUs. The work [26] applies the same overlap technique with [13] to efficient inference. The work [12] fuses communication with elementwise operations, overlapping with GEMM in a medium-grained fashion. In comparison, Flux applies much finer-grained decomposition for GPUs, fuses GEMM with communication, and supports both training and inference.

Pipeline parallelism [15, 16, 17, 42, 43, 44] is another common technique for large deep learning models. These work [15, 16, 17] potentially can overlap some pipeline communication with gradient reduction, reducing pipeline bubbles. Particularly, work [16, 17] combine tensor parallelism, pipeline parallelism, and data parallelism together for large deep learning model training. Flux can be applied in addition to further improve performance.

Accelerating collective communication [12, 18, 19, 45, 46] is another direction for improving network utilization. Flux is a communication overlapping solution and can work with accelerated collective communication. Communication compression techniques [20, 47, 48, 49, 50, 51, 52] for deep learning also improve network utilization by reducing communication sizes. Flux can be combined with the above methods.

ZeRO sharding techniques [20, 27, 28, 29, 30] partition weights and/or gradients onto multiple devices with data parallelism, and perform AllGather operation before computations. Those AllGather communications can be easily prefetched and overlapped with independent computation. Flux can be applied to activations, weights and gradients, thus can be coupled with the above techniques.

8 Conclusion

Communication overlapping techniques are crucial for running large deep learning models with tensor parallelism. Conventional overlapping techniques perform poorly on GPUs. The paper propose a novel technique, Flux, to resolve the issues. By over-decomposing communication with corresponding computation, and fusing them into a single large kernel, the proposed technique can significantly reduce the exposed communication time and effectively improve system FLOPS utilization regardless of training or inference.

9 Acknowledgements

We would like to express our sincere gratitude to Chengji Yao, Zhekun Zhang, Dongyang Wang and Yawei Wen for their assistance and guidance throughout the development of FLUX. Their expertise and insights were instrumental in overcoming the challenges encountered during this project. We also thank all our colleagues who contributed their time and knowledge, providing support and encouragement.

References

[1] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," 2020.

[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[3] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "Palm: Scaling language modeling with pathways," 2022.

[4] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, E. Zhang, R. Child, R. Y. Aminabadi, J. Bernauer, X. Song, M. Shoeybi, Y. He, M. Houston, S. Tiwary, and B. Catanzaro, "Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model," 2022.

[5] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation," 2021.

[6] M. Dehghani, J. Djolonga, B. Mustafa, P. Padlewski, J. Heek, J. Gilmer, A. Steiner, M. Caron, R. Geirhos, I. Alabdulmohsin, R. Jenatton, L. Beyer, M. Tschannen, A. Arnab, X. Wang, C. Riquelme, M. Minderer, J. Puigcerver, U. Evci, M. Kumar, S. van Steenkiste, G. F. Elsayed, A. Mahendran, F. Yu, A. Oliver, F. Huot, J. Bastings, M. P. Collier, A. Gritsenko, V. Birodkar, C. Vasconcelos, Y. Tay, T. Mensink, A. Kolesnikov, F. Pavetić, D. Tran, T. Kipf, M. Lučić, X. Zhai, D. Keysers, J. Harmsen, and N. Houlsby, "Scaling vision transformers to 22 billion parameters," 2023.

[7] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, "Robust speech recognition via large-scale weak supervision," 2022.

[8] Y. Zhang, D. S. Park, W. Han, J. Qin, A. Gulati, J. Shor, A. Jansen, Y. Xu, Y. Huang, S. Wang, Z. Zhou, B. Li, M. Ma, W. Chan, J. Yu, Y. Wang, L. Cao, K. C. Sim, B. Ramabhadran, T. N. Sainath, F. Beaufays, Z. Chen, Q. V. Le, C.-C. Chiu, R. Pang, and Y. Wu, "Bigssl: Exploring the frontier of large-scale semi-supervised learning for automatic speech recognition," *IEEE Journal of Selected Topics in Signal Processing*, vol. 16, p. 1519–1532, Oct. 2022.

[9] X. Liu, D. McDuff, G. Kovacs, I. Galatzer-Levy, J. Sunshine, J. Zhan, M.-Z. Poh, S. Liao, P. D. Achille, and S. Patel, "Large language models are few-shot health learners," 2023.

[10] S. Wu, O. Irsoy, S. Lu, V. Dabravolski, M. Dredze, S. Gehrmann, P. Kambadur, D. Rosenberg, and G. Mann, "Bloomberggpt: A large language model for finance," 2023.

[11] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray,

N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[12] A. Jangda, J. Huang, G. Liu, A. H. N. Sabet, S. Maleki, Y. Miao, M. Musuvathi, T. Mytkowicz, and O. Saarikivi, "Breaking the computation and communication abstraction barrier in distributed machine learning workloads," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 402–416, 2022.

[13] S. Wang, J. Wei, A. Sabne, A. Davis, B. Ilbeyi, B. Hechtman, D. Chen, K. S. Murthy, M. Maggioni, Q. Zhang, *et al.*, "Overlap communication with dependent computation via decomposition in large deep learning models," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pp. 93–106, 2023.

[14] NVIDIA, "TransformerEngine." `https://github.com/NVIDIA/TransformerEngine`, 2022.

[15] J. Lamy-Poirier, "Breadth-first pipeline parallelism," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[16] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, *et al.*, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021.

[17] F. Li, S. Zhao, Y. Qing, X. Chen, X. Guan, S. Wang, G. Zhang, and H. Cui, "Fold3d: Rethinking and parallelizing computational and communicational tasks in the training of large dnn models," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 5, pp. 1432–1449, 2023.

[18] A. Shah, V. Chidambaram, M. Cowan, S. Maleki, M. Musuvathi, T. Mytkowicz, J. Nelson, O. Saarikivi, and R. Singh, "{TACCL}: Guiding collective algorithm synthesis using communication sketches," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 593–612, 2023.

[19] M. Cowan, S. Maleki, M. Musuvathi, O. Saarikivi, and Y. Xiong, "Mscclang: Microsoft collective communication language," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 502–514, 2023.

[20] G. Wang, H. Qin, S. A. Jacobs, C. Holmes, S. Rajbhandari, O. Ruwase, F. Yan, L. Yang, and Y. He, "Zero++: Extremely efficient collective communication for giant model training," *arXiv preprint arXiv:2306.10209*, 2023.

[21] V. Thakkar, P. Ramani, C. Cecka, A. Shivam, H. Lu, E. Yan, J. Kosaian, M. Hoemmen, H. Wu, A. Kerr, M. Nicely, D. Merrill, D. Blasig, F. Qiao, P. Majcher, P. Springer, M. Hohnerbach, J. Wang, and M. Gupta, "CUTLASS." https://github.com/NVIDIA/cutlass, 2024.

[22] NVIDIA, "Megatron-LM." `https://github.com/NVIDIA/Megatron-LM`, 2021.

[23] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[24] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[25] V. A. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro, "Reducing activation recomputation in large transformer models," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[26] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[27] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16, IEEE, 2020.

[28] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "{ZeRO-Offload}: Democratizing {Billion-Scale} model training," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 551–564, 2021.

[29] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning," in *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, IEEE, 2021.

[30] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer, *et al.*, "Pytorch fsdp: experiences on scaling fully sharded data parallel," *arXiv preprint arXiv:2304.11277*, 2023.

[31] Z. Jiang, H. Lin, Y. Zhong, Q. Huang, Y. Chen, Z. Zhang, Y. Peng, X. Li, C. Xie, S. Nong, *et al.*, "Megascale: Scaling large language model training to more than 10,000 gpus," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24')*, 2024.

[32] NVIDIA, "NCCL." https://github.com/NVIDIA/nccl, 2016.

[33] NVIDIA, "NVSHMEM." `https://developer.nvidia.com/nvshmem`, 2020.

[34] Z. Chen, A. Kerr, R. Cai, J. Kosaian, H. Wu, Y. Ding, and Y. Xie, "EVT: Accelerating deep learning training with epilogue visitor tree," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2024 (in press).

[35] M. Osama, D. Merrill, C. Cecka, M. Garland, and J. D. Owens, "Stream-k: Work-centric parallel decomposition for dense matrix-matrix multiplication on the gpu," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 429–431, 2023.

[36] Alibaba, "Megatron-LLaMA." `https://github.com/alibaba/Megatron-LLaMA`, 2023.

[37] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[38] A. Danalis, K.-Y. Kim, L. Pollock, and M. Swany, "Transformations to parallel codes for communication-computation overlap," in *SC'05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, pp. 58–58, IEEE, 2005.

[39] A. Danalis, L. Pollock, M. Swany, and J. Cavazos, "Mpi-aware compiler optimizations for improving communication-computation overlap," in *Proceedings of the 23rd international conference on Supercomputing*, pp. 316–325, 2009.

[40] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero, "Overlapping communication and computation by using a hybrid mpi/smpss approach," in *Proceedings of the 24th acm International Conference on Supercomputing*, pp. 5–16, 2010.

[41] T. Gysi, J. Bär, and T. Hoefler, "dcuda: hardware supported overlap of computation and communication," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 609–620, IEEE, 2016.

[42] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

[43] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 1–15, 2019.

[44] B. Yang, J. Zhang, J. Li, C. Ré, C. Aberger, and C. De Sa, "Pipemare: Asynchronous pipeline parallel dnn training," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 269–296, 2021.

[45] S. Rashidi, M. Denton, S. Sridharan, S. Srinivasan, A. Suresh, J. Nie, and T. Krishna, "Enabling compute-communication overlap in distributed deep learning training platforms," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 540–553, IEEE, 2021.

[46] Microsoft, "MSCCL++." `https://github.com/microsoft/mscclpp`, 2023.

[47] N. Dryden, T. Moon, S. A. Jacobs, and B. Van Essen, "Communication quantization for data-parallel training of deep neural networks," in *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, pp. 1–8, IEEE, 2016.

[48] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "Qsgd: Communication-efficient sgd via gradient quantization and encoding," *Advances in neural information processing systems*, vol. 30, 2017.

[49] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "Terngrad: Ternary gradients to reduce communication in distributed deep learning," *Advances in neural information processing systems*, vol. 30, 2017.

[50] H. Xu, K. Kostopoulou, A. Dutta, X. Li, A. Ntoulas, and P. Kalnis, "Deepreduce: A sparse-tensor communication framework for federated deep learning," *Advances in Neural Information Processing Systems*, vol. 34, pp. 21150–21163, 2021.

[51] H. Xu, C.-Y. Ho, A. M. Abdelmoniem, A. Dutta, E. H. Bergou, K. Karatsenidis, M. Canini, and P. Kalnis, "Grace: A compressed communication framework for distributed machine learning," in *2021 IEEE 41st international conference on distributed computing systems (ICDCS)*, pp. 561–572, IEEE, 2021.

[52] J. Fei, C.-Y. Ho, A. N. Sahu, M. Canini, and A. Sapio, "Efficient sparse collective communication and its application to accelerate distributed deep learning," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pp. 676–691, 2021.