Multi-GPU for Piranha, a Multiparty Computation Framework

Yibai Meng
University of California, Berkeley
mengyibai@berkeley.edu

Shuxian Wang
University of California, Berkeley
wsx@berkeley.edu

Alex Schedel
University of California, Berkeley
alexschedel@berkeley.edu

1 ABSTRACT

Secure multi-party computation (MPC) is an essential tool for privacy-preserving machine learning. Piranha [10] is a opensource GPU-based platform developed by SkyLab that provides a generic interface for implementing secure multi-party computation algorithms, while leveraging the acceleration GPU provides.

Currently, Piranha can only use one GPU per computation party. This limits it's ability to work with large models. Therefore, for our CS262A project, we explored distributing the computation of a single party among multiple GPU, to increase speed and enhance scalability. This will allow for effective scalabilty in regard to both time and memory for the system.

We experimented with various form of model parallelism, including pipeline parallelism and tensor parallelism. We are able to decrease the per GPU memory requirements to enable training models with much larger batch sizes. We also realized some training speedup in limited circumstances. We also discovered a important optimization that greatly increases Piranha's performance, albeit only marginally related to multi-GPU parallelism.

Our code is at https://github.com/YibaiMeng/piranha, feel free to have a look.

2 INTRODUCTION AND BACKGROUND

2.1 Multi-party Communication

Multi party computation (MPC) is a secure computing paradigm aimed at allowing multiple parties not only to communicate information securely, but to jointly compute over that information. A key distinction between MPC and conventional cryptographic algorithms is that rather than securing a system from outside attackers or adversaries, MPC protects the data of cooperating parties from each other. This means that computation is able to be done in a completely trust free environment, as no party is able to access any information other than their own.

MPC is facilitated by a combination of cryptography and distributed computing protocols. First, data is divided into several individual *shares*, each of which represent some amount of data held by a single party. This data is then securely encrypted by each party to ensure that others cannot directly read their data. Through use of a secret sharing algorithm, these shares contain information unintelligible to each individual party, only yielding the secret once a sufficient number of shares are combined. This allows the parties to jointly compute on encrypted information without revealing any of the underlying data.

MPC can be used in a variety of applications, such as secure voting systems, distributed storage systems, and, as in the case of Piranha, secure communication protocols for use in machine learning systems. However, the cryptographic and distributed computing protocols which underpin the paradigm have a major drawback, that being that they are often quite computationally expensive, making it difficult to justify their use in large scale machine learning frameworks without optimization.

2.2 Piranha

As noted above, writing efficient secure MPC algorithm requires extensive knowledge. This is where Piranha, a general purpose, modular platform for accelerating secret sharing MPC protocols using NVIDIA GPU's, enters the picture. Piranha aims to leverage the benefits of GPU parallelization without requiring domain specific CUDA knowledge on the part of the programmer. The Piranha system itself is split into a three level of abstractions, shown in Figure 1.

(1) **The Device Level**. This layer accelerates secret-sharing protocols by providing additional integer kernels not found in many present GPU libraries. Additionally, this layer is responsible for abstracting away much of the domain specific knowledge of GPU programming from the user as well as data management. Data is managed on a *DeviceData buffer* and resides only on the GPU. This is a key point of speed up and parallelization under the current system.
(2) **The Modular Protocol Level**. This layer implements all the cryptographic primitives required by the MPC protocols, additionally it allows developers to maximize limited GPU memory with in-place computation and iterator-based support for non-standard memory access patterns. Piranha currently contains support for 2, 3, and 4 party MPC (all sharing the same interface), and is designed to be extensible with other MPC protocols as well.

(3) **The Application Level**. This layer allows applications to remain agnostic to the underlying MPC protocols they use. Common layers in neural network is implemented, with the primitives exposed from the protocol layer. The end user may directly use those layers to compose a working neural network.

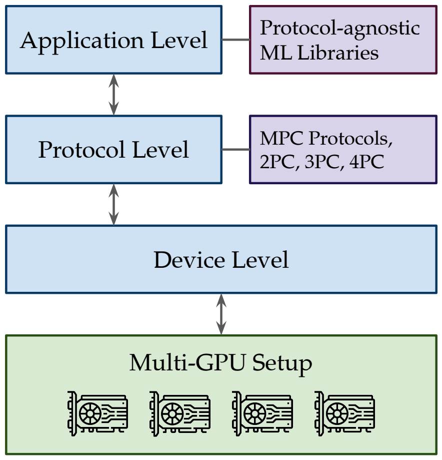


Figure 1: The architecture of Piranha. Blue indicates core layers, purple indicates modular components, and green indicates the underlying hardware

2.3 Multi-GPU Parallelism

People have been using multiple GPUs to train neural networks and do computation in general since their inception. In general, people would like to achieve two purpose when using multiple GPUS:

- Fit very large models onto limited hardware
- Significantly speed up training

Based on the nature of these methods, they can be broadly split into to categories:

(1) **Data Parallel**: The model is being replicated multiple times on multiple GPUs, while the input data is split. Training or inference happens concurrently for the split data, and updates are communicated and synchronized if necessary.
(2) **Model Parallel**: The model is split among multiple GPUs, and data is passed through each layer and goes from GPU to GPU. Pipelining is often used to increase performance.

These methods are often used among side each other. Sometimes, the classification of a parallelization method depends on the circumstances. For example, for tensor parallelism, if model parameters and optimizer weights are split among the devices, it could be classified as model parallelism, whereas if input matrices are the ones split, it could become data parallelism.

2.4 Our Approach

In this report, we discuss our experiments bringing multi-GPU to Piranha. Thanks to the three levels of abstractions provided by Piranha, we do not need to alter the crytographic protocols in the Modular Protocol level, lowering our engineering effort. We implemented model parallelism, especially pipelined model parallelism in piranha. We also implemented tensor parallelism for the underlaying `gemm` operations. During our experimentation, we also uncovered an significant inefficiency in the original Piranha code.

The rest of the report is organized as follows. Section 3 discusses the concept of model parallelism, its implementation, as well as the notable implementation details in the context of Piranha. Section 4 discusses tensor parallelism and our implementation. Section 5 discusses the inefficient way Piranha is conducting inter-party communication, and how we remedies it. Section 6 evaluates our results. Section 8 and Section 7 summarizes possible future works and prior related work on this subject.

3 MODEL PARALLELISM

As mentioned in Section 2.3, model parallelism partition on the entire model into groups of consecutive layers, and assign each GPU to a group of layers. Collectively, all GPU's serve a model by each constituting its group of layers, and the resource requirement on each device becomes only a fraction of the entire model.

3.1 Sequential Model Parallelism

The most straightforward way to implement model parallelism is to assign layers to dedicated GPU's, with the forward and back-propagation of the layers calculated in sequence. After all the layers on a GPU finish execution, we transmit the data to another GPU, and start the process anew. This way, each party only needs to hold a portion of the entire network. The following algorithm illustrates this idea:

```
for all iteration $i$ do
    $B \leftarrow$ input batch
    for all GPU $g$ do
        Transmit $B$ onto $g$
        for all Layer $L$ on GPU $g$ do
            $B \leftarrow$ forward direction of $L$ with input $B$
    $G \leftarrow$ loss of $B$
    for all GPU $g$ in reversed order do
        Transmit $G$ onto $g$
        for all Layer $L$ on GPU $g$ in reversed order do
            $G \leftarrow$ back-propagation of $L$ with gradient $G$
            Apply updates to $L$ with gradient $G$
```

Unfortunately, this method of model parallelism will be slower than using a single GPU. When one layer is active, other layers will be waiting idly. However, sequential model parallelism can reduce the memory requirement for individual GPU's by a factor of N. As GPU onboard memory is often a limiting factor during the training of large models, this simple method is widely used.

3.2 Pipeline Parallelism

We want the memory saving of model parallelism, but we also want to avoid the incurred performance penalties. One way to achieve this goal is to use pipeline parallelism. We do this by splitting the incoming minibatch into *micro-batches*, creating a pipeline, which allows different GPUs to concurrently participate in the computation process.

Same as sequential model parallelism, the layers of the model is partitioned onto various GPUs. We call the collection of sequential layers on each GPU a *pipeline group*. Different from sequential model parallelism, for forward and backward, each pipeline launches a asynchronous thread. Each thread goes through the the microbatches of the minibatch, and for each microbatch iterate over all the layers within this pipeline group, calling forward and backward. For forward, all microbatches are independent from each other, therefore the final resulting activation is identical to that of ordinary calculation. The activation will then be sent asynchronously to the next pipeline group, while the sending pipeline group is already at the next microbatch.

Back propagation is a little different as not all microbatches are completely independent of each other. The gradient of the loss relative to each microbatch input is independent. It is thus treated the same as forward activation, sent to the previous pipeline group asynchronously. The gradient of the loss relative to the weight, on the other hand, depends on all the microbatches. Due to the chain rule, each microbatch only provides a portion of the gradient, and the final results are their sum. Therefore after independently back-propagating each microbatch, we synchronize and the update the weights. See a pseudocode representation of a thread on GPU g below:

```
for all microbatch $b$ in minibatch $B$ do
    Wait for activation from microbatch $b$ on the previous pipeline group to arrive.
    for all Layer $L$ on GPU $g$ do
        $b \leftarrow$ forward direction of $L$ with input $b$
    Async send the output onto the next pipeline group.
Synchronize all threads
Calculate loss gradient
for all microbatch $b$ in minibatch $B$ do
    Wait for gradient from microbatch $b$ on the next pipeline group to arrive
    for all Layer $L$ on GPU $g$ in reversed order do
        $G \leftarrow$ back-propagation of $L$ with gradient $G$
    Async send the output onto the previous pipeline group.
Apply updates to $L$ with gradient $G$
```

Because now transmission of activation and gradient between layers are done *asynchronously*, microbatches in one GPU can be running concurrently with another microbatch in another GPU. This is shown in Figure 2. Compared to sequential model parallelism, in pipelined parallelism, GPUs are idle less of the time. Due to the dependency relationship between the microbatches, there are still bubbles before and after the forward and backward pass.

3.3 Partition Selection

As mentioned in Section 3, for each party, we need to partition the model / layers onto different GPU's. Therefore, for each specific model, we need to chose where to draw the boundary for each partition. This choice of partition will significantly affect the effectiveness of pipeline parallelism.

It is not feasible to find an optimal partition simply by trial and error. For a network with L layers and a setup with P GPU's, the number of possible partitions are on the order of L^{P-1}. Therefore, we need some theoretical insight into the execution time for each partition selection to make an informed decision.

This is not as trivial a task as it may at first glance appear. As seen in Figure 2 and Figure 6, each pipeline group consists of different layers, with vastly different forward inference and backward propagation times. Each microbatch is also dependent on the results of the corresponding microbatch from the previous group. Therefore, we need to take the interdependency, varied memory use, and computation time all into account during partitioning, to ensure that minimal time is spent on waiting for previous pipeline groups to complete.

We model the per iteration time as follows: For each layer l, the forward time for each microbatch on a single GPU is f_l, and the backpropagation gradient calculation time is g_l. The gradient update time is very small compared with gradient calculation, so we ignore it in our modeling. Let the number of pipeline stages / number of GPU's be P. Let the layers assigned to each pipeline group i be $L_1^i, L_1^i, \ldots, L_{P_i}^i$, where each pipeline group has P_i layers. Each minibatch is split into M microbatches. Let F_i be the time it takes for pipeline group i to complete the forward operation, starting from the beginning of the *whole iteration*: that is, starting from when the first pipeline group begins its forward calculation. Similarly, let G_i be the runtime of the backpropagation gradient calculation, starting from the beginning of the gradient calculation.

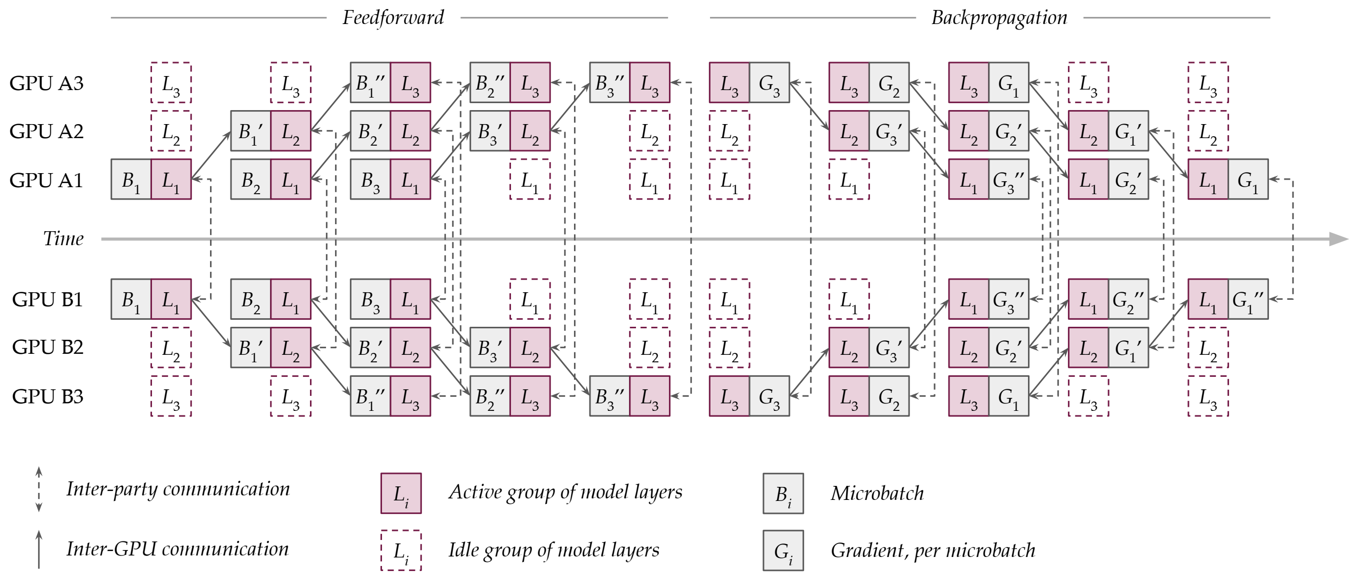


Figure 2: The pipeline processing with a 2-party protocol along time. Here we partition the model into three groups of layers L_1, L_2, and L_3. And to depict bubbles, we also partition the input batch into three microbatches B_1, B_2, and B_3.

With the above formulation, the problem becomes minimizing the maximum $F_i + G_i$ over all pipeline groups:

$$\min_{L^1,L^2,\ldots,L^P} \{\max_{i=1}^{P} F_i + \max_{i=1}^{P} G_i\} \tag{1}$$

Each microbatch might need to spend some time waiting for its input from the previous or subsequent pipeline group to become ready. We use B_m to represent the time spent waiting for the input to layer L_1^i of microbatch m to be ready for pipeline group i during forwarding (we use the letter B because it represents the "Bubble" in the calculation). For backward, we similarly define $\bar{B}_m$.

We can now express F with B and f with the minimal gradient update time safely ignored:

$$F_i = \sum_{m=1}^{M} B_m^i + M \sum_{l \in L^i} f_l \tag{2}$$

We have a recursive relationship for the bubble:

$$B_m^i = \max\{0, m \sum_{l \in L^{i-1}} f_l + \sum_{j=1}^{m} B_j^{i-1} - (m-1) \sum_{l \in L^i} f_l - \sum_{j=1}^{m-1} B_j^i\} \tag{3}$$

and

$$B_1^1 = 0 \tag{4}$$

The calculation of the backward time G is similar:

$$G_i = \sum_{m=1}^{M} \bar{B}_m^i + M \sum_{l \in L^i} g_l \tag{5}$$

where

$$\bar{B}_m^i = \max\{0, m \sum_{l \in L^{i+1}} g_l + \sum_{j=1}^{m} \bar{B}_j^{i+1} - (m-1) \sum_{l \in L^i} g_l - \sum_{j=1}^{m-1} \bar{B}_j^i\} \tag{6}$$

and

$$\bar{B}_1^P = 0 \tag{7}$$

With the above relations, we can calculate the time per training iteration for a given partition. In our implementation, we do "test training" of the network on a single GPU for a few iterations and record the timing of the forward and backward steps for each layer. We then search through all the possible partitions, given the number of GPU's / pipeline stages and the number of microbatches per minibatch. We then use the fastest possible partition.

Doing preliminary test training is necessary, as real life neural networks can have vastly different runtimes for each individual layer. In the context of MPC, this heterogeneity is even more extreme. For example, counter-intuitively, the forward calculation of the ReLU layer takes up the bulk of the time, as the underlying algorithm requires significant coordination between parties.

The partitioning can also be optimized for memory usage. This is easily solved using dynamic programming. However, this might lead to the pipelined algorithm showing no speedup, as layers with little memory usage such as ReLU may incur significant inter-party communication and runtime costs. It would be better to consider both when selecting the partitions.

3.4 Implementation Details

3.4.1 Inter-GPU communication. In the view of the controlling machine, or *host*, the GPU's are attached PCIe peripheral devices. By default, transfer between host and devices goes through the PCIe channel, so inter-device communication actually requires the host to relay the data (DMA is used for most occasions, so fortunately no CPU cycles are wasted). For newer models of NVIDIA GPU's, however, it is possible to add separate physical connections between GPU's, called NVLinks, and configure the communication to go through the GPU's directly, bypassing the host completely.

CUDA provides a inter-GPU memory transfer primitive `cudaMemcpypeer` and `cudaMemcpypeerAsync`. CUDA can automatically detect the capabilities and topology of devices, using NVLink when possible, defaulting back to PCIe when NVLink is not available. More recent versions of CUDA also provide Unified Virtual Addressing (UVA), allowing addresses on hosts and different GPU's to share a single memory space, storing the memory location information into the pointer itself. When UVA is enabled, we can use the ordinary `cudaMemcpy` to transfer data between devices. For our purposes of modifying Piranha, this approach has significant benefit: the current codebase heavily utilizes `thrust` and `cutlass`, and these libraries do not contain facilities for specifying which device the code is executing on. Using UVA allows us to lessen the code change needed to support inter-device communication, reducing the engineering effort.

3.4.2 Inter-GPU Synchronization. Synchronization is very important when dealing with multiple independent flows of execution. These flows include multiple parties, multiple host threads for each pipeline stage, and one or more devices for each pipeline stage. We want to make sure the dependent input is ready before commencing further microbatches. However, we also want to fully utilize the GPU's computational resources. CUDA GPU's have dedicated copy engines for memory transfer, independent of computation by the streaming multiprocessors. That is, we want the memory transfer and computation to happen concurrently.

As mentioned before, each party runs on a host with multiple GPU's and each pipeline group is controlled by a host thread. For each microbatch, the operators are enqueued onto the GPU, and executed synchronously relative to the device. After each microbatch is finished, the host thread synchronizes with the device, and invokes `cudaMemcpypeerAsync` as a *different stream*. This is necessary, otherwise the memory transfer would block the execution of the next batch.

The next pipeline group needs to be notified when the memory transfer is complete. Simply synchronizing on the memory transfer stream *would not work*, as this would require the command be invoked *after the start* of the memory transfer, the precise timing of which would be difficult for another thread to know. Instead, we use `cudaLaunchHostFunc`, to enqueue a host function into the CUDA stream to be executed after the completion of the `memcpy`. The host function will then `notify` a mutex-protected conditional variable in the following pipeline group, starting its microbatch. A counter in used to differentiate the `memcpy` of different microbatches.

We also fixed two subtle issues with the original code regarding synchronization and parallelism. First is the use of `cudaThreadSynchronize`. Despite its name, this primitive synchronizes *all* the operations on the device. This deprecated function is scattered throughout the GPU kernels in the codebase, sometimes causing the memory copy to be sequential regarding the computation. As all kernel executions on the same stream are sequential, device level synchronization is not needed.

Another issue is the "Default stream" synchronization behavior[7]. By default, the default stream would exhibit "legacy behavior". This would allow it to block relative to other non-blocking streams. We changed the setting to "per-thread" behavior, making different streams truly independent of each other.

4 TENSOR PARALLELISM

Tensor parallelism is distributing the work of primitive tensor operations, for example matrix multiplication, over multiple GPU's. Many operations are easily parallelized, and in fact many operations even have multiple ways to divide up the work. For matrix multiplication $AB = C$, we have three ways of partitioning the work:

(1) We can divide along the first dimension of A, with
$$\begin{bmatrix} A_1 & A_2 \end{bmatrix} B = \begin{bmatrix} A_1 B & A_2 B \end{bmatrix}.$$
(2) Due to the symmetry of the problem, we can also divide along the second dimension of B, with
$$A \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} = \begin{bmatrix} AB_1 \\ AB_2 \end{bmatrix}.$$
(3) And we can also divide by the second dimension of A and the first dimension of B simultaneously, having
$$\begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \begin{bmatrix} B_1 & B_2 \end{bmatrix} = A_1 B_1 + A_2 B_2.$$

Additionally, all the division schemes above can also be combined and mixed recursively.

The main challenge in achieving efficient multi-GPU coordination is cutting down the cost incurred by inter-GPU communication. All the aforementioned schemes require duplicating matrices or sending parts of matrices across different devices before and after the operations. Ideally, the memory copy of these operations should involve a continuous range for minimum communication cost, while the part of memory being transmitted depends on whether the matrices involved are stored in column-major or row-major order. For example, dividing A along the row dimension if A itself is written in row-major order requires copying scattered memory between devices. Dividing and copying a row-major matrix B along the column-dimension, however, only requires a single continuous memory copy command. Accounting for how the matrix is stored in memory, we can choose the division strategy based on the memory layout scheme of A and B. Specifically, we have the following table for choosing the optimal strategy:

A is column major	B is column major	Strategy
T	T	(1)
T	F	(1) or (2)
F	T	(2)
F	F	(3)

5 INTER-PARTY COMMUNICATION OPTIMIZATION

As we can see from Fig 3, a significant portion of Piranha's runtime is spent on sending and receiving data from other parties, making it a priority for us to optimize. Piranha does the following to transmit and receive data across parties:

```
func transmit(size) {
   hostBuffer = malloc(size);
   cudaMemcpyDeviceToHost(deviceBuffer,
       hostBuffer);
   socketSend(hostbuffer);
   free(hostBuffer);
}
func recv(size) {
   hostBuffer = malloc(size);
   socketRecv(hostbuffer);
   cudaMemcpyHostToDevice(hostBuffer,
       deviceBuffer);
   free(hostBuffer);
}
```

This implementation has two inefficiencies: first, `malloc` is frequently called to temporarily allocate a host size buffer which are then immediately reclaimed. In our tests, the median size of a transmission is around 1 MiB and thousands of transfers are made every iteration. Since the buffers are used for transitory storage anyways and data are transmitted to each socket sequentially, we can pre-allocate a shared buffer, and use that instead of allocating every time `transmit` and `recv` are called.

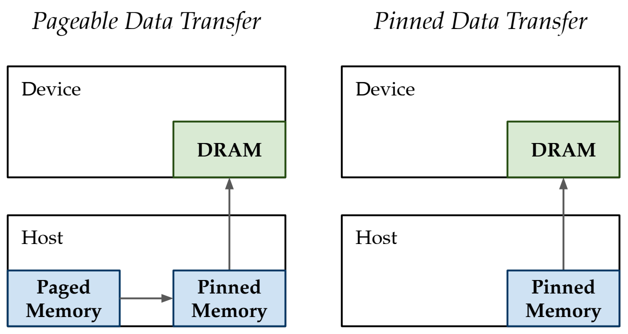


Figure 3: A comparison for host to device data transfer on GPU, between pagable memory and pinned memory

Another inefficiency is the use of *pageable memory* during host to device transfers. All user space memory is paged memory. Virtual memory, specific to each process, is translated to physical memory on the fly during memory access. Each page may be moved anywhere in physical memory at any time, even to disk. This causes no issue to the host, as the address translation happens under the hood by the MMU. Even if the page is moved to disk, a page fault would be raised and the page promptly be copied back to memory.

However, this causes issues when direct memory access (DMA) is involved. CUDA uses DMA to transfer memory between host and device. Considering the fact that pageable data may change its physical location at any time, and that the DMA controller is oblivious to activity on the CPU, it is not advisable to conduct a DMA transfer on pageable memory.

To solve this issue, CUDA uses *pinned memory* (also called *page-locked* memory). Pinned memory is memory that cannot be paged in or out. That is, it cannot be temporarily moved to disk. With this constraint, it is safe for the CUDA DMA driver to assume that the memory location will remain unchanged throughout the memory transfer process. For each memory transfer, CUDA first allocates a temporary pinned host array, copies the host data to the pinned array, and then transfers the data from the pinned array to device memory. This process is shown in Figure 3[3].

Given CUDA's memcpy procedure, we can use *pinned memory* to speedup our procedure. We modified our host buffer to use pinned memory, and pre-allocate the buffer at once during initialization. We also modified the transfer logic to do host-device transfers and socket send receives iteratively, so that we do not need to allocate our buffer to be as large as the largest message, which could go into the GiB rage given the batch-size and network structure. We have seen significant speedup from this relatively simple change.

6 RESULTS

Apart from the experiments for tensor parallelism, we conducted our evaluations on a 8-GPU server, with NVIDIA RTX A5000s, and 256 GiB of host memory. Due to our limited GPU resources, all the multi-party computation was conducted on the same machine with inter-party communication going over localhost. All the measurements do not consider the time for initialization, loading of training data and labels, as well as evaluation on the test set. GPU resources are quite expensive, therefore due to the limited time we had access to the machine, we did not conduct end-to-end training, instead opting to benchmark the behavior over 5 iterations. Our metric of success here would be faster training time, and lower memory consumption per GPU.

6.1 Inter Party Communication

As mentioned in Section 5, we modified Piranha to use pinned memory during interparty communication. As shown in Figure 4, using pre-allocated pinned memory has a significant effect on larger neural networks like VGG. For smaller networks the effects are more limited, but still noticeable. We can see that computation time did not change in a significant way, as expected.

The method of mallocing and freeing pinned memory during each send and receive is not very efficient. This is expected, as pinned memory allocation takes more effort than a simple malloc: pinned memory allocation need to set a whole page as pinned, while malloc does not care about page boundaries.

There's also a question of the amount of memory to pre-allocate, or the cache size. A larger cache allow the procedure to conclude in one iteration, calling memcpy and socket send / recv only once. Too small of a cache would result in too many iterations. However, it is not the case that larger caches always result in better outcomes. As we can see in Fig 5, a "sweet spot" is around 5 MiB. As TCP packets are only tens of kilobytes, and the host network throughput is significantly lower than that of the GPU memory bus, making the cache larger is not very meaningful. For all our experiments on parallelism, we use pre-allocated pinned memory as a buffer, and set the cache size to 5 MiB.

6.2 Model Parallelism

Theoretically, pipelined model parallelism could show a significant increase in training and inference speed. Based on the per layer execution times of the baseline single GPU two party training for a 4-GPU pipeline and 2 microbatches per pipeline, the results in Section 3.3 indicate that we can see a 50% speedup (splitting at the second, seventh and eighteenth layer). However, our experimental results are somewhat underwhelming, as seen in Table 1. Here we can see

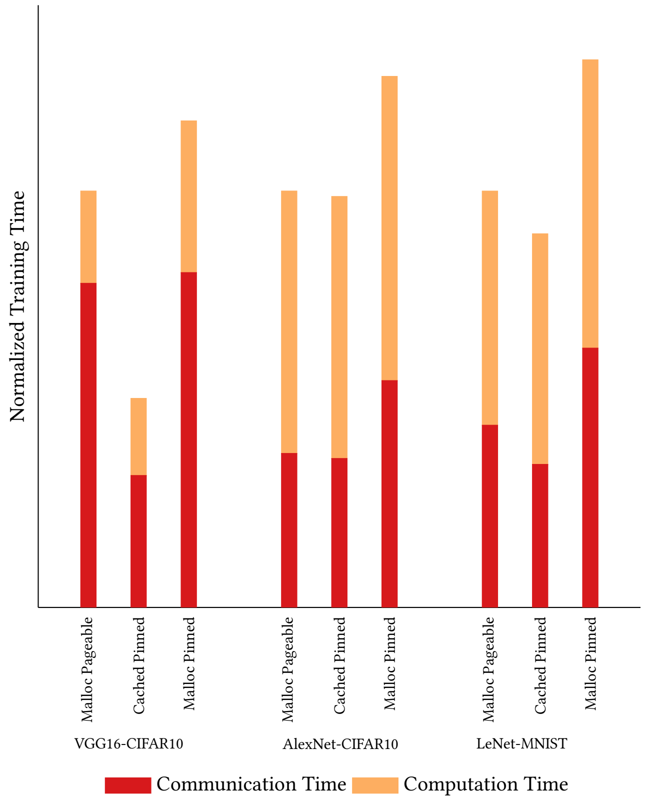


Figure 4: Comparison of host buffer and allocation method used during inter-party communication. Timing are normalized relative to the baseline (malloc pageable). All uses two party protocol (SecureML), single GPU with batch size 128.

some speedup for pipeline model parallel, for big network like VGG16. For smaller networks like AlexNet and LeNet, pipelining in fact makes it slower.

It's important to note that the baseline is a modified version of the original Piranha, only changing the inter-party communication buffer to pinned memory, with everything else remaining unchanged. For VGG, all the pipelined modes outperform the original version.

We have some thoughts on why this might be the case. Fig 6 contains side-by-side comparison of an example of non-pipelined and pipelined training. We can see that the pipelining is successful, as work on different GPUs are executing concurrently. If we zoom in, we can see GPU computation kernels executing alongside each other. However, for the pipelined run, the execution times of the subgroups are noticeably longer than the corresponding groups in the sequential run. We compared the per GPU timing information, as shown in Table 2. Although both the computation and

Neural Network	Parallelism Mode	Batch Size	Micro-batch Size	GPU's per party	Training Time (5 Iters)	Speedup	Inter-Party Comm. Time*/ s	Max Mem / MiB
VGG16-CIFAR10	None	128	-	1	39.83	1.00	25.13	4830.16
VGG16-CIFAR10	Sequential	128	-	4	40.30	0.99	23.54	3998.01
VGG16-CIFAR10	Pipelined	128	64	4	35.97	1.11	27.71	2134.01
VGG16-CIFAR10	Pipelined	128	32	4	44.39	0.90	58.92	1202.01
VGG16-CIFAR10	None	256	-	1	66.00	1.00	30.53	9660.32
VGG16-CIFAR10	Sequential	256	-	4	66.14	1.00	30.47	7996.01
VGG16-CIFAR10	Pipelined	256	128	4	59.40	1.11	49.06	4268.01
VGG16-CIFAR10	Pipelined	256	64	4	64.38	1.03	53.24	2404.01
AlexNet-CIFAR10	None	128	-	4	5.23	1.00	1.90	157.16
AlexNet-CIFAR10	Pipelined	128	64	4	6.30	0.83	3.59	74.89
LeNet-MNIST	None	256	-	4	4.52	1.00	1.15	428.72
LeNet-MNIST	Pipelined	256	128	4	4.05	1.12	1.45	232.03

Table 1: Experimental results on some neural networks. All are two-party computation with communications are over localhost. *For pipelined model parallelism, the inter-party communication times are the "sum" of the time spent doing interparty communication times in each pipeline group / GPU. As each pipeline group thread progresses independently, there are overlaps between their communications. Therefore, some of the sums are greater than the total training time.

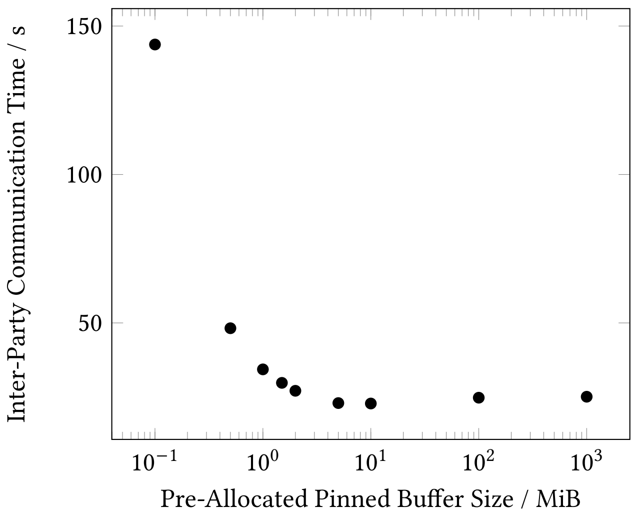


Figure 5: The relationship between time spent in inter-party communication and pre-allocated pinned buffer size. VGG16-CIFAR10, two party protocol (SecureML), single GPU with batch size 128.

communication time are longer than that of the sequential run, the gap in communication time is bigger.

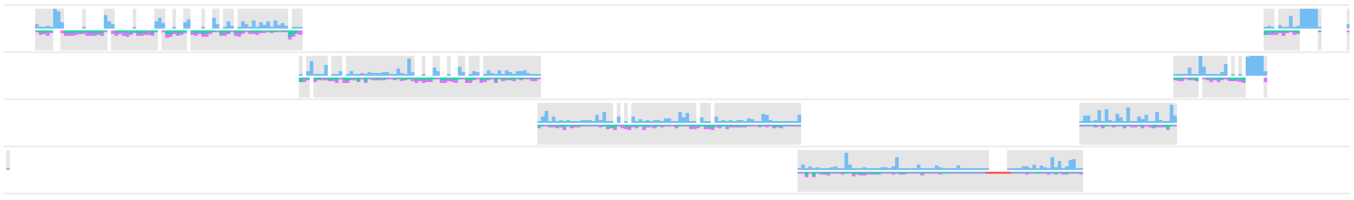
We know the bottleneck isn't GPU-host communication or inter-GPU communication. Both of these operations are lumped together by the profiler, and they occupied 4.80 s and 4.14s for the pipelined and non-pipelined version, respectively, less than one tenths of the total time. Further more, looking at the log, we can see the inter-GPU communication are being conducted in concurrently relative to the main GPU stream. We also know that the computation kernels themselves aren't a bottleneck, as they take only 4.927s and 5.219s for the pipelined and non-pipelined version, respectively.

We believe the most likely reason is that the PCIe became saturated due to socket traffic. For the above example, more than 60 GiB of data are transmitted and received in under one minute, with the peak throughput on the order of 20 Gbps. Although this is significantly lower than the bandwidth of the GPU's PCIe channels, it still puts a strain on other parts of the system, as the inter-party data goes through a TCP/IP connection, and then goes through OS internal systems to reach the other localhost IP. The fact that networking conditions significantly affect training speed is also noted in the original piranha paper. Therefore, it might be beneficial for us to conduct tests in more realistic networking situations, such as in intra-datacenter LAN or WAN conditions.

Although our implementation of pipeline parallelism isn't able to notably increase overall speed, we are able to significantly decrease memory consumption. As shown in Table 1, the pipelined versions have smaller maximum memory consumption, enabling us to train larger neural networks without requiring larger GPUs. In fact, the 512 batch sized VGG16-CIFAR10 can only be training using pipeline parallel, as the single GPU version would go OOM.

6.3 Tensor Level Parallelism

The speedup gained from the tensor-level parallelism depends strongly on the specific hardware configuration. Particularly, the ratio between the inter-device data throughput


(a) Sequential model parallelism

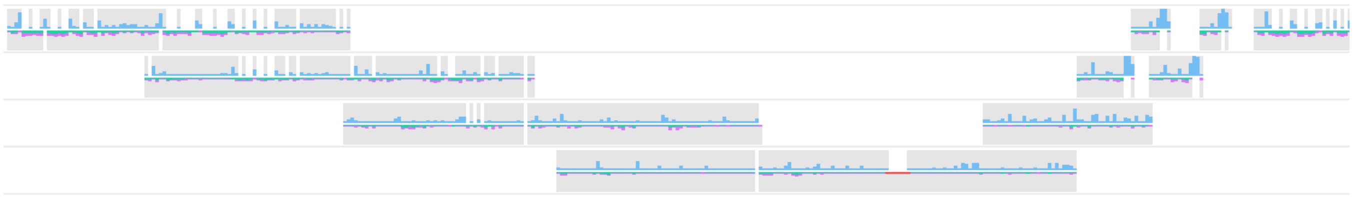

(b) Pipelined model parallelism

Figure 6: These are the GPU activities profiled by Nsight Systems. They represent one iteration during the training of VGG-CIFAR10 using two party computation. Only one party is shown, as the progression of two parties are identical. Each party uses 4 GPUs. The batch size is 256. a) is parallelized sequentially, while b) is pipelined, with a microbatch size of 128. The horizontal timescale of these two lines are identical.

	Total Time		Inter-party Communication			Intra-GPU Computation		
	Pipelined	Sequential	Pipelined	Sequential	Pipe / Seq	Pipelined	Sequential	Pipe / Seq
GPU0	20.73	16.96	14.84	11.91	1.25	5.90	5.05	1.17
GPU1	24.70	16.73	17.78	11.32	1.57	6.92	5.41	1.28
GPU2	27.38	17.75	17.95	11.18	1.61	9.44	6.57	1.44
GPU3	20.82	12.53	12.25	7.10	1.73	8.57	5.43	1.58

Table 2: Per GPU timing comparison for the run shown in Fig 6. All time in seconds. Communication time includes both socket send/recv and GPU host device communication.

and that of the on GPU tensor operation is the most important factor. Lower ratios imply negligible communication cost and thus enables more effective parallelism. The type of inter-device connectivity greatly influences the data transfer speed. NVLink and NVSwitch provides high bandwidth peer-to-peer connections, while PCIe setups have more limited bandwidth, and simultaneous transmission might contend with each other. For NVLink setups, we can see a positive speedup scaling with the number of GPU's as shown in Figure 7, especially when the dimension of the matrices are large enough.

Also, as shown in Figure 8, we profiled the GPU activity during multi-GPU matrix multiplication of two contracting setups. For a NVLink-connected setup in 8a, the profiling diagram indeed shows a concurrent run of memory transmission and operator execution, resulting in a shorter total execution time. However, for setups with limited interconnectivity but fast processing power as in 8b, the cost of memory transmission will impose a severe slowdown when running tensor-level parallel algorithms. And the situation is worsen by the fact that PCIe-connected GPUs usually shares a global bandwidth, resulting in sequential memory transmissions and much worse performance.

Even though we can achieve speedups for large size of operands, our implementation does not perform well for the smaller matrices more commonly found in typical training and inference workload. Hence the tensor parallelism is not integrated into pipeline parallelism, and is not activated during model training.

7 RELATED WORK

Secure Multi Party Computation: Secure multi party computation provides privacy-preserving approaches to machine learning. There's a vast library of works on this topic, considering multiple factors, such as the number of parties involved and the percentage of dishonest participants allowed. There are frameworks intended for general secure multiparty protocol development, such as [2] and [8]. Piranha [10], the work we are trying to improve on, is one of the first general

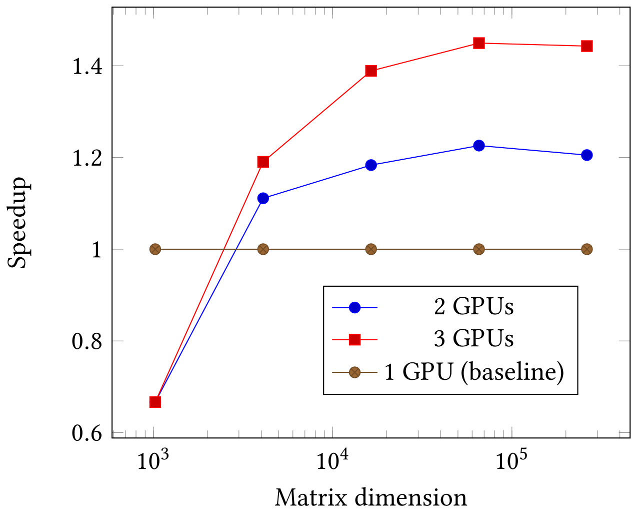
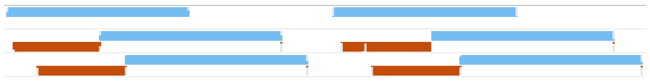
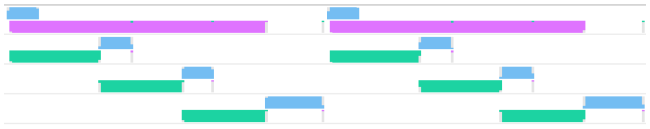


Figure 7: Matrix multiplication benchmark result. Test performed on a system with three Tesla V100-SXM2 32GB, with NVLink interconnection (50GB/s of P2P bandwidth)



(a) Matrix multiplication profile for NVLink setup



(b) Matrix multiplication profile for PCIe setup

Figure 8: GPU activities of matrix multiplication profiled by Nsight Systems. The first setup has three NVLink-connected Tesla V100. Here the red blocks represent memory transmission through NVLink, the green blocks represent memory transmission through PCIe, and the blue blocks represents single-device matrix multiplication.

purpose secure multiparty computation framework that effectively leverages GPU. However, we are not aware of any framework leveraging multiple GPUs for improved performance.

Multi-GPU parallelism: The idea of using multiple GPUs for neural network training, and computation in general is well known. Frameworks such as PyTorch implements various data parallel and model parallel strategies. For pipeline parallelism, PyTorch implements the methods shown in [4] and [5]. Pipeline parallelism can also be done asynchronously, as shown in [6]. [11] explores advanced methods to make pipeline parallelism more effective. PyTorch and other frameworks also support tensor parallelism. Model parallel and data parallel are often used in concert with each other. There are work o There are also work on utilizing both model parallelism and data parallelism, such as [9].

8 FUTURE WORK

To further enhance Piranha, and allows for better scaling, we can use more efficient pipeline parallel algorithms, such as PipeDream[6]. PipeDream reduces the "bubble" by applying asynchronous backward updates. That is, each pipeline group will be alternating between applying forward updates and backward gradients. In it's ideal steady state, PipeDream have no bubbles, achieving N fold speedup.

We can also use more efficient and more fine-grained partitioning algorithms, like in Alpa[11]. Alpa considers the whole computation graph during partitioning, instead of confining to layers. In the context of piranha, it means the objects being partitioned are MPC functions and MPC Shares, instead of neural network layers. Alpa also uses dynamic programming during it's search for the best partition, allowing it to deal with deeper networks.

As we discussed in Section 6.2, MPC is a communication bound protocol. As none of the computation take place on the host, there's no need for interparty communication to go through the CPU at all. We could explore using GPU Remote direct memory access (RDMA), to directly send the device data through various transport to another party, for example, using GPUDirect from NVIDIA [1] to directly send data from GPU to NIC through their shared PCIe root.

The most important future work we believe is to integrate Piranha into a modern deep learning framework. All the parallel algorithms and optimization we've discussed in this report have been implemented in deep learning frameworks such as PyTorch and Tensorflow. Finding a way to integrate piranha would make it easier to leverage the advances in multi-GPU parallelism by industry and academia.

9 CONTRIBUTIONS

Yibai implemented pipeline parallelism and inter-party communication optimization and wrote the report. Shuxian implemented tensor parallelism and pipeline parallelism and wrote the report. Alex wrote the report, report graphics and testing.

REFERENCES

[1] 2022. GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html
[2] Yuanfeng Chen, Gaofeng Huang, Junjie Shi, Xiang Xie, and Yilin Yan. 2020. Rosetta: A Privacy-Preserving Framework Based on TensorFlow. https://github.com/LatticeX-Foundation/Rosetta.

[3] Mark Harris. 2012. *How to Optimize Data Transfers in CUDA C/C++*. https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/

[4] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, and Zhifeng Chen. 2018. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *CoRR* abs/1811.06965 (2018). arXiv:1811.06965 http://arxiv.org/abs/1811.06965

[5] Chiheon Kim, Heungsub Lee, Myungryong Jeong, Woonhyuk Baek, Boogeon Yoon, Ildoo Kim, Sungbin Lim, and Sungwoong Kim. 2020. torchgpipe: On-the-fly Pipeline Parallelism for Training Giant Models. (2020). arXiv:2004.09910

[6] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3341301.3359646

[7] NVIDIA. 2022. *CUDA Runtime API Stream synchronization behavior*. https://docs.nvidia.com/cuda/cuda-runtime-api/stream-sync-behavior.html

[8] OpenMined. 2022. PySyft. https://github.com/OpenMined/PySyft.

[9] Saptadeep Pal, Eiman Ebrahimi, Arslan Zulfiqar, Yaosheng Fu, Victor Zhang, Szymon Migacz, David W. Nellans, and Puneet Gupta. 2019. Optimizing Multi-GPU Parallelization Strategies for Deep Learning Training. *CoRR* abs/1907.13257 (2019). arXiv:1907.13257 http://arxiv.org/abs/1907.13257

[10] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. 2022. Piranha: A GPU Platform for Secure Computation. Cryptology ePrint Archive, Paper 2022/892. https://eprint.iacr.org/2022/892 https://eprint.iacr.org/2022/892.

[11] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. *CoRR* abs/2201.12023 (2022). arXiv:2201.12023 https://arxiv.org/abs/2201.12023