Breaking the Computation and Communication Abstraction Barrier in Distributed Machine Learning Workloads

Abhinav Jangda
University of Massachusetts Amherst
United States

Jun Huang
Ohio State University
United States

Guodong Liu
Chinese Academy of Sciences
China

Amir Hossein Nodehi Sabet
University of California, Riverside
United States

Saeed Maleki
Microsoft Research
United States

Youshan Miao
Microsoft Research
China

Madanlal Musuvathi
Microsoft Research
United States

Todd Mytkowicz
Microsoft Research
United States

Olli Saarikivi
Microsoft Research
United States

ABSTRACT

Recent trends towards large machine learning models require both training and inference tasks to be distributed. Considering the huge cost of training these models, it is imperative to unlock optimizations in computation and communication to obtain best performance. However, the current logical separation between computation and communication kernels in machine learning frameworks misses optimization opportunities across this barrier. Breaking this abstraction can provide many optimizations to improve the performance of distributed workloads. However, manually applying these optimizations requires modifying the underlying computation and communication libraries for each scenario, which is both time consuming and error-prone.

Therefore, we present CoCoNet, which contains (i) a domain specific language to express a distributed machine learning program in the form of computation and communication operations, (ii) a set of semantics preserving transformations to optimize the program, and (iii) a compiler to generate jointly optimized communication and computation GPU kernels. Providing both computation and communication as first class constructs allows users to work on a high-level abstraction and apply powerful optimizations, such as fusion or overlapping of communication and computation. CoCoNet enabled us to optimize data-, model- and pipeline-parallel workloads in large language models with only a few lines of code. Our experiments show that CoCoNet significantly outperforms state-of-the-art distributed machine learning implementations.

CCS CONCEPTS

• **Software and its engineering** → **Domain specific languages**; **Compilers**; • **Computing methodologies** → **Parallel computing methodologies**.

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507778

KEYWORDS

Distributed Machine Learning, Collective Communication, MPI, CUDA, Code Generation, Compiler Optimizations

ACM Reference Format:
Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. 2022. 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 (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3503222.3507778

1 INTRODUCTION

As the trend towards larger machine-learning models continue, from BERT [21] with 340 million parameters, GPT-2 [41] with 1.5 billion parameters, to GPT-3 [17] with 175 billion parameters, model training and inferencing have to be distributed. Moreover, as the computations become resource hungry, optimizing for even the last percentage can have huge benefits in terms of time, energy, and money savings [10, 49].

In machine learning systems today, computation and communication are treated as independent abstractions implemented in different libraries. For instance, computation libraries, such as cuBLAS [2] and cuDNN [3], provide optimized tensor algebra operations, while communication libraries, like NVIDIA Collective Communications Library [8], provide high-performance implementations of collective communication, such as AllReduce. Machine learning frameworks, such as PyTorch [40], call computation and communication kernels from these libraries. Thus, in machine learning applications built atop of such frameworks, the computation and communication operations are invoked separately.

While this separation allows independent optimization of computation and communication kernels, breaking this abstraction boundary can unlock new optimizations that are otherwise not feasible. These optimizations include the following. *Interface* optimization eliminates a mismatch between the caller and the callee of an abstraction. For example, a machine learning model's parameters are stored in non-contiguous buffers, one buffer per layer

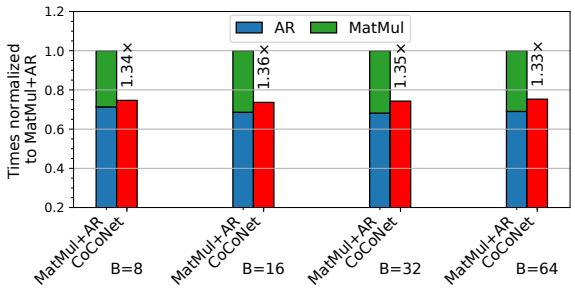


Figure 1: Speedup of co-optimized overlapping over sequential MatMul and AllReduce (for model parallel GPT-2 Model input matrix of [B×1024, 768] and weights of [768, 3072]) on 16 Tesla V100 GPUs.

and hence, need to copy all buffers into a single buffer before calling a collective communication like AllReduce. This copy can be avoided if the communication operation takes a list of arrays as input instead of requiring a single buffer. *Fusion* optimization decreases memory bandwidth usage by generating a single kernel to perform multiple communication and computation operations. *Reorder* optimization moves the computation before or after the communication, thereby either distributing the computation or enabling new fusion possibilities. Finally, *overlapping* optimization orchestrates multiple computation and communication operations in a fine-grained manner to fully utilize both network and compute resources. We elaborate on this possibility below.

In model parallelism, which is one of the distributed machine learning approaches, each layer is distributed across multiple GPUs [47] and the computation for each layer consists of a matrix multiplication (MatMul) on each node followed by an AllReduce. The existing implementation of model parallelism calls individually optimized library functions for MatMul and AllReduce. However, the implementation cannot utilize both network and computation resources simultaneously because the network is idle during MatMul. We can completely utilize both network and computation resources simultaneously by overlapping the computation of MatMul with the communication of AllReduce in a fine-grained manner. The idea is to slice the output into smaller chunks and start the AllReduce communication on a chunk as soon as the MatMul kernel has computed it. To ensure minimum wait time for the AllReduce kernel, we need to schedule the MatMul kernel to compute chunks in the order the AllReduce kernel communicates them. For instance, in the ring algorithm for AllReduce, the n^{th} node sends the chunks to the next node in the order starting from the n^{th} chunk. As such, the MatMul kernel on the n^{th} node needs to generate the chunks in this order. Furthermore, we need to invoke only one MatMul kernel and AllReduce kernel to avoid the overhead of launching multiple kernels. Figure 1 shows that this fine-grained overlapping of MatMul with AllReduce can hide 80% of the execution time of MatMul and provides 1.36× speedup.

However, manually writing these optimizations for each scenario is unproductive, for example, the implementation of above overlapping optimization contains ≈2k lines of CUDA code. Thus, in this paper, we show that by carefully designing a *language* for expressing combinations of computation and communication the benefits of existing machine learning framework's abstraction can be maintained while simultaneously allowing a *compiler* to apply powerful optimizations. To this effect, we propose CoCoNet[1] for generating co-optimized custom computation and communication kernels. Figure 2 presents the overview of CoCoNet. CoCoNet includes a domain specific language (DSL) to express programs containing both computation and communication operations. Inspired by Halide [42], CoCoNet includes a *scheduling* language to specify an execution schedule of the program using a set of transformations. CoCoNet's *autotuner* automatically applies these transformations to optimize a program by breaking the communication and computation boundary. Hence, CoCoNet enables users to quickly generate optimized implementations for specific hardware, topology, and data sizes. CoCoNet's *code generator* automatically generates high-performance computation and communication kernels from a program and its schedule. We used CoCoNet to optimize data-parallel training, model-parallel inference, and pipeline-parallel inference. CoCoNet generated kernels for the Adam [32] and LAMB [52] optimizers speeds up the training time of BERT models by upto 1.68× and can train BERT 3.9 Billion parameter models using only data parallelism, which is not possible with state of the arts. CoCoNet's kernels for model parallelism speeds up the inference in BERT 3.9 Billion and GPT-2 8.2 Billion parameter models by upto 1.51×. CoCoNet's optimized pipeline parallelism kernels speeds up inference times in GPT-2 8.2 Billion and GPT-3 175 Billion parameter models by upto 1.77×. Our implementation of CoCoNet is available at https://github.com/parasailteam/coconet.

2 THE COCONET DSL

The CoCoNet DSL extends the data representation in existing machine learning frameworks and provides constructs to express both computation and communication. The CoCoNet DSL is embedded in C++. Unifying the expression of computation and communication for distributed machine learning in the same DSL is the foundation to enable optimizations across computation and communication.

In this paper, we follow the MPI [22] terminology: `RANK` is the process ID of a distributed process, `GROUP` is a set of concurrent distributed processes, and `WORLD` is the `GROUP` that includes all processes. CoCoNet supports dividing consecutive ranks into one or more process groups.

2.1 Tensor Layout

CoCoNet extends the concept of a tensor in machine learning frameworks from a single device data into distributed forms. Besides item datatype, like `FP32` and `FP16`, and shape, a CoCoNet tensor also includes a *layout* that describes the distributed allocation of tensor's data across a set of ranks. There are three layouts for a tensor: *sliced*, *replicated*, and *local*. A *sliced* tensor is equally distributed among all nodes in a group along a specified dimension with `RANK` identifying the slice for that process. For example, in Figure 3, `w` is sliced among all ranks in `WORLD` in the first dimension and `in` is sliced in the third dimension. A tensor can also be *replicated*

[1] CoCoNet stands for "**Co**mmunication and **Co**mputation optimization for neural **Net**works.

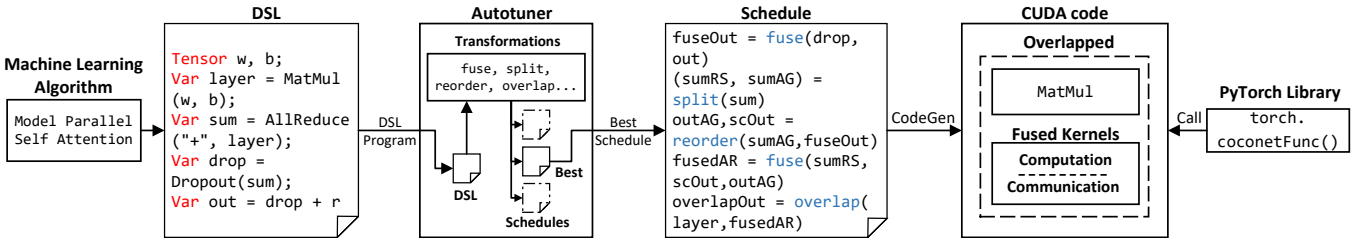


Figure 2: Overview of CoCoNet's workflow. First, a user expresses a machine learning algorithm in the DSL that contains both computation (MatMul) and communication (AllReduce). Then, the autotuner applies transformations to optimize the program while keeping the algorithm unchanged, such as fusing AllReduce and Dropout into FusedAllReduce and overlapping this with MatMul. Finally, CoCoNet generates custom communication and computation code, which is available through PyTorch.

```
1  Tensor w(FP16, [H,H], Sliced(0), WORLD, RANK);
2  Tensor b(FP16, [H], Replicated, WORLD);
3  Tensor in(FP16, [B,S,H], Sliced(2), WORLD, RANK);
4  Tensor r(FP16, [B,S,H], Replicated, WORLD);
5
6  // layer(FP16, [B,S,H], Local, WORLD, RANK)
7  Var layer = MatMul(in, w);
8  // sum(FP16, [B,S,H], Replicated, WORLD)
9  Var sum = AllReduce("+", layer);
10 // dropout(FP16, [B,S,H], Replicated, WORLD)
11 Var dropout = Dropout(sum + b, 0.1);
12 // out(FP16, [B,S,H], Replicated, WORLD)
13 Var out = dropout + r;
14
15 Execute self_attention({w,in,b,r}, {out});
```

Figure 3: An example program in CoCoNet. (B: batch size, S: sequence length, H: hidden dimension size)

across all ranks in a group where it has the same value on each rank and it does not have a rank identifier. For example, the bias `b` and the residual connection `r` are replicated as shown in Figure 3. A *local* tensor has same shape on all ranks but different values on all ranks. A local tensor requires `RANK` to identify the values. For example, in Figure 3, `layer` is a local tensor that represents the result of MatMul operation. A `Scalar` is a zero-dimensional tensor that represents a variable available on all ranks. We discuss the layout of intermediate tensors in the next section.

2.2 CoCoNet's Operations

A CoCoNet program inherits the concept of data-flow graph (DFG) from existing machine learning frameworks with operations as vertices and data dependencies as edges. Operations in CoCoNet can be classified as (i) local computations, such as pointwise computations, matrix multiplication, and convolution, and (ii) cross rank communication operations, such as AllReduce, AllGather, and P2P Send-Recv. Table 1 shows all operations supported by CoCoNet.

A `Var` represents the intermediate tensor obtained after performing an operation. For instance, Figure 3 describes the Megatron-LM [47] model parallel logic of Self-Attention layer in CoCoNet. In this example, the linear layer's weight (`w`) and the input (`in`) are sliced across all ranks while the bias (`b`) and residual (`r`) are replicated on all ranks. A `Var`'s shape and distribution layout are inferred based on the operation and inputs to the operation. For example, line 7 performs a MatMul operation on the input (`in`) and weights (`w`). Since MatMul between two sliced tensors produces a local tensor, `layer` represents the partial result with *local* layout. At line 9, AllReduce computes the sum of `layer` of all ranks and returns a *replicated* tensor with the same values on each rank. The computations at lines 11–13 add the bias, use dropout as an activation, and add the residual. At line 11, the addition of `sum` and `b` follows PyTorch's broadcast semantics[2] by replicating `b` in all dimensions of `sum`. Thus, the shape and layout of output of these operations are same as `sum`. Finally, `Execute` defines the name, inputs, and outputs of the program.

Table 1: Operations supported by CoCoNet includes all common communication and computation operations.

Communication Operations	AllReduce, AllGather, ReduceScatter, Reduce, Broadcast, P2P Send-Recv
Layers	Matrix Multiplication, Convolution
Activations	Dropout, tanh, ReLU
Tensor Operations	+, −, ∗, ÷, Norm, ReduceTensor, Sqrt, Pow, Update

2.3 Fused Collective Communication Operations

CoCoNet enables efficient computations on the output of communication by providing fused collective communication operations, such as FusedAllReduce. Consider the AllReduce in Figure 3 followed by a Dropout (lines 9–11). The abstraction in existing machine learning frameworks requires the output of AllReduce to be stored in memory and then re-loaded by Dropout. FusedAllReduce avoids such stores and loads by directly passing the output of communication to following computations through registers. In addition to the argument of AllReduce, a FusedAllReduce takes computations as extra arguments. Section 5.2 discusses the implementation of Fused Collective Communication Operations.

2.4 Overlapping Operations

CoCoNet supports overlapping multiple dependent computation and communication operations using the `Overlap` construct. For example, consecutive MatMul and AllReduce in Figure 3 (lines 7–9)

[2] https://pytorch.org/docs/stable/notes/broadcasting.html

can be overlapped to fully utilize both network and computation resources. Section 5.3 discusses the implementation of this construct.

2.5 Custom Operations

In CoCoNet, the implementation of an operator needs to define three key properties of the operator: (i) syntax, (ii) semantics, and (iii) code generation. The syntax of an operator is defined using C++ constructors and the semantics are defined by implementing rules to describe the layout and size of the output tensor based on the input tensors. Finally, the code generation requires implementing a function to generate a call to existing libraries or generate fused GPU kernels. The implementation of syntax and semantics can be achieved in a few lines of code, however, implementing the code generation for complex operations like Matrix Multiplication and Convolution can potentially take hundreds of lines of code. Fortunately, in practice the code generation for complex operations can call an optimized implementation of existing libraries.

3 COCONET TRANSFORMATIONS

CoCoNet provides four semantics preserving *transformations* to optimize a program written in the DSL. All transformations are valid based on rules described in the sections below. CoCoNet automatically checks the validity of each transformation based on these rules and throws an error for an invalid transformation.

We call an order of transformations a *schedule*. A user can manually specify the schedule to optimize the program. Additionally, a user can invoke the autotuner to automatically find the best performing schedule for the given problem sizes and the underlying architecture. Below we present each transformation by applying them on the program from Figure 3 and show equivalent CoCoNet programs generated after applying each transformation in Figure 4.

3.1 Splitting Communication

The `split` transformation breaks a collective communication operation into two communication operations. One of the two split policies supported by CoCoNet is

AllReduce Split RS-AG splits an AllReduce into a ReduceScatter to produce a sliced tensor and an AllGather on the sliced tensor to return a replicated tensor.

Running Example The AllReduce in Figure 3 is split into `rsSum` that does a ReduceScatter on `layer` and `agSum` that does an AllGather on `rsSum`.

```
(rsSum, agSum) = split(layer, ARSplitRSAG);
```

The program ① of Figure 4 is the implementation of this schedule where the input to Dropout is replaced by `agSum`.

Validity Since an AllReduce can always be split to a ReduceScatter and an AllGather, this transformation is always valid.

3.2 Reordering Operations

The `reorder` transformation swaps operations with an AllGather or a Broadcast in the DFG of a program. We explain this transformation for AllGather below:

AllGather Reorder reorders an AllGather with communication and computation operations. This transformation changes the layout of the operations, the input and output of operations, and the input and output of the AllGather. We explain this transformation below using the running example.

Running Example In Figure 4, applying the `reorder` transformation changes the program ① to ② by reordering AllGather (`agSum`) with computations `d` and `out`. The reorder transformation replaces these operations in the DFG with three new operations: `scD` and `scOut`, both of which performs sliced computations, and `agOut`, which gathers the final result of computations.

```
(scD, scOut, agOut) = reorder(d, out, agSum);
```

The new sliced computations perform the same operations as original computations with two differences: (i) the output of AllGather used in the computation is replaced by the input of AllGather, and (ii) since the input of AllGather is sliced, all tensors input to the computations are also sliced along the same dimension as the input of AllGather. After reorder, `scD` performs the same computation as `d` but `scD` takes `rsSum` and `Slice(r)` as input. Therefore, the layout of `scOut` is also sliced while the computation is same as `out`. Furthermore, the new AllGather is performed on the outputs of the computations, for example, after reorder, the AllGather (`agOut`) is performed on `scOut`. Figure 5 shows the workflow of this schedule.

Validity The `reorder` transformation is valid only if operations being reordered with an AllGather can be sliced along the dimension the AllGather is performed. The rules of slicing an operation depend on the type of operation and the dimensions of inputs to the operations. For example, `d` and `out` can be sliced because the computations have the same dimensions as `agOut`. Section 4 shows how P2P Send can be reordered with an AllGather.

3.3 Fusing Operations

Fusing multiple computations is a common technique used by existing compilers [18, 20, 24, 27, 42]. CoCoNet extends this concept to fuse multiple computations and communications in a single operation and provides this capability using the `fuse` transformation. Below we explain two fuse policies supported by CoCoNet:

Computation Fuse fuses a series of computations in a single operation that performs all these operations.

AllReduce Fuse fuses a series of ReduceScatter, sliced computations, and AllGather operations in a single FusedAllReduce that performs all these operations.

Running Example We can fuse ReduceScatter (`rsSum`), computations (`scD` and `scOut`), and AllGather (`agOut`) in program ② of Figure 4 into a FusedAllReduce to obtain program ③.

```
fuseAR = fuse(rsSum, scOut, agOut, ARFuse);
```

The `comp` method of `fusedAR` specifies the computation to be fused with FusedAllReduce and returned `out` is the output.

Validity Fusing multiple operations into one operation is valid only if the dependencies in the DFG after fusion are preserved.

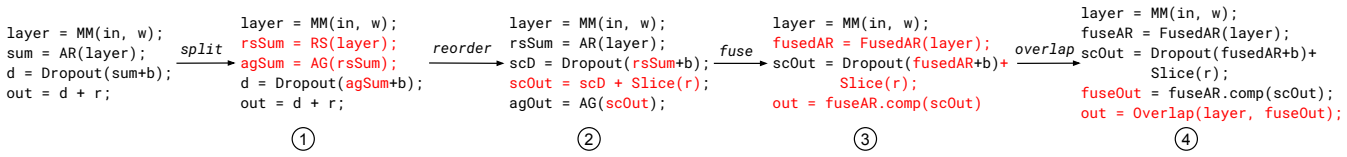


Figure 4: CoCoNet programs produced by performing transformations on the program of Figure 3. Each schedule can be represented as a standalone program. Lines in red highlights changes at a step. (AR: AllReduce, AG: AllGather, and RS: ReduceScatter)

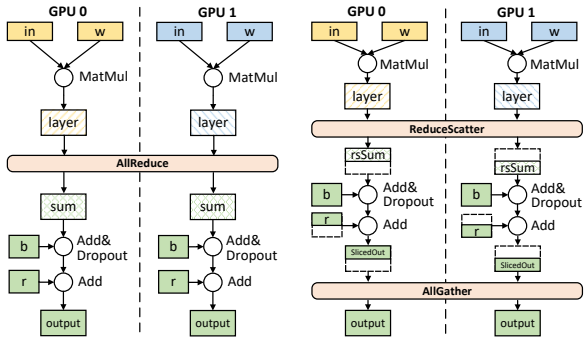


Figure 5: Equivalent programs (from Figure 3) using AllReduce (on left) or using ReduceScatter + AllGather (on right).

3.4 Overlapping Operations

CoCoNet provides the `overlap` transformation to overlap a series of producer-consumer operations to utilize multiple resources of hardware simultaneously.

Running Example In the program ③ of Figure 4 we overlap the matrix multiplication (`layer`) with FusedAllReduce (`fuseAR`) to obtain program in ⑤.

```
layerWithAR = overlap(layer, fusedAR);
```

Validity Overlapping multiple operations is valid only when all operations have a producer-consumer relationship between them.

3.5 Automatic Exploration of Schedules

CoCoNet provides an *autotuner* to automatically explore the space of all schedules of a program and return the schedule that provides the best performance for the underlying architecture and input sizes. First, the autotuner fuses all pointwise computations up to a pre-defined threshold to decrease the search space and then exhaustively explores the schedule space in a breadth first search manner. Finally, the autotuner generates code for all schedules in its search space, executes all programs, and returns the schedule with minimum execution time. Table 3 shows that the autotuner takes only a few seconds to explore the schedule space for all workloads.

4 DISTRIBUTED WORKLOADS IN COCONET

We additionally optimized two distributed machine learning workloads using CoCoNet: (i) parameter update using Adam [32], and (ii) point-to-point communication in pipeline parallelism.

Adam in Data Parallel Training: Figure 6a shows the traditional implementation of parameter update using Adam. First, all ranks

```
1 Var avg = AllReduce("+", g);
2 Var m_ = Update(m, (m*beta1+(1-beta1)*avg));
3 Var v_ = Update(v, (v*beta2+(1-beta1)*avg*avg));
4 Var m1 = m_/(1-Pow(beta1, t));
5 Var v1 = v_/(1-Pow(beta2, t));
6 Var p_ = Update(p, (p - lr * m1/(Sqrt(v1))));
7
8 Execute adam({g,p,v,m,lr}, {p_});
```

(a) Traditional implementation where tensors g is local to each rank and p,m, and v are replicated on all ranks.

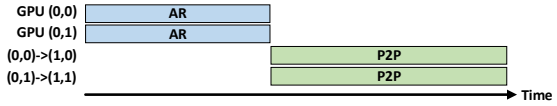
```
1 comps = fuse(m_, v_, m1, v1, p_,
2              ComputationFuse);
3 (rsG, agG) = split(avg, ARSplitRSAG);
4 (scComp, agP, agM, agV) = reorder(agG, comps,
5                                   AGReorder);
6 asSlice(m); asSlice(v); dead(agM); dead(agV);
7 fuseAR = fuse(rsG, scComp, agP, AllReduceFuse);
```

(b) An Optimized Schedule. Tensors g is local, p is replicated on all ranks, while m and v are sliced among all ranks.

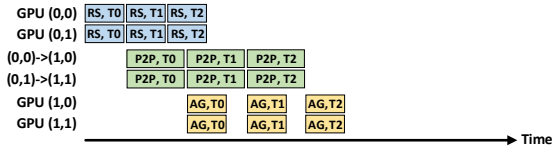
Figure 6: Optimizing parameter update using Adam in CoCoNet. The implementation takes four input tensors: parameters (p), gradients (g), momentum (m), and velocity (v).

average the gradients using AllReduce and then perform computations to update the optimizer state and model parameters. `Update` updates the values of a tensor and reflects the new values in that position in the DFG (lines 2–3). Figure 6b presents a schedule that optimizes this by distributing the computation on all ranks in a single kernel. Line 2 fuses all computations in `comps`. Line 3 splits the AllReduce into a ReduceScatter and an AllGather, such that computations take output of AllGather (`agG`) as input. Line 5 reorders AllGather with computations, such that, each rank performs computations on a slice of tensors. Line 6 slices optimizer states on all ranks to decrease memory usage and removes corresponding AllGather. Finally, line 7 fuses all operations in a single kernel.

Point-to-Point Communication in Pipeline Parallelism: Figure 7a shows a scenario of pipeline parallelism in Megatron-LM with two transformer layers assigned to two groups each with two ranks. Rank i in group j is shown by (j, i). Each group uses model parallelism within its transformer layer. Pipeline parallelism in Megatron-LM works as follows. First, all ranks in the first group reduce their input using AllReduce to get replicated output. Then



(a) In Megatron-LM each GPU sends redundant data.



(b) Communication operations can be overlapped at the granularity of each *communication buffer tile* of data in single kernel call.

Figure 7: Two different schedules of pipeline parallelism.

```
1 Var sum = AllReduce("+", in);
2 Var send = Dropout(recv+b,0.1) + r;
3 Var output = Send(send,
4                   GroupRank(GROUP+1, RANK));
5
6 Execute transformer({in}, {output});
```

(a) Traditional implementation. Each rank of a group sends same data to next group.

```
1 fuseSend = fuse(send, output, SendFuse);
2 (rsSum, agSum) = split(sum, ARSplitRSAG);
3 (scSend, agOut) = reorder(fuseSend, agSum,
4                           AGReorder);
5 overlapOut = overlap(rsSum, scSend, agOut);
```

(b) An Optimized Schedule. Each rank sends only a slice of data to ranks in next group and all operations are overlapped.

Figure 8: Optimizing pipeline parallelism of Megatron-LM. Input tensors: layer output `in`, bias `b`, and residual `r`.

each rank performs pointwise computations over the replicated output. Finally, the first group sends the result of computations to the corresponding rank in the second group using point-to-point (P2P) sends. (Line 2 in Figure 8a shows these computations but are omitted in Figure 7 for simplicity). Since the output of AllReduce in Figure 7a is replicated, redundant data is sent using P2P. We can avoid this redundant communication by splitting the AllReduce to ReduceScatter and AllGather and reordering the P2Ps with the AllGather. Hence, the inter-group communication is reduced by the group size. We can further optimize by overlapping all communication operations. Figure 7b shows that if the buffers are split into multiple tiles (`T0`–`T2` in the figure), intra-group and inter-group communications can be overlapped.

Figure 8a is the original program, while Figure 8b optimizes it by applying transformations. Line 1 fuses the P2P send with computations. Line 2 splits the AllReduce and reorders the returned AllGather with the fused P2P send at Line 4. Hence, P2P send and computations are performed on only a slice of data on the next group where the AllGather is also performed. Finally, all three new operations get overlapped in Line 5.

5 THE COCONET CODE GENERATOR

CoCoNet generates CUDA kernels for computation and communication operations for running on a distributed system with NVIDIA GPUs. For each operation, CoCoNet either generates (i) a call to a collective communication operation, (ii) a CUDA kernel for fused computations, (iii) a CUDA kernel for fused-collective communications (Section 5.2), or (iv) CUDA kernels for overlapping of communication and computation operations (Section 5.3). Moreover, CoCoNet generates code for performing operations on multiple non-contiguous tensors (Section 5.4). After generating CUDA kernels, CoCoNet traverses the program's DFG to generate kernel calls. CoCoNet wraps generated programs as custom operators and integrates them into PyTorch, so that, applications like Megatron-LM can invoke them directly (Section 5.5). We now discuss how CoCoNet adapts NVIDIA Collective Communication Library (NCCL), a widely-used hand-optimized high performance communication library, into a runtime to execute above CUDA kernels.

5.1 NCCL Architecture

NCCL communicates data stored in the global memory of one GPU to a memory location on another GPU using CUDA kernels. NCCL's CUDA kernels perform communication by directly copying data from memory of one GPU to another GPU using GPUDirect Remote Data Memory Access [5]. NCCL's architecture defines four key properties: (i) topology, (ii) protocols, (iii) channels, and (iv) threads in a thread block of the CUDA kernel. NCCL automatically sets key configuration values for these properties based on the size of the input buffer, network architecture, and the size of `WORLD`. To ensure good performance, CoCoNet's code generation must carefully reconfigure these properties when extending NCCL to custom communication and computation. We now provide a high level overview of these properties.

Topology NCCL creates logical topologies, such as ring and tree, over the underlying interconnect network.

Channels NCCL maps copies of a logical topology on the underlying interconnect network. Each copy is called a channel and is assigned to one CUDA thread block.

Protocols NCCL sends data using one of the three protocols: `LL`, `LL128`, and `Simple`. These protocols make different tradeoffs between latency and bandwidth based on the type of inter-node synchronization used: `LL` has the lowest latency and `Simple` provides the highest bandwidth.

Number of Threads NCCL sets a fixed number of threads for each channel (and thread block). NCCL's kernels have high register usage, which limits the number of thread blocks per SM to one.

NCCL Workflow After determining the topology, protocol, number of channels, and number of threads, NCCL calls its CUDA kernel for communication. Each collective communication has three levels of tiling to fully utilize the massive parallelism of GPUs. Data is first divided into *buffer tiles* equal to the size of the communication

buffer. Each buffer tile is further divided among all ranks and channels to obtain *chunks*. Each channel communicates a chunk of data at a time. The *threads* in channels copy elements in and out of the buffers and apply reduction operations (`sum`, `min`, `max`) if needed. We now present details about CoCoNet's code generation.

5.2 Fused Collective Communications

CoCoNet extends the code generation described in the previous sections to support fused collective communication operations. Fused Collective Communication extends NCCL's existing kernels to enable arbitrary pointwise computations and reductions (i.e., beyond `min`, `max`, and `sum`). We inspected more than 10K lines of code in NCCL to identify where computations can be added to pass intermediate values from communication to fused computations directly through registers. CoCoNet supports fusion of both pointwise operations and reductions into NCCL collectives.

Each NCCL protocol utilizes a different mechanism for communication and CoCoNet generates code for all of them. The important features of a protocol are the pack type (64-bit for `LL`, 128-bit for `LL128` and `Simple`) and the load/store access pattern (shared memory for LL128, global memory for `LL` and `Simple`). CoCoNet generates template code for all element types in NCCL, and dispatches accordingly at runtime. There are some subtleties in the code generation worth discussing:

Mixed Precision When the element types of computations and the input tensors are different, CoCoNet finds the largest element type and based on the pack type of the protocol calculates how many elements can be loaded at once. All code will then be generated to operate on these many elements.

Sliced Tensor When a sliced tensor is used by a fused collective communication, all memory accesses performed need to be mapped to elements of the sliced tensor. CoCoNet generates code that produces this mapping. To perform an AllGather on sliced tensors, the inverse of this mapping is produced.

Tensor Reduction To reduce a sliced tensor, each rank reduces locally and do an AllReduce. This AllReduce reuses already established connections among ranks in the surrounding communication kernel to avoid extra startup latency.

5.3 Overlapping of Communication and Computation

Overlapping of computation and communication has been studied in the context of executing stencil computations in a distributed system [14–16, 19, 20, 33, 36, 37, 44, 50, 51]. These works use non-blocking MPI operations to communicate data and simultaneously perform computations on CPUs. A similar approach for overlapping of computation and communication operations for a GPU workload would involve dividing all operations into sub-operations and ensuring dependency between sub-operations using CUDA streams. However, this approach would provide sub-optimal performance because each sub-operation is performed on only a part of data, which leads to in-efficient computation and under-utilization of communication bandwidth.

Figure 9 shows how the fine-grained overlapping of CoCoNet addresses this issue using the example of a MatMul followed by a ring AllReduce. First, it schedules the MatMul kernel (based on CUTLASS [4]) to produce chunks in the same order as the AllReduce consumes them. Here, the n^{th} rank sends chunks in the order starting from the n^{th} chunk. Hence, the MatMul kernel on n^{th} rank produces chunks in the same order. Second, CoCoNet invokes both kernels only once on different streams and synchronizes the AllReduce with the MatMul using an efficient fine-grained spin-lock on a memory buffer to ensure that the AllReduce wakes up as soon as the MatMul produces a chunk. Third, to provide opportunities to tune the 2-D tile sizes of the MatMul kernel, CoCoNet generates a 2-D AllReduce kernel that communicates 2-D chunks, while NCCL AllReduce only supports 1-D continuous chunk.

The example in Figure 9 works as follows. At T = ①, all ranks invoke MatMul and AllReduce kernels. On rank 0, after computing chunk 0, the MatMul kernel wakes the AllReduce kernel at T = ②, which starts communicating chunk 0. While on rank 1, at T = ② the MatMul kernel wakes the AllReduce kernel to communicate chunk 1. Concurrently, both MatMul kernels compute their corresponding next chunk. At T = ③, MatMul kernels finished computing chunk 1 on rank 0 and chunk 2 on rank 1 and wakes up corresponding AllReduce kernels to communicate these chunks. This process continues until all chunks are processed.

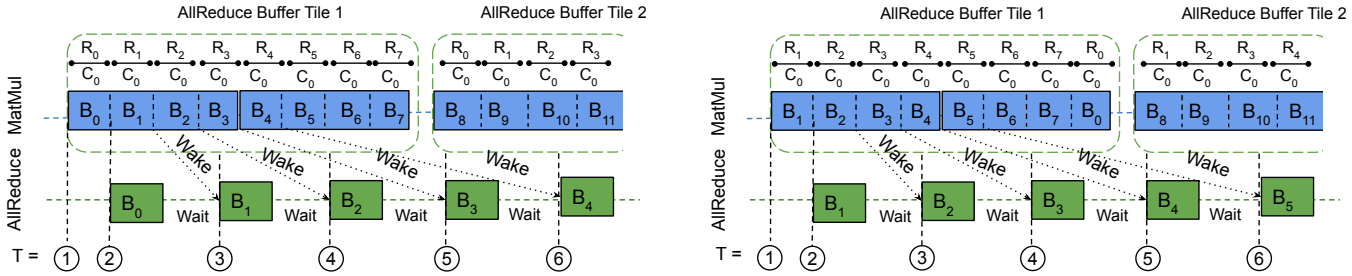
This process allows the MatMul kernel and AllReduce to be overlapped in a fine-grained manner, which reduces the startup latency of AllReduce. Since AllReduce communicates on the same chunk sizes, it achieves maximum communication bandwidth. Furthermore, the MatMul kernel achieves maximum efficiency because the kernel is invoked on the full matrix size. Figure 1 shows that this overlapping provides up to 1.36× better performance and hides more than 80% of the MatMul time.

5.4 Operations on Scattered Tensors

In data parallelism, communication and computation occur on different layers of widely different sizes. Since machine learning frameworks allocate parameters and gradients of layers in non-contiguous buffers, gradients are copied to a large buffer to avoid launching multiple AllReduce operations.

CoCoNet supports generating a single kernel for both computation and communication operations acting on non-contiguous tensors. In this section, we show how CoCoNet modifies NCCL to generate a single communication kernel for scattered tensors. This code generation is non-trivial because NCCL has several design decisions based on the assumption that it is communicating a single contiguous buffer. For example, each thread of a NCCL channel copies only a few elements in each iteration, and hence indexing the correct tensor at a particular offset requires a linear search through all non-contiguous tensors, which can lead to significant overhead. CoCoNet solves this problem by first dividing each tensor into buckets of size at most 2^{10} elements and then assigning buckets to warps in a round-robin manner. This mechanism allows each thread to quickly find the offset in a tensor, since a warp can directly index in its assigned bucket. CoCoNet pre-calculates the number of buckets that belong to the same contiguous buffer and calculates the offset for all of them once.

The process of breaking each tensor to buckets has computation overhead and extra memory requirements. Since this bucketing is



(a) Workflow of overlap on rank 0. Rank 0 starts with chunk 0.

(b) Workflow of overlap on rank 1. Rank 1 starts with chunk 1.

Figure 9: Workflow of CoCoNet's overlapping of MatMul with AllReduce for a Float 16 matrix [8192, 3072] on 8 ranks (R_0 to R_7) with 1 channel (C_0) and 16 MB buffer size. Size of each 2-D chunk (B_0 to B_{15}) is [1024, 1024]. CoCoNet's AllReduce and MatMul enables overlapping without decreasing the communication bandwidth and the efficiency of computation kernels.

Table 2: Time to perform parameter update of all 360 tensors of BERT using Adam/LAMB on 256 Tesla V100 GPUs with scattered tensors implementation and a single contiguous tensor of size equal to the sum of size of all tensors.

Optimizer	Scattered Tensor	Single Tensor
Adam	33.89 ms	33.21 ms
LAMB	37.04 ms	36.71 ms

done only once on the CPU and training tasks run for thousands of iterations on the same tensors, the computation overhead is negligible. Each bucket is represented by a pair of 64-bit tensor address and a 32-bit offset into the associated tensor, leading to $12 \times \left\lceil \frac{N}{2^{10}} \right\rceil$ bytes of extra memory for a tensor with N elements. However, this memory overhead is negligible for large models. For example, for BERT model with 334M elements, the memory requirement is 0.6%. Table 2 shows that the overhead of scattered tensors is insignificant over contiguous tensors.

5.5 PyTorch Integration

We integrated CoCoNet generated code as a function to PyTorch's `torch.distributed` module. This design allows us to re-use the logic for initializing NCCL and provide compatibility with models already using `torch.distributed`. We added wrapper functions for calling CoCoNet generated operations. These wrapper functions prepare the arguments for calling CoCoNet's operations, which includes pre-calculating pointers to the buckets for scattered tensors and clearing the spin-lock buffers for overlapping. Machine learning models can invoke CoCoNet functions using PyTorch.

6 EVALUATION

This section evaluates the effectiveness of CoCoNet through standalone experiments and end-to-end distributed machine learning scenarios of data, model, and pipeline parallelism.

Our experiments are performed on a cluster of 16 NVIDIA DGX-2 nodes where each node contains dual 24-core Intel Xeon CPUs and 16 NVIDIA Tesla V100 (32GB) GPUs. Each GPU within a node is connected to six NVSwitches with six NVLinks (25 GBps per NVLink). Nodes are connected with 8 non-blocking EDR InfiniBand (100 Gbps) network. All nodes run Ubuntu 20.04, CUDA 11.3, cuDNN 8.2 and PyTorch 1.10.

6.1 Data Parallel Training

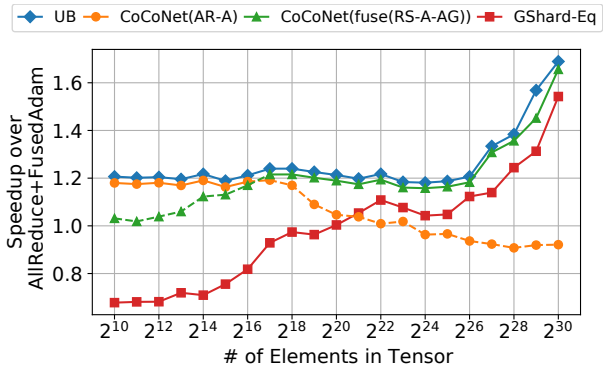
In data parallelism, communication involves an AllReduce of gradients among all ranks. The output is used by the optimizer to update the model parameters. We evaluate CoCoNet generated code for two widely-used optimizers, Adam and LAMB. All our experiments in this section were performed on all 16 DGX-2 nodes in our cluster.

6.1.1 Standalone Experiments. We first perform standalone experiments to explore different CoCoNet schedules over a range of input tensors from 2^{10} to 2^{30} elements. The autotuner generates and executes implementations with different configurations, including all NCCL protocols and all channels from 2 to 64. For each tensor, the autotuner reports the best average result of 1000 iterations.

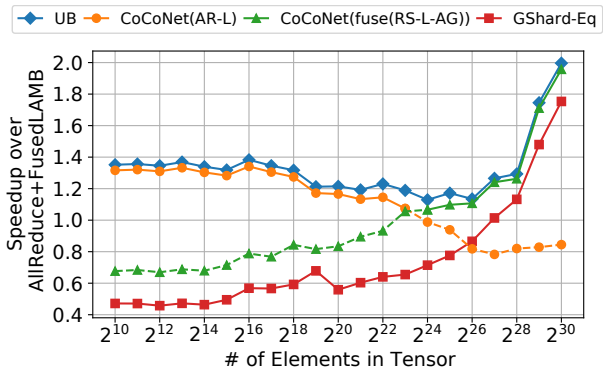
Baselines The baselines perform parameter update by first doing AllReduce over gradients and then call FusedAdam or FusedLAMB from NVIDIA Apex [6]. Both FusedAdam and FusedLAMB fuses all the parameter update computations.

CoCoNet Schedules The autotuner generates following three schedules of Adam and LAMB by applying different CoCoNet transformations for each input size and reports the best schedule to the user for each input size:

(1) **AR-Opt** (Opt = Adam/LAMB) refer to the traditional parameter update technique, i.e., an AllReduce over gradients and then each GPU individually performs the optimizer computation. These schedules fuse all computations into a single kernel, thereby simulating the baseline implementations of FusedAdam and FusedLAMB.
(2) **GShard-Eq** or **RS-Opt-AG** (Opt = Adam/LAMB) are generated from *AR-Opt* by first splitting the AllReduce into ReduceScatter and AllGather, and then reordering AllGather with the fused optimizer computations. Hence, these schedules distribute parameter update across all ranks, similar to GShard [34] and ZeRO [43]. Since GShard does not support execution on GPUs, we refer to this schedule as GShard-Eq in our results.
(3) **fuse(RS-Opt-AG)** (Opt = Adam/LAMB) are generated by fusing all operations of *RS-Opt-AG* into FusedAllReduce.



(a) Mixed-precision Adam. AR-Adam(AR-A) runs best till 2^{16}. fuse(RS-A-AG) represents fuse(RS-Adam-AG) and runs best after 2^{17}.



(b) Mixed-precision LAMB. AR-LAMB(AR-L) runs best till 2^{16}. fuse(RS-L-AG) represents fuse(RS-LAMB-AG) and runs best after 2^{17}.

Figure 10: CoCoNet speedup on 256 GPUs. For each size, CoCoNet chooses the best schedules. UB (upper bound) takes AllReduce-only as max achievable speedup.

Results. Figure 10 shows the speedup of CoCoNet schedules over the baseline for several tensor sizes. The results are shown for mixed-precision [12] using Float 16, and the results for Float 32 are qualitatively similar. In these figures, UB represents the cost of AllReduce alone without doing any computation, and thus is the upper bound of possible speedups.

Even though the *AR-Opt* schedules emulate the baseline implementations, they are faster on smaller tensors. This is because the baseline implementations perform additional preprocessing to optimize the amount of thread-parallelism and instruction-level parallelism per invocation. While this preprocessing cost hurts smaller tensors, its benefit shows up for larger tensors where *AR-Opt* performs worse.

Since *GShard-Eq* and *fuse(RS-Opt-AG)* schedules distribute the optimizer computation, they perform better than the baseline for large tensors. The performance of *fuse(RS-Opt-AG)* shows the advantage of fusing computation and communication kernels as these schedules achieve near optimal speedups for large tensors. These schedules are respectively 13% and 14% faster than GShard-Eq for Adam and LAMB.

For smaller tensor sizes, multiple kernel calls are required for GShard-Eq schedules significantly hurt performance. Interestingly, *fuse(RS-Opt-AG)* schedules are slower than *AR-Opt* schedules for smaller tensor sizes though they require one less kernel call because the fused kernels have a higher register usage, thereby restricting the thread-level parallelism. This demonstrates that the fusion of communication and computation is not always a good idea.

Table 3a shows that the lines of generated code for each schedule are significantly more than the implementation in CoCoNet and the autotuner explored all schedules in 10 seconds. In summary, CoCoNet provides performance improvements over baselines with fewer lines of code. The *AR-Opt* and the *fuse(RS-Opt-AG)* reach close to optimal performance for smaller and larger tensors respectively. This amounts to a speedup of 1.2× to 1.7× for Adam and 1.35× to 2.0× for LAMB. There is no schedule that performs best for all sizes, which demonstrates the need for the autotuner.

Table 3: Lines of code of implementation of distributed machine learning workloads in CUDA and CoCoNet, and time taken by the autotuner to find the best schedule.

(a) Data Parallel optimizer update using Adam and LAMB

Schedule	Generated CUDA	Program in CoCoNet	Autotuner Time
AR-Adam	16	12	
RS-Adam-AG	24	16	9 secs
fuse(RS-Adam-AG)	150	17	
AR-LAMB	80	15	
RS-LAMB-AG	140	17	10 secs
fuse(RS-LAMB-AG)	220	18	

(b) Model Parallel Self Attention and Multi Layer Perceptron

Schedule	Generated CUDA	Program in CoCoNet	Autotuner Time
MM-AR-C	20	10	
MM-RS-C-AG	140	13	12 secs
ol(MM,fuse(RS-C-AG))	≈ 2k	14	

(c) Pipeline Parallel Transformer Layer

Schedule	Generated CUDA	Program in CoCoNet	Autotuner Time
AR-P2P-C-AG	20	10	
RS-P2P-C-AG	140	13	11 secs
ol(RS,fuse(P2P-C),AG))	≈ 2k	14	

6.1.2 Integeration with BERT. We use CoCoNet generated optimizers to train three large BERT models from NVIDIA [7]. We use mixed precision training with both Adam with 8192 global batch size and LAMB with 65536 global batch size.

Baselines We consider three baselines for this experiment:

- **NV BERT** [7] is the NVIDIA BERT Script. It copies gradients of each layer into a single buffer, calls AllReduce on the buffer, and copy back the results into original gradients. Finally, it calls either FusedAdam or FusedLAMB.

- **PyTorch DDP** [35] stores all gradients in buckets of 25MB and overlaps the AllReduce on each gradient bucket with computations during training. After reducing all gradients it calls FusedAdam or FusedLAMB.
- **ZeRO** [43] copies gradients into a contiguous buffer and then distributes Adam's computation similar to *RS-Opt-AG* schedules above. The ZeRO implementation of LAMB does not support distributing optimizer state among GPUs because significant engineering efforts are required to implement reduction over distributed gradients and weights in a distributed LAMB implementation [11].

CoCoNet Integeration We integrated the scattered tensors implementation of *fuse(RS-Opt-AG)* schedule for both Adam and LAMB in PyTorch. These implementations provide three benefits over the baselines: (i) the scattered tensor implementation avoids copying all gradients to a single buffer and allocating this buffer, (ii) the fused schedule performs best for the tensor sizes used in BERT, and (iii) the fused schedule distributes memory of optimizer state among all GPUs.

Results Table 4 shows the speedup provided by CoCoNet in training three BERT models over baselines. For Adam optimizer, CoCoNet provides speedup over all baselines in training BERT 336M because CoCoNet's fused schedules perform better than other implementations. CoCoNet provides even higher speedup on larger BERT models because the fused schedules decrease memory usage by distributing Adam's state over all GPUs, which improves the efficiency of matrix multiplication GPU kernels by enabling higher batch size per iteration. For example, for BERT 1.2B CoCoNet provides 1.53× speedup over NV BERT and PyTorchDDP because of the optimized fused schedule and higher batch size enabled by CoCoNet. On 3.9B parameter model, NV BERT and PyTorch go Out of Memory. ZeRO also supports higher batch size for BERT 1.2B and 3.9B but CoCoNet still gives speedup because of the advantages of scattered tensor implementation of fused schedules.

Results for LAMB are similar. CoCoNet provides up to 1.64× speedup over all baselines. For LAMB, the speedup over ZeRO is higher than Adam because ZeRO does not support distributing LAMB optimizer state, and hence, supports smaller batch sizes as compared to CoCoNet.

In summary, CoCoNet significantly improves data-parallel training time of BERT models. CoCoNet's schedules can be automatically generated and CoCoNet's scattered tensors implementation can support a wide range of optimizers. Not only does the fusion of computation and communication lead to performance improvement over the baselines of PyTorch DDP and ZeRO, it also decreases the memory usage, which helps in increasing the batch size to train models faster.

6.2 Model Parallelism

Megatron-LM [47] uses a model parallel approach for inference and training of transformer models, such as BERT [21] and GPT-2 [41]. A transformer layer contains a self-attention block and a multi-layer perceptron (MLP) block. Last few operations of a self-attention block are the same computations as shown in Figure 3. An MLP block's last operations are similar to Figure 3 with the input tensor and weight sizes as $[B, S, 4 \times H]$ and $[4 \times H, H]$ (B, S, and H are batch size, sequence length, and hidden size, respectively). Since model parallelism is applied within one node, all experiments in this section are performed on a single NVIDIA DGX-2 node.

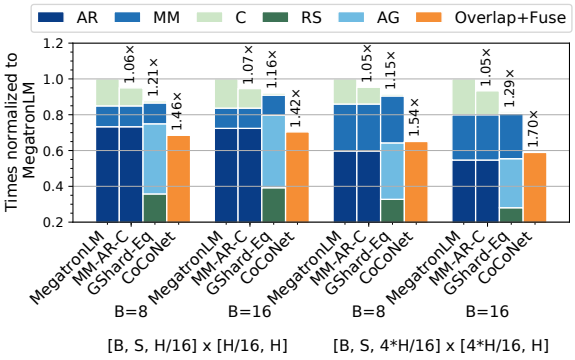


Figure 11: Times of CoCoNet's schedules of model parallel self-attention and multi-layer perceptron of GPT-2 normalized to corresponding Megatron-LM's implementation.

6.2.1 Standalone Experiments. We first perform standalone experiments to evaluate different schedules generated by the autotuner. We compare following schedules for model parallel self-attention code of Figure 3 and similar operations of multi-layer perceptron:

(1) **Megatron-LM** is the baseline implementation of Figure 3 in Megatron-LM.
(2) **MM-AR-C** improves the Megatron-LM implementation by fusing all pointwise computations into one kernel.
(3) **GShard-Eq** or **MM-RS-C-AG** uses the same techniques as GShard. It is generated from *MM-AR-C* by splitting the AllReduce into a ReduceScatter and an AllGather, and reorders AllGather with computations. This schedule represents GShard because GShard is not available for GPUs.
(4) **ol(MM,fuse(RS-C-AG)** is generated from the previous schedule by fusing the ReduceScatter, computation, and AllGather into a FusedAllReduce and then overlapping it with the MatMul. The autotuner returned this as the best schedule and hence represents CoCoNet in our results.

Results We evaluate these schedules with sizes of GPT-2 8.3 Billion parameter model (i.e., $S = 1024$, $H = 3072$) for 8 and 16 batch sizes. Figure 11 shows the times of all schedules normalized to the time of implementation in Megatron-LM. *MM-AR-C* schedule provides speedup over Megatron-LM's implementation because this schedule fuses all pointwise computations in a single GPU kernel. GShard-Eq (*MM-RS-C-AG*) provides 1.15× to 1.29× speedup over Megatron-LM by distributing computations on all ranks. CoCoNet's best schedule (*ol(MM,fuse(RS-C-AG))*) provides 1.42× to 1.70× speedup over Megatron-LM and 1.21× to 1.34× over GShard-Eq because it overlaps FusedAllReduce with the matrix multiplication. Table 3b shows that the lines of generated CUDA code for each schedule are significantly more than the implementation in CoCoNet and the autotuner explored all schedules in 12 seconds.

Table 4: Maximum Micro Batch Size supported by all implementations and speedup of CoCoNet over the baselines when training BERT with three parameter configurations using Adam and LAMB optimizer. OOM represents Out of Memory.

Optimizer	# of Parameters	Maximum Micro Batch Size				Speedup of CoCoNet over		
		NV BERT	PyTorch DDP	ZeRO	CoCoNet	NV BERT	PyTorch DDP	ZeRO
Adam	336 M	32	32	32	32	1.18×	1.22×	1.10×
	1.2 B	8	8	32	32	1.53×	1.52×	1.10×
	3.9 B	OOM	OOM	8	8	–	–	1.22×
LAMB	336M	64	64	64	128	1.20×	1.20×	1.15×
	1.2B	8	8	8	64	1.67×	1.68×	1.64×
	3.9B	OOM	OOM	OOM	8	–	–	–

6.2.2 Integration with Megatron-LM. After integrating CoCoNet's overlap schedule in Megatron-LM, we found that CoCoNet improved inference times of BERT 3.9B parameter model by 1.51× and GPT-2 8.3B parameter model by 1.48×. Hence, overlapping matrix multiplication with fused collective communication significantly improves inference times.

6.3 Pipeline Parallelism

CoCoNet can decrease inference times in pipeline parallelism by fusing computation and communication and overlapping multiple communication operations. We evaluate CoCoNet on computations of model and pipeline parallelism in Megatron-LM for GPT-2 8.3B and GPT-3 175B parameter models. A transformer layer contains several operations but the operations of interest for this experiment are presented in Figure 8a. All experiments in this section are performed on all 16 NVIDIA DGX-2 nodes.

6.3.1 Standalone Experiments. We first perform standalone experiments to evaluate different schedules generated by the autotuner. We compare the following schedules for pipeline parallelism code of Figure 8a:

(1) **Megatron-LM** is the implementation of Figure 8a in Megatron-LM and serves as a baseline for this experiment.
(2) **AR-C-P2P-AG** is generated by slicing the output of AllReduce to perform sliced P2P sends and computations, and finally an AllGather to collect the output of computations. This schedule improves over Megatron-LM by slicing the P2P sends and fusing all the computations.
(3) **GShard-Eq** or **RS-C-P2P-AG** is generated from the previous schedule by splitting the AllReduce into a ReduceScatter and an AllGather, then reordering the AllGather with P2P send and computations. Since this schedule is similar to GShard, it represents GShard-Eq in our results.
(4) **ol(RS,fuse(C-P2P),AG)** is generated from previous schedule by fusing computations with P2P sends, and overlapping all three communication operations (Figure 7b). This schedule is returned by the autotuner as the best schedule and hence, represents CoCoNet in our results.

Results Figure 12 shows the breakdown of each operation with one transformer layer assigned to each node. The sequence length ($S = 2048$) and the hidden size ($H = 12288$) are of GPT-3 175B model. CoCoNet's best schedule *ol(RS,fuse(C-P2P),AG)* is 11.75×–12.21× faster than Megatron-LM's implementation, 2.84× faster than *AR-C-P2P-AG*, and 1.66×–1.72× faster than GShard (*RS-C-P2P-AG*). The speedups are because: (i) sliced P2P reduces cross node communication volume, (ii) fusing communication and computation operations improves memory bandwidth utilization, and (iii) overlapping communication using different connections (NVLink within node and InfiniBand across nodes) improves network bandwidth utilization, while other schedules utilize only one stack at a time. Table 3c shows that the lines of generated CUDA code for each schedule are significantly more than the implementation in CoCoNet and the autotuner explored all schedules in 11 seconds.

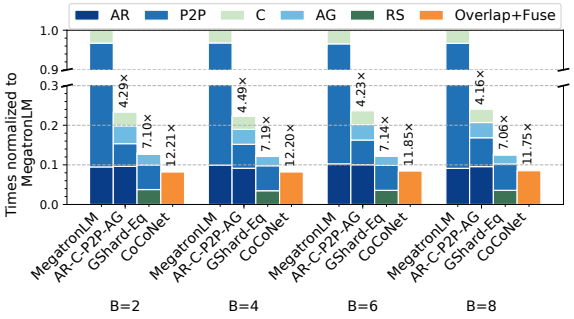


Figure 12: Times of three schedules for GPT-3 175B in CoCoNet for pipeline and model parallelism normalized to Megatron-LM's corresponding implementation.

6.3.2 Integration with Megatron-LM. We evaluated the inference throughput of GPT-2 8.3B and GPT-3 175B parameter models by integrating CoCoNet's *ol(RS,fuse(C-P2P),AG)* schedule in Megatron-LM. Table 5 shows the speedups achieved by CoCoNet.

CoCoNet significantly improves inference throughput of GPT-3 and GPT-2 due to its fusion and fine-grained overlapping of multiple communication operations.

7 RELATED WORK

Distributed Machine Learning Abstractions Existing machine learning frameworks [1, 13, 29, 40, 45] and DSLs [18, 20] provide abstractions for writing distributed machine learning workloads. Similar to CoCoNet, in these abstractions, a distributed machine learning program takes input tensors, performs operations on tensors, and returns tensors as the output. However, unlike these

Table 5: Speedup in inference by CoCoNet's implementation of pipeline parallelism for GPT-2 and GPT-3. Layers per node were obtained by equally distributing layers on all nodes. To evenly distribute layers of GPT-2, number of layers were increased to the nearest multiple of 16, i.e., 80.

Model	Layers per node	Maximum Micro Batch Size	Speedup
GPT-2 8.3B	5	16	1.77×
GPT-3 175B	6	2	1.33×

abstractions, CoCoNet preserves the layout information for each tensor. The layout information enables CoCoNet to perform static type checking of each operation, and automatically perform transformations on the program, which is not possible with existing abstractions.

Distributed Neural Network Training Several works have improved data-, model-, and pipeline-parallel techniques for both training and inference. Mesh-Tensorflow [46] and GShard [34] create *shards* of weights and model state that can be split among ranks. Horovod [45] introduced the *Tensor Fusion* optimization that copies all gradients to a single buffer of 64MB, calls AllReduce on the buffer, and then copies the updated value to original gradients. ZeRO [43] splits weights and model state among ranks and uses ReduceScatter and AllGather to distribute computation. FlexFlow [30] performs operator splitting as a way to represent both data-parallelism and model-parallelism, but does not optimize computation with communication. CoCoNet provides several optimizations over these works that are possible only by breaking the abstraction: (i) scattered tensors that remove extra storage and memory copy operations, (ii) fusion communication collectives, and (iii) novel communication and computation overlapping techniques. PyTorch's DDP [35] overlaps AllReduce of gradients with the forward and backward pass. However, unlike CoCoNet, PyTorch's DDP requires extra memory for overlapping, which can increase training time for very large models [9] and do not support slicing of optimizer parameter update that significantly decrease memory usage. GPipe [26], Pipedream [38], and Narayanan et al. [39] proposed pipeline training to improve model parallelism, by dividing the forward and backward pass into several mini-batches, which are then pipelined across devices. vPipe [53] improves these works by providing higher GPU utilization. CoCoNet improves on these works by overlapping inter and intra-node communication operations. BytePS [31] utilizes CPU in heterogenous clusters to improve training, which is complementary to CoCoNet.

Optimizing Stencil Computations Prior works have proposed several DSLs and optimizations for data-parallel stencil computations on CPUs, GPUs, and other accelerators. Halide [42] and Fireiron [24] separate the algorithm and schedule, which describes the optimizations like fusion, and loop tiling. TVM [18] extends Halide for generating optimized compute kernels. Lift [25, 48] and PolyMage [27] automatically optimize stencil computations for a single GPU. Distributed-Halide [20] extends Halide with scheduling primitives that allow distributing parallelizable dimensions of loops. CoCoNet extends these works to reason about and compose collective communication with computation, which is crucial for distributed machine learning scenarios.

Overlapping Computation and Communication State-of-the-art works on overlapping [14, 33, 36, 37, 50] use either pipelined execution to overlap communication and computation or non-blocking MPI operations. Pencil [51] improves upon these works by performing pipelining within a process and supports computations in multiple connected iteration spaces. Several techniques distribute tiles and automatically generate communication [16, 20, 44]. Basu et. al. [15] uses overlapped tiling in each process to remove communication between processes. Denis and Trahay [19] studied the efficiency of overlap. dCUDA [23] provides hardware supported overlap. These works for MPI+OpenMP are valid for CPU based stencil computations that require sends and receives to share the halo regions. However, unlike CoCoNet, these works do not support overlapping between collectives communication and complex computations like convolutions and matrix multiplications. CoCoNet supports overlapping multiple computation and communication operations on GPUs without an accelerator.

8 CONCLUSION

This paper introduced CoCoNet, a language to describe distributed machine learning workloads and optimize them across computation and communication boundary. We show that CoCoNet generated code significantly improves several training and inference times of large language models. In the future we plan to automate the optimizations through smart search.

9 DATA AVAILABILITY STATEMENT

The artifact for this paper [28] contains the source code of our implementation of CoCoNet and the benchmarking infrastructure to reproduce all the results in Section 6.

ACKNOWLEDGMENTS

We thank the reviewers and our shepherd, Tyler Sorensen, for their constructive feedback. This work was partially supported by the National Science Foundation grant CCF-2052696.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact appendix describes how to reproduce results for standalone experiments in Figure 10, 11, and 12 and integration results in Section 6.1.2, 6.2.2, and 6.3.2. This artifact includes the CoCoNet DSL and compiler, and CoCoNet's generated code integrated with PyTorch, Megatron-LM, and NVIDIA Bert. To reproduce the results, the experiments should be executed on a system similar to our experimental system. However, all experiments can be executed on a system with more than one NVIDIA GPUs.

A.2 Artifact Check-list (meta-information)

- **Program:** CoCoNet DSL and compiler written in C++.
- **Compilation:** A C++ compiler (`g++` or `clang`) to compile CoCoNet. A C++ compiler with MPI support (`mpicxx`) and CUDA compiler (`nvcc`) to compile generated programs.

- **Binary:** Each CoCoNet program compiles to a binary that generates an MPI program containing CUDA kernels.
- **Data set:** BERT, GPT-2, and GPT-3 training datasets for integration experiments.
- **Run-time environment:** Ubuntu 20.04 with Python 3.7+, CUDA 11.0+, and OpenMPI 4.0+.
- **Hardware:** We performed experiments on 16 NVIDIA DGX-2 nodes, i.e., a total of 256 NVIDIA Tesla V100 GPUs. However, the experiments can be executed on any system with two or more GPUs.
- **Run-time state:** Python, MPI, and CUDA.
- **Execution:** Use `mpirun` to run the experiments.
- **Metrics:** Decrease in execution time of benchmarks.
- **Output:** Execution time of each experiment and CoCoNet speedup over baselines.
- **Experiments:** Execution of standalone experiments and training and inference tasks of BERT, GPT-2, and GPT-3 models.
- **How much disk space required (approximately)?:** 100 GB in total. 90% of the space usage is required for storing dataset.
- **How much time will be spent in preparing the workflow (approximately)?:** 1 hour.
- **How much time is needed to complete experiments (approximately)?:** 5 hours.
- **Publicly available?:** Yes.

A.3 Description

A.3.1 How to Access. The CoCoNet implementation and the benchmarking infrastructure used in our evaluation are publicly available as the artifact [28]. This artifact contains a zip file with two directories: (i) `coconet`, which is the implementation of CoCoNet, and (ii) `coconet-experiments`, which is the benchmarking infrastructure. Latest versions of these directories are available at https://github.com/parasailteam/coconet and https://github.com/parasailteam/coconet-experiments.

A.3.2 Hardware Dependencies. All benchmarks can be executed on a distributed system with two or more NVIDIA GPUs. However, our results will be reproducible on the evaluation system described in Section 6.

A.3.3 Software Dependencies. Our experiments require a system running Ubuntu 20.04 with Python 3.8+ and CUDA 11.0+. Prerequisites and their installation procedure is described in `README.md` files of `coconet` and `coconet-experiments` directories.

A.3.4 Data Sets. The standalone benchmarks (Figure 10, 11, and 12) do not require any dataset. Datasets required for executing experiments in Section 6.1.2, 6.2.2, and 6.3.2 can be obtained by following *Dataset* section of `README.md` in `coconet-experiments`.

A.4 Installation

Following instructions have been tested with Ubuntu 20.04.

Standalone Experiments Dependencies. Install dependencies by following the *Prerequisites* section in `README.md` file of `coconet` directory.

Integration Experiments Dependencies. Follow the *Prerequisites* section in `README.md` file of `coconet-experiments` directory to build PyTorch and install all dependencies for Megatron-LM and NVIDIA Bert.

A.5 Experiment Workflow

A.5.1 Standalone Experiments. This section describe how to execute standalone experiments of Section 6 and produce results for Figure 10, Figure 11, and Figure 12. All of these experiments will take 1 hour combined.

(1) Install all CoCoNet prerequisites in `coconet/README.md`.
(2) The `experiments/` directory contains all scripts for standalone experiments.

```
$ cd coconet/experiments/
```

(3) Since all our experiments uses MPI to run the executable on all GPUs, set the environment variable `NPROC` to the number of GPUs in the system. In our experiments, we set `NPROC` to 256 as follows:

```
$ export NPROC=256
```

Note: Setting `NPROC` to a value more than the number of GPUs in a system can lead to failed experiments.
(4) If the experiments are performed on a system with multiple nodes then additional arguments to `mpirun` can be passed by setting the `MPI_ARGS` environment variable.

Data-Parallel Experiments.

(1) To execute standalone data parallel experiments execute `data-parallel-exp.py`. This script takes a directory to store the results as an argument. Additionally, the script requires `MASTER_ADDR` and `MASTER_PORT` to be passed as `MPI_RUN_ARGS`. If the experiments are done on a single system, then it is common to set `MASTER_ADDR=127.0.0.1` and `MASTER_PORT=10000`.

```
$ export MPI_ARGS="-x MASTER_ADDR=127.0.0.1"
$ export MPI_ARGS="$MPI_ARGS -x MASTER_PORT=10000"
$ python data-parallel-exp.py results/
```

The above execution of script will execute all data parallel executables and store the results in the `results` directory.
(2) Generate both graphs of Figure 10 by executing the script `gen-data-parallel-graphs.py`. This script takes the directory with results generated in the previous step as an argument.

```
$ python gen-data-parallel-graphs.py results/
```

Graphs are stored in two files of `experiments` directory: `results-adam-fp16.pdf` and `results-lamb-fp16.pdf`.

Model-Parallel Experiments.

(1) To execute standalone model-parallel experiments execute `model-parallel-exp.py`. Similar to the previous script, this script also takes a directory to store results as its argument.

```
$ python model-parallel-exp.py results/
```

The script will execute all model parallel executables and stores the results in the `results` directory.
(2) Generate Figure 11 by executing following script. This script will take above results directory as its argument.

```
$ python gen-model-parallel-graphs.py results/
```

Graph is stored as `results-model-parallel.pdf`.

Pipeline-Parallel Experiments.

(1) To execute standalone pipeline-parallel experiments execute `pipeline-parallel-exp.py`. This script also requires a directory to store results as its command line argument.

```
$ python pipeline-parallel-exp.py results/
```

Above execution of the script will execute all pipeline parallel executables and store the results in `results` directory.

(2) To generate Figure 12 execute the script `gen-pipeline-parallel-graphs.py`. This script takes the directory containing above results as its argument.

```
$ python gen-pipeline-parallel-graphs.py results/
```

The graph is stored in `results-model-parallel.pdf`.

A.5.2 Integration Experiments. In this section, we will execute the integration experiments of Section 6.1.2, 6.2.2, and 6.3.2.

Prerequisites. Install prerequisites and obtain dataset by following the steps in `coconet-experiments/README.md`.

Data-Parallel Training. Go to `Nvidia-Bert` directory and execute `coconet-experiments.py`.

```
$ cd NV-BERT
$ python coconet-experiments.py
```

This script will execute data parallel training experiments and then print Table 4. This experiment will take 1 hour to complete. This script contains maximum batch sizes supported by each implementation for our evaluation system of 256 Tesla V100 GPUs. It is possible that for a different system the maximum batch size will be different. The batch size dictionary in `coconet-experiments.py` can be modified to find maximum batch size for underlying system.

Model-Parallel Inference. Go to `MegatronLM-Model-Parallel` directory and execute `coconet-experiments.py`.

```
$ cd MegatronLM-Model-Parallel
$ python coconet-experiments.py
```

This script will execute model parallel inference experiments and then print the values in Section 6.2.2. This experiment will take less than 30 minutes to complete.

Pipeline-Parallel Inference. Execute `coconet-experiments.py` in the directory `MegatronLM-Pipeline-Parallel`.

```
$ cd MegatronLM-Pipeline-Parallel
$ python coconet-experiments.py
```

This script will execute pipeline parallel inference experiments and then print the table in Section 6.3.2. This experiment will take 3 hour to complete.

A.6 Evaluation and Expected Results

Standalone Experiments. The figures generated by the experiments of Section A.5.1 can be matched with the figures: 10, 11, and 12.

Integration Experiments. The results generated in experiments of Section A.5.2 can be matched with the results in Section 6.1.2, 6.2.2, and 6.3.2.

REFERENCES

[1] Accessed: 2022-01-12. Apache mxnet. https://mxnet.apache.org/.
[2] Accessed: 2022-01-12. cuBLAS. https://docs.nvidia.com/cuda/cublas/index.html.
[3] Accessed: 2022-01-12. cuDNN. https://docs.nvidia.com/cuda/cudnn/index.html.
[4] Accessed: 2022-01-12. CUTLASS. https://github.com/NVIDIA/cutlass.
[5] Accessed: 2022-01-12. GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html.
[6] Accessed: 2022-01-12. NVIDIA Apex. https://github.com/NVIDIA/apex.
[7] Accessed: 2022-01-12. NVIDIA BERT. https://github.com/NVIDIA/DeepLearningExamples.
[8] Accessed: 2022-01-12. NVIDIA Collective Communication Library. https://github.com/NVIDIA/nccl.
[9] Accessed: 2022-01-12. NVIDIA Megatron-LM. https://github.com/NVIDIA/Megatron-LM/.
[10] Accessed: 2022-01-12. OpenAI's GPT-3 Language Model: A Technical Overview. https://lambdalabs.com/blog/demystifying-gpt-3/.
[11] Accessed: 2022-01-12. Parameter fusion in optimizer partition makes LAMB behaves differently. https://github.com/microsoft/DeepSpeed/issues/490.
[12] Accessed: 2022-01-12. Training with Mixed Precision. https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html.
[13] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*.
[14] Youcef Barigou and Edgar Gabriel. 2017. Maximizing Communication–Computation Overlap Through Automatic Parallelization and Run-time Tuning of Non-blocking Collective Operations. *International Journal of Parallel Programming* 45 (2017). https://doi.org/10.1007/s10766-016-0477-7
[15] P. Basu, A. Venkat, M. Hall, S. Williams, B. Van Straalen, and L. Oliker. 2013. Compiler generation and autotuning of communication-avoiding operators for geometric multigrid. In *20th Annual International Conference on High Performance Computing*. https://doi.org/10.1109/HiPC.2013.6799131
[16] Uday Bondhugula. 2013. Compiling Affine Loop Nests for Distributed-Memory Parallel Architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. https://doi.org/10.1145/2503210.2503289
[17] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*.
[18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*.
[19] A. Denis and F. Trahay. 2016. MPI Overlap: Benchmark and Analysis. In *2016 45th International Conference on Parallel Processing (ICPP)*. https://doi.org/10.1109/ICPP.2016.37
[20] Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. 2016. Distributed Halide. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. https://doi.org/10.1145/2851141.2851157
[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019*. Association for Computational Linguistics. https://doi.org/10.18653/v1/n19-1423
[22] Message Passing Interface Forum. 2012. MPI: A Message-Passing Interface Standard Version 3.0.
[23] Tobias Gysi, Jeremia Bär, and Torsten Hoefler. 2016. dCUDA: Hardware Supported Overlap of Computation and Communication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. https://doi.org/10.1109/SC.2016.51
[24] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. 2020. Fireiron: A Data-Movement-Aware Scheduling Language for GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. https://doi.org/10.1145/3410463.3414632
[25] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High Performance Stencil Code Generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. https://doi.org/10.1145/3168824
[26] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems 32*.

[27] Abhinav Jangda and Arjun Guha. 2020. Model-Based Warp Overlapped Tiling for Image Processing Programs on GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. https://doi.org/10.1145/3410463.3414649

[28] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hosseing Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. 2022. CoCoNet: Co-Optimize Computation and Communication for Distributed Neural Networks. https://doi.org/10.6084/m9.figshare.18480953. https://doi.org/10.6084/m9.figshare.18480953.v3

[29] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

[30] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of Machine Learning and Systems*.

[31] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation*. https://www.usenix.org/system/files/osdi20-jiang.pdf

[32] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015*. http://arxiv.org/abs/1412.6980

[33] N. Koziris, A. Sotiropoulos, and G. Goumas. 2003. A pipelined schedule to minimize completion time for loop tiling with computation and communication overlapping. *J. Parallel and Distrib. Comput.* 63, 11 (2003). https://doi.org/10.1016/S0743-7315(03)00102-3

[34] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2021. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. In *International Conference on Learning Representations*.

[35] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.* (2020). https://doi.org/10.14778/3415478.3415530

[36] H. Lu, S. Seo, and P. Balaji. 2015. MPI+ULT: Overlapping Communication and Computation with User-Level Threads. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. https://doi.org/10.1109/HPCC-CSS-ICESS.2015.82

[37] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. 2010. Overlapping Communication and Computation by Using a Hybrid MPI/SMPSs Approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*. https://doi.org/10.1145/1810085.1810091

[38] 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*. https://doi.org/10.1145/3341301.3359646

[39] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. 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*. https://doi.org/10.1145/3458817.3476209

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*.

[41] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[42] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/2491956.2462176

[43] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations toward Training Trillion Parameter Models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

[44] I. Z. Reguly, G. R. Mudalige, and M. B. Giles. 2018. Loop Tiling in Large-Scale Stencil Codes at Run-Time with OPS. *IEEE Transactions on Parallel and Distributed Systems* (2018). https://doi.org/10.1109/TPDS.2017.2778161

[45] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. arXiv:1802.05799 [cs.LG]

[46] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. In *Advances in Neural Information Processing Systems*.

[47] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL]

[48] M. Steuwer, T. Remmelg, and C. Dubach. 2017. LIFT: A functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO.2017.7863730

[49] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and Policy Considerations for Deep Learning in NLP. arXiv:1906.02243 [cs.CL]

[50] Hari Subramoni, Sourav Chakraborty, and Dhabaleswar K. Panda. 2017. Designing Dynamic and Adaptive MPI Point-to-Point Communication Protocols for Efficient Overlap of Computation and Communication. In *High Performance Computing*.

[51] Hengjie Wang and Aparna Chandramowlishwaran. 2020. Pencil: A Pipelined Algorithm for Distributed Stencils. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. https://doi.org/10.1109/SC41405.2020.00089

[52] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. 2020. Large Batch Optimization for Deep Learning: Training BERT in 76 minutes. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Syx4wnEtvH

[53] Shixiong Zhao, Fanxin Li, Xusheng Chen, Xiuxian Guan, Jianyu Jiang, Dong Huang, Yuhao Qing, Sen Wang, Peng Wang, Gong Zhang, Cheng Li, Ping Luo, and Heming Cui. 2022. vPipe: A Virtualized Acceleration System for Achieving Efficient and Scalable Pipeline Parallel DNN Training. *IEEE Transactions on Parallel and Distributed Systems* (2022). https://doi.org/10.1109/TPDS.2021.3094364