applied sciences

MDPI

Article

Optimizing Single DGX-A100 System: Overcoming GPU Limitations via Efficient Parallelism and Scheduling for Large Language Models

Kyeong-Hwan Kim and Chang-Sung Jeong *

Department of Electrical Engineering, Korea University, Seoul 02841, Republic of Korea; kyunghwan@korea.ac.kr
* Correspondence: csjeong@korea.ac.kr

Abstract: In this study, we introduce a novel training algorithm specifically designed to overcome the limitations of GPU memory on a single DGX-A100 system. By utilizing the CPU and main memory in the training process and applying a strategy of division and parallelization, our algorithm enhances the size of the trainable language model and the batch size. In addition, we developed a comprehensive management system to effectively manage the execution of the algorithm. This system systematically controls the training process and resource usage, while also enabling the asynchronous deployment of tasks. Finally, we proposed a scheduling technique integrated into the management system, promoting efficient task scheduling in a complex, heterogeneous training environment. These advancements equip researchers with the ability to work with larger models and batch sizes, even when faced with limited GPU memory.

Keywords: heterogeneous systems; natural language processing; model parallelism

Citation: Kim, K.-H.; Jeong, C.-S Optimizing Single DGX-A100 System: Overcoming GPU Limitations via Efficient Parallelism and Scheduling for Large Language Models. *Appl. Sci.* **2023**, *13*, 9306. https://doi.org/10.3390/app13169306

Academic Editors: Andres Alvarez-Meza and David Cárdenas-Peña

Received: 18 July 2023
Revised: 11 August 2023
Accepted: 14 August 2023
Published: 16 August 2023

1. Introduction

Artificial intelligence natural language processing models have developed rapidly based on the transformer-based model [1–3]. The transformer model is an artificial intelligence model that can understand and generate text like a human. These models perform well when more parameters are included [4,5]. In order to train a model with many parameters, an AI supercomputer is required. NVIDIA DGX-A100 system [6] is a widely used AI supercomputer, consisting of 8 A100 Tensor Core GPUs with 40 GB of GPU memory. Recently, the size of language models has become very large, and in order to secure the GPU memory required for learning, dozens to hundreds of DGX-A100 clusters are used for training.

However, obtaining a substantial number of these supercomputers is challenging for many researchers. This barrier often results in an inability to procure the ample GPU memory necessary for language model training.

In this paper, we introduce a novel training algorithm designed to overcome GPU memory constraints in a single DGX-A100 system. This enhancement facilitates the training of larger language models and enables the expansion of batch sizes. Our approach manages the entire model by dividing it into sequential sub-models on the CPU and main memory [7,8]. Each sub-model is then composed of parallel tasks, which are transmitted to a multi-GPU environment, executed simultaneously, and the results returned to the CPU.

The proposed algorithm operates in an environment where frequent communication and computation between various resource types are essential. The asynchronous management and deployment of resources and tasks are crucial for efficient functioning. Processing performance is further optimized via strategic task placement across multiple execution streams (GPU, CPU, and IO).

We have developed a comprehensive management system to effectively organize, synchronize, and execute complex tasks in this intricate environment. Additionally, we

present a scheduling technique that organizes the workflow within the management system. This approach allows the training algorithm to utilize heterogeneous resources more efficiently, enabling the training of larger language models.

Our methodology significantly reduces the GPU memory required for large-scale model training. It has been specifically optimized for the DGX-A100 system, allowing researchers to train more extensive models and larger batch sizes even within limited resources. The major contributions of this study are threefold.

First, we focus on developing a training algorithm tailored to the heterogeneous environment of the DGX-A100 system. A core technology was implemented to enhance the system's overall efficiency, overcoming GPU memory limitations via a novel method of model division and parallelization using CPU and memory.

Second, we have crafted a high-performance management system that ensures the smooth execution of our newly developed algorithm. This system orchestrates the training process and resources, managing and deploying computational tasks asynchronously. The result is not only optimized learning task efficacy but also a substantial enhancement in the overall learning process performance.

Lastly, we introduce a strategic scheduling technique for the training algorithm within our management system. This method places tasks effectively across multiple execution streams and coordinates communication and computational tasks between heterogeneous resources. The coordination leads to a maximized overall training efficiency, aligning the process with the optimal resource usage [9].

The rest of this paper unfolds as follows: Section 2 establishes the need and uniqueness of our study by exploring the previous efforts to overcome GPU memory constraints and enable large-scale parallel learning. Section 3 meticulously details our methodology, encompassing the training algorithm with strategies for parallel processing across GPUs and CPUs, a management system to oversee parallel tasks and heterogeneous resources, and a workflow section that explains the task allocation within various execution flows. Section 4 focuses on the experimental design, where we test models of varying sizes and batch sizes to validate our methodology's performance, with a special emphasis on measuring scalability on a single DGX-A100 node.

2. Related Works

Language models based on transformers show good results in NLP. These language models record better performances as the number of model parameters increases. Recently proposed large-scale language models require multiple GPU nodes for training. However, these GPU nodes are expensive and difficult to acquire. Research in various directions is being conducted to solve the shortage of GPU nodes.

To increase the size of the training model, it is very important to secure the GPU memory. The most intuitive way is to use less GPU memory or use it efficiently to free up memory. To use less memory, there is a model that effectively uses parameters compared to the existing model or reduces the memory used in the calculation process [10–13]. First, an effective model structure and hyperparameters must be set to achieve this. We will adopt the previously studied architecture [14] and hyperparameter block structure and provide use cases for them.

Recently, a method of securing multiple GPU nodes to train a large language model and effectively utilizing these nodes to organize training has been mainly researched. A large model must be divided and trained on multiple GPUs to configure effective training with the corresponding nodes. Splitting the model means dividing and loading the parameters across multiple GPUs, and when the parameters are split, the calculation results and communication overhead must be considered. The strategy for dividing the model comprises layer-level model parallelism and tensor-level model parallelism. In general, layer-level model parallelism divides and distributes layers to each GPU and executes them sequentially. This constitutes a pipeline and is called pipeline parallelism. To effectively implement pipeline parallelism, it is essential to consider how to divide

models and pipeline scheduling effectively, and research is being conducted regarding them [15–18].

However, GPU idle time inevitably occurs because the pipeline necessarily accompanies the pipeline bubble. Tensor-level model parallelization divides and distributes the basic tensor operations (Matmul, Conv) constituting the layer to multiple GPUs [19–21]. This calculates the parameters for one operation by dividing them into several GPUs, then collects the results between them. Communication takes place in this process. Recently, in order to eliminate the complexity of model parallelism, Auto Parallelism is being studied. Auto Parallelism automatically implements model parallelism by considering the hardware and model structure at the system level [22]. This shows novel parallelism performance in all general-purpose deep-learning models, but it is necessary to design model parallelism directly to secure resource use efficiency in large-scale artificial intelligence models.

To effectively scale up the size of a trained model, it is necessary to partition or parallelize the model effectively, but there is a limit to the models that can be trained when there are not enough GPU nodes. To solve this problem, research is being conducted on CPU and main memory techniques for training [23,24]. Utilizing CPU and GPU simultaneously can be implemented simply via heterogeneous frameworks or platforms [25]. However, specific implementations must consider the structure of the model and the parallelism technique. The CPU and main memory technique for learning is largely implemented in two contexts: Out-of-Core (OoC) computation and Model Swapping. Ooc is a method of using CPU memory to provide virtual memory that can be used when GPU memory is insufficient [26]. When necessary calculations are performed on the GPU, the data are transferred from the CPU memory to the GPU memory, and when the calculation is complete, it is moved back to the CPU memory [27]. Model Swapping is a method of moving the model parameters to the CPU memory. Both technologies can expand the memory, but this must be implemented carefully considering the communication overhead.

Recently, several studies construct large-scale deep learning training or inference environments by utilizing Model Swapping strategies to solve memory shortages. Harmony [28] proposes using scheduling and data communication techniques to increase the limits of training deep-learning models on a single commercial server. Computron [29] implements a model parallel swap design that accelerates model parameter transfers by utilizing the cluster's aggregate CPU-GPU link bandwidth. This design can make the swapping of large models feasible and improve resource utilization. These frameworks can serve the deep-learning models of various structures.

This study developed a large-scale language model learning system optimized for the DGX-A100 environment by reflecting the reviewed research trends. Utilizing the model structure [30] and model configuration hyperparameters [14] that have been verified in previous studies, the learning process is efficiently conducted, and memory usage is minimized. In this study, the existing language model parallelism methods such as Layer Parallelism and Tensor Parallelism were adopted to efficiently parallelize the model. Previous studies have shown that transformer models can be split into multiple layers, and these split layers can be further split via Tensor Parallelism. Studies have proven that communication overhead can be minimized while maintaining the original calculation results [19–21]. As a way to solve the GPU memory shortage problem, recent studies suggest a model swapping method [27,29,31]. This method frees up GPU memory by moving model parameters to CPU memory. In this study, this existing method was expanded and optimized in the direction of minimizing the communication overhead between the CPU and GPU during language model training on the DGX-A100 system. Therefore, this study proposed a new method to enable the efficient learning of large-scale language models by optimizing the DGX-A100 environment and minimizing communication overhead while using the parallel processing method and model swapping technique suggested in previous studies.

3. Methods

3.1. Training Algorithm

We introduce an algorithm to overcome the GPU memory limitations when training transformer-based models such as BERT or GPT-2 on a DGX-A100 system. The proposed approach manages the entire model within the CPU and main memory. The entire model is divided into several sequential sub-models, and each sub-model is transferred to multi-GPU to be processed in parallel, performing partial forward, and backward propagation. The computed results on the GPU are then transferred back to the CPU, and the GPU memory is initialized. This method resolves the memory constraints of the GPU in large-scale language training.

3.1.1. Dividing and Parallelizing Transformer-Based Models

Transformer-based models are mainly composed of three types of layers: embedding layer, transformer layer, and unembedding layer. The model performs sequential computations in order, starting with one embedding layer, followed by multiple transformer layers and, finally, one unembedding layer. Each layer consists of a large number of parameters, and the count of these parameters can be further expanded using hyperparameters. Additionally, transformer layers can have longer sequence lengths. We manage individual layer types as sub-models, enabling flexible training with variable sequence lengths and focusing on multi-GPU parallelization optimized for individual layers. This makes it possible to respond flexibly to the expansion of layer parameters.

Each sub-model is composed of large-scale parameters that perform various roles. As the size of the parameters maintained for large-scale training expands, the capacity required to hold the parameters and the size of the intermediate computation results increase, making it impossible to maintain one sub-model within a single GPU memory. To solve this, we employ Tensor Parallelism. Tensor Parallelism equally divides all the parameters that constitute the sub-model among m GPUs. The set of parameters W_i that constitute the i-th sub-model is defined as follows:

$$W_i = \{w_{i1}, w_{i2}, w_{i3}, \ldots\}$$

where $w_{i1}, w_{i2}, w_{i3}, \ldots$ are evenly divided into m components corresponding to the row or column of the GPU. Each parameter's divided set is denoted as follows:

$$w_{ik} = \bigcup_{j=1}^{m} w_{ik}^{j}$$

where w_{ik}^{j} is the portion allocated to the j-th GPU of the k-th parameter of the i-th sub-model. Finally, the set of parameters W_{ij} allocated to the j-th GPU of the i-th sub-model is expressed as follows:

$$W_{ij} = \{w_{i1}^{j}, w_{i2}^{j}, w_{i3}^{j}, \ldots\}$$

Figure 1 illustrates the way the model is divided and parallelized as described above.

To apply parallel computation for the evenly divided individual parameters, we utilize the Tensor Parallelism technique used in Megatron-LM [32]. GPUs carry out computations on the divided parameters in parallel across the multi-GPU environment, and the final results can be obtained, identical to those without parallelization, via reduce or gather operations. During this process, communication overhead between GPUs may occur. The DGX-A100 is connected via NVLink and NVSwitch, enabling the rapid transfer of intermediate results generated during the reduce and gather processes, thereby minimizing communication overhead. Therefore, Tensor Parallelism allows for the easy and efficient expansion of the model by simply dividing the existing sub-model parameters and transmitting them to multi-GPU, without altering the original results.

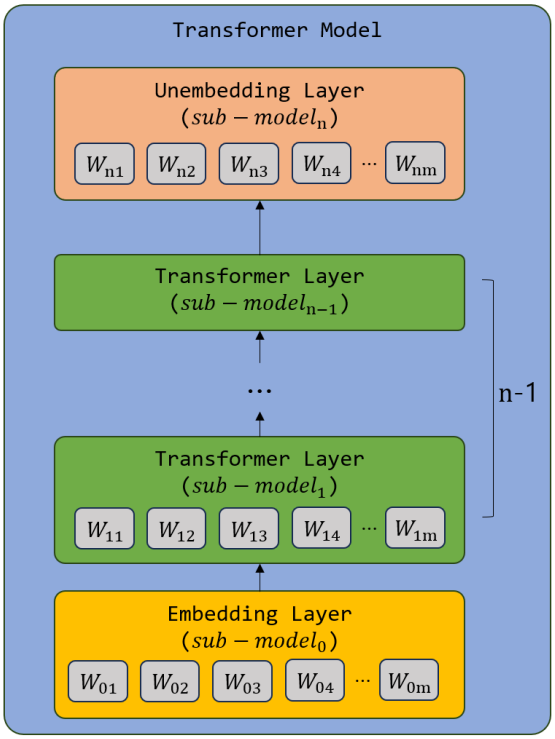

Figure 1. Division of the transformer-based model according to main components.

3.1.2. Heterogeneous Language Model Training

The proposed heterogeneous training consists of three major steps: forward propagation, backward propagation, and optimization. Each step proceeds sequentially for the divided sub-model. Forward propagation takes data batches as input and finally obtains a loss function. Backward propagation calculates the gradient for each parameter via backpropagation with the acquired loss value. Optimization uses a gradient to update parameters.

1. Forward Propagation
   Forward propagation propagates data input to sequential sub-models in order and acquires the final loss value. One sub-model is forward propagated across multiple GPUs. The context for forward propagation of the sub-model in the GPU consists of parameters and inputs. Before executing the sub-model, the context on the CPU side is requested and sent to the GPU. When the context is ready, it forwards and sends the sub-model result to the CPU. Sub-model results are reused in the backward propagation process. When the result is sent to the CPU, all memory allocated to the GPU is initialized. The purpose is to repeat this process to obtain the final loss value. If the given model has N sub-models, and we define the i-th sub-model as S_i, the output of the i-th sub-model as O_i, and the entire parameter set corresponding to the i-th sub-model as W_i. Then, the process of calculating the total loss value is as follows. (Here, O_0 is the initial input to the network, and Y is the label.)

$$O_i = S_i(O_{i-1}; W_i), for\ i = 1, 2, \ldots, N$$

$$Loss = CrossEntropy(Y, O_N)$$

2. Backward Propagation
   Backward propagation obtains the gradient for each model's parameters by backpropagating the loss value in the reverse order of the previously defined sub-models. One sub-model is backpropagated on multiple GPUs. The context needed to perform backward propagation of the sub-model in the GPU consists of the gradient computed

from the previous sub-model and the parameters of the current sub-model. Using the conditions utilized earlier in the forward propagation, the gradient with respect to the parameter set W_i corresponding to the i-th sub-model S_i is computed as follows:

$$\frac{\delta Loss}{\delta W_i} = \frac{\delta Loss}{\delta O_N} \frac{\delta O_N}{\delta O_{N-1}} \cdots \frac{\delta O_{i+1}}{\delta O_i} \frac{\delta O_i}{\delta O_{W_i}}$$

The sub-model requests a context on the CPU side to backpropagate the gradient and sends it to the GPU. To perform backpropagation, forward propagation must be conducted once for a given context to generate a computational graph. Therefore, in this process, when the context is transmitted to the GPU, forward propagation is performed once, and the gradient of the parameter is obtained by providing the gradient of the previous sub-model as an input to the generated computation graph. The calculated gradient is sent back to the CPU, and when the transfer is complete, all memory allocated to run the sub-model is released.

3. Optimization

Optimization proceeds using the AdamW optimizer with the i-th sub-model's parameters W_i and the computed gradient $\frac{\delta Loss}{\delta W_i}$. Optimization is performed using CPU multithreading for the AdamW operation [33,34]. Compared to forward or backward propagation, which have a computational complexity of $O(n^3)$, AdamW has a significantly lower computational complexity of $O(n^2)$. Therefore, even in a CPU multi-thread environment, computations can be completed in a shorter time compared to propagation operations. Moreover, by conducting optimization operations on the CPU and main memory, the GPU can focus on allocating memory for computation-intensive propagation operations.

Figure 2 summarizes the entire training process.

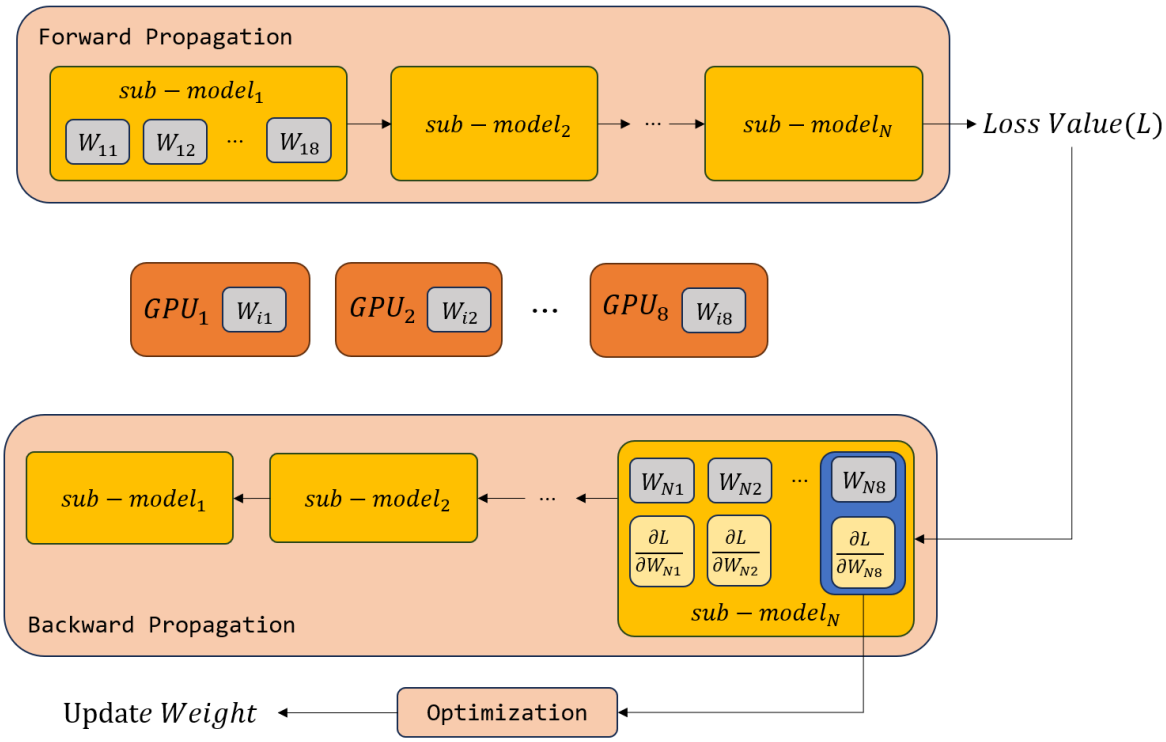

Figure 2. Overview of the three main phases in the training process.

3.2. Heterogeneous Training Management System (HTMS)

HTMS is a system that effectively manages the previously proposed heterogeneous training. The system manages the training process in four layers, and managers of various roles are placed in each layer to achieve the goal. The Supervision Layer initializes and manages manager objects of all layers [35]. It also monitors the status of all training processes and responds to errors. The context Management Layer manages resources and data for learning. The Task Management Layer requests Context Manager to execute tasks related to Context IO, composes tasks related to learning, and creates and executes Worker. The Worker Layer is a process group layer executed by the Task Management Layer. Figure 3 is a diagram of the layers and layer components that make up the entire training system.

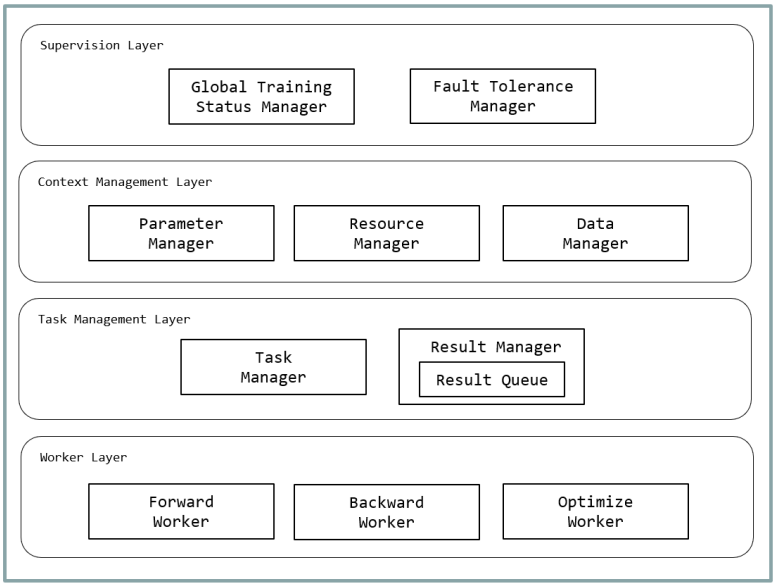

Figure 3. All layers and components of the Heterogeneous Training Management System (HTMS).

3.2.1. Supervision Layer

It is the layer that oversees the entire system. It is responsible for initialization, monitoring the training process, and responding to problems when they arise. This layer supports recording learning from a holistic view, managing to keep things normal, and communicating with users.

1. Global Training Status Manager
   The Manager initializes other Manager objects based on the user's initial settings to manage and monitor global training state information. Specifically, it includes two functions: logging and monitoring. Through logging, the Manager contributes to recording abnormal entries or outliers in a separate log file during the training process. It also systematically documents performance metrics such as loss values and model evaluations.
   The monitoring process supplies the user with real-time information, including trends in loss values, the scale of tensor values, and the probability distribution of feature values and output vectors. This process enables the user to either halt the training process as needed or to apply new hyperparameters during intermediate stages of training, allowing for the continuation of the training process.
   Furthermore, the Global Status Manager works in conjunction with the Fault Tolerance Manager to re-save the entire status for irrecoverable faults and enable subsequent training to be restored.
2. Fault Tolerance Manager
   The proposed system proceeds with a complex parallelized training process and can

cause various faults. We largely solve faults by dividing them into resource and algorithm aspects. Regarding resources, it responds when CPU, GPU, and memory computer resources are insufficient or unstable. The most common case is when other programs run on the system, and the available resources are low. The system detects a lack of resources and gradually reduces the batches of input sizes constituting training until normal training is possible. Alternatively, when some CPU cores or GPUs are outdated, overall system performance may be degraded. Computational resources that exceed the allowable time are excluded from the calculation. Regarding the algorithm, if the system diverges during the training process, the divergence point is identified and logged, and the current state is saved and terminated.

3.2.2. Context Management Layer

This is the layer that manages the learning context to proceed with training. The training context is divided into parameter, resource, and graph data, and a manager is placed to manage each. Corresponding managers manage all contexts in the CPU and sometimes issue a transfer task that transfers CPU data to the GPU to progress training on the GPU.

1. Parameter Manager
   Parameter Manager manages all parameters of the model in the CPU. When the manager receives a context request from the Task Management Layer, it divides the parameters corresponding to one sub-model. It issues an IO task that transmits to the multi-GPU. Furthermore, when the gradient optimization process is finished, the contents are reflected in the current parameters. Parameter updates are thread-safe against other asynchronous operations or calculations.
2. Resource Manager
   Resource Manager provides resources (CPU, GPU, main memory, and GPU memory) to execute tasks issued by various managers. The manager identifies available resources by referring to Global Training Status. It considers the currently requested and available resources, provides the resources if available, and only tasks that have received them can be executed. In particular, GPU memory is frequently allocated and deallocated due to the nature of the training algorithm, resulting in a large overhead. When memory of the same size is requested for performance optimization, the previously allocated memory is induced to be reused.
3. Data Manager
   Data Manager manages the input data for the model's forward propagation or backpropagation. The Manager has an input dataset for initial forward propagation input and issues IO tasks that organize them in batches and transfer them from the CPU to the GPU. To make transmission efficient, data are divided according to the number of GPUs, and the divided data are transmitted to multi-GPU parallelly. After that, the entire data are collected by gathering operation on multi-GPU. Through this, the overhead generated via input data transmission is minimized. In addition, the forward propagation result for the model is required to backpropagate the same data to multi-GPU or sequential partial models on the GPU. The Manager issues an IO task that stores the results from forward propagation and sends them to the GPU upon request.

3.2.3. Task Management Layer

Algorithms related to training are executed in units of sub-models, and the results are received and transmitted to the managers of the Context Management Layer.

1. Task Manager
   Task Manager issues one task among forward propagation, backward propagation, and optimization for one sub-model. Propagation runs on multi-GPU, and optimization runs on multi-thread. Since an execution context is required to execute a task, it is requested in the Context Management Layer. When the execution context is ready, the

manager places parallelized tasks and waits until all the arranged tasks are finished. When all tasks are executed, task completion is notified via the Result Manager, the result is transmitted to the CPU, and no longer needed resources are returned.

2. Result Manager
   The Result Manager issues an IO task that transfers the results generated using the results of each sub-model from the GPU to the CPU. Furthermore, it operates a Result Queue that stores results asynchronously, calculates information in the Queue, and transmits it to the Context Management Layer.
3. Worker Layer
   In the Worker Layer, training tasks related to the assigned execution are arranged and executed. In this layer, only one propagation and optimization task can run concurrently. One task consists of several parallelized jobs running in multi-process or multi-thread. Several jobs work by forming a group, and the jobs within the group exchange or collect calculation results via MPI.

3.3. Management System Workflow

Management system managers are involved in various execution flows and compose the execution workflow. Workflow belongs to one of the two states of forward, backward, and optimization at execution time. Forward, backward, and optimization states proceed iteratively. The two states carry out actual training tasks, which are placed in three streams (IO, GPU Execution, and CPU Execution) according to the resource type. In the IO stream, communication tasks between the CPU and GPU are arranged; in the GPU execution stream, multi-GPU propagation tasks; and in the CPU, optimization tasks executed in multi-thread are arranged. The management system asynchronously arranges tasks in each stream using managers of each layer and synchronizes each task by events. Furthermore, we utilize the Double-Buffering Concept to minimize the waiting time of GPU resources due to IO tasks. We introduce how the managers of the system place tasks in the stream according to the state and how the corresponding tasks are synchronized.

3.3.1. Doubled Context Buffer

In large-scale language learning, large-capacity communication occurs between CPU and GPU. The computationally intensive GPU can become idle if the context transfer is incomplete, causing training inefficiency. Therefore, to minimize the idle time of the GPU, the Doubled Context Buffer technique is used for learning. The technique is used to declare two context buffers in the GPU in advance. The IO task that receives the context necessary for the next GPU calculation is simultaneously executed during the GPU calculation process. GPU execution and IO tasks can proceed independently, and the technique can minimize idle time. Through experiments, we suggest that idle time can be minimized and buffer space can be used effectively when two buffers are maintained.

3.3.2. Forward State

Forward State is a state in which sub-models are sequentially executed, and the loss value is finally acquired. To this end, three tasks (Context Preparation, Sub-model Forward Propagation, and Result Transfer) are sequentially performed on sub-models and repeated for each sub-model.

Context Preparation requests context transfer to GPU to execute sub-model on CPU on multi-GPU. The Resource Manager allocates a memory buffer to the multi-GPU to transfer the context to the GPU. There are two types of GPU memory buffers: parameter buffers and model input buffers. After the buffer allocation, Parameter Manager and Data Manager transfer parameters and inputs to the GPU buffer space. Context Preparation refers to all of these tasks, and when the task is finished, it enters a ready-to-execute state. Context Preparation is placed in IO stream. When ready for execution, an execution request event is sent to the system.

Sub-model Forward Propagation is a task executed via the execution request event issued after Context Preparation is completed. It executes tasks on context deployed on multi-GPU. This is placed in the GPU Execution Stream, which is the execution flow of multi-GPU. Task Manager executes worker process tasks parallelized on multi-GPU, and each process collects the generated results. At this time, only one forward execution is always in progress at the same time. A job completion event is sent to the system when the job is complete.

Result Transfer sends the execution results of workers to the Result Manager. Result Manager processes the data of Queue and updates other Manager objects. Furthermore, if necessary, recover deployed resources. In the forward state, the output of the sub-model is used as the input of backward propagation, so the calculation result of the sub-model is transmitted to the Result Manager. Since Result Manager receives input from multiple GPUs, it receives data via an asynchronous Result Queue. The transfer operation is placed on the IO Stream. Furthermore, if the next sub-model to be executed is a model with the same structure as the previous one, the previously allocated buffer space is reused without initialization. Through this, overhead due to allocation can be minimized. Figure 4 shows the schedule in which tasks are placed in the IO stream and GPU stream when the system enters the forward state.

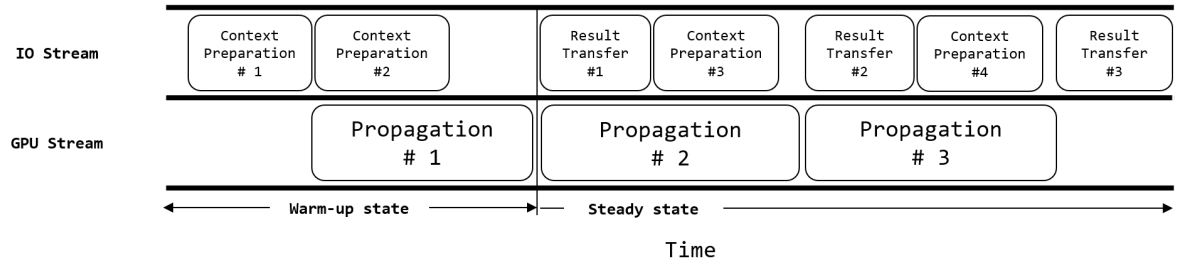

Figure 4. Forward state execution workflow.

3.3.3. Backward and Optimization State

The Backward State can proceed if the loss value is acquired in the Forward State. Gradient, a parameter update index for each sub-model, is acquired when proceeding with Backward State. When the gradient is transferred to the CPU, the optimization operation is performed independently of the next backward operation using multi-threads in the CPU. Through this, Backward work and Optimization work proceed simultaneously. State includes 3-task-type Forward State and additional sub-model optimization execution task.

Context Preparation work proceeds in the same way as Forward State. However, for backward operation, Data Manager additionally transmits the results generated in the forward process. As a result, the context transmission time in the backward process is greatly increased.

Sub-model backward execution performs backward propagation for a given sub-model. To perform backpropagation on a sub-model, forward calculation results for the model's computation nodes are required. Therefore, it is necessary to move forward again using the sub-model and the forward input result of the previous step and form an intermediate calculation context. When the result of the intermediate operation node is completed, the backward process proceeds. The system is notified via an event when the backward process is completed. Result Transfer proceeds in the same way as in Forward State.

Sub-model optimization is executed via the transmission completion event that occurs when the backward process is finished, and the gradient is transmitted to the CPU. When the task is executed, the gradient is taken from the Result Queue in the Result Manager, and the AdamW optimization calculation, which calculates how many parameters need to be updated, is executed. This is divided and executed on multi-thread and is carried out in CPU Execution Stream, which is a multi-thread workflow. When the calculation is complete,

Parameter Manager reflects it. Figure 5 shows the schedule in which tasks are placed in IO, GPU, and CPU streams when the system enters the backward and optimization state.

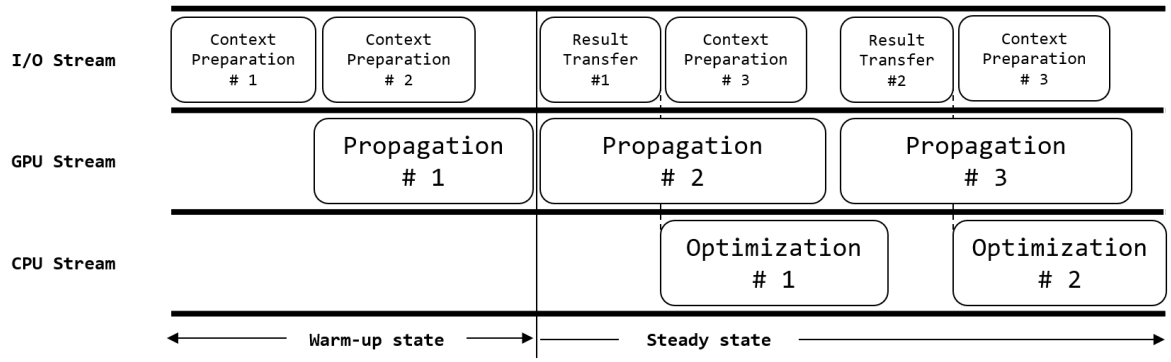

Figure 5. Backward state execution workflow.

4. Experiments

4.1. Experiment Setup

The proposed system divides a transformer-based language model into sub-models, declares them on the CPU, and sequentially loads and divides them onto multi-GPU for execution. At this time, the transformer layer is the core part of the language model, having more parameters and a higher computational complexity than other layers. We provide a scalability experiment that measures the size of the transformer layer that the proposed system can load onto the GPU and, accordingly, the maximum batch size that can be trained. Table 1 defines four sizes to be used in experiments to measure the scalability and time of transformer layers of various sizes.

Table 1. Transformer layer size criteria.

	Hidden	Nlayers	Dhead	Nheads
Base	768	12	96	8
Large	1536	24	96	16
XLarge	4096	32	128	32
XXLarge	8192	32	128	64

Additionally, we show that the scheduling method we provide hardly generates any GPU idle time. Conventional methods inevitably enter the GPU into an idle state in the process of loading contexts onto the GPU. However, by utilizing our proposed Doubled Context Buffer method and asynchronous scheduling, the context transfer can be completed before performing propagation operations on the GPU. We measure various types of compute time and communication time that constitute the entire training and analyze and provide the occurrence of idle time under the previously proposed transformer layer and batch size configuration.

The proposed experiments were performed on the C4 dataset [36] (Colossal Clean Crawled Corpus), and the individual texts inside were composed into 512-long token sequences using the Tokenizer used in Megatron-LM. The C4 dataset, collected from various web sources and preprocessed and refined, provides noise-minimized data that can be directly used for model training. Furthermore, the entire experiment was conducted on the DGX-A100 system, loaded with 8 NVIDIA-A100 Tensor Core GPU 40 GB.

The training parameters were set as follows to guide the training process. The learning rate was initialized at 0.001 and gradually reduced to 0.00005. The learning scheduling was executed in 20 stages throughout the entire training process, employing the StepLR technique to linearly decrease the learning rate. Each training stage consisted of randomly

extracting one million samples from the C4 corpus and iterating through three epochs. The optimization algorithm was implemented using AdamW, with the hyperparameters set to beta1 = 0.9 and beta2 = 0.999.

4.2. Scalability Test

The proposed system can train a language model of a size that could not be trained in the past or can train a very large batch size for a small language model. Figure 6 illustrates a comparison between our proposed method and the existing layer-based Model Parallelism, highlighting the maximum configurable batch sizes for different layer sizes (Base = 2048, Large = 1024, XLarge = 378, and XXLarge = 128 for our proposed layer; Base = 168, Large = 32, XLarge = 12, and XXLarge = 4 for layer-based Model Parallelism). This approach allows for greater flexibility in configuring the Double-Buffered Context and provides a detailed measure of how the new method performs in comparison to existing techniques.

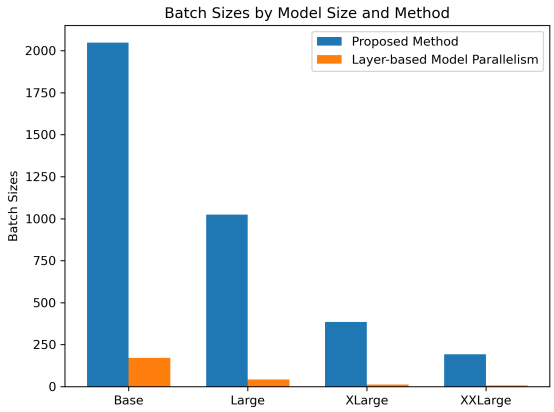

Figure 6. Comparison of maximum configurable batch sizes for proposed layer sizes and layer-based Model Parallelism.

Our experimental results demonstrate a model scalability exceeding tenfold than of the traditional layer-based Model Parallelism. Furthermore, our approach allows for the training of XLarge and XXLarge models, which were untrainable with the existing layer-based Model Parallelism. The batch sizes per layer proposed in our method reserve approximately 35 GB of GPU memory space, and the remaining space is sufficient to adequately store the next context for double buffering.

4.3. Computation and Communication Analysis

We accurately measure various computation and communication times needed to train a configured language model with specific batch sizes and proposed layer sizes. This measurement serves as a benchmark for calculating the total time required for training. Additionally, by aggregating the measured computation and communication times for each block size and the proposed batch size, we demonstrate that the proposed scheduling scheme has almost no GPU idle time due to waiting for context transfers. First of all, forward propagation, backward propagation, and optimization time, which are the main computational tasks in the training process, are measured for the batch size for the major layer size presented above and presented in Figure 7.

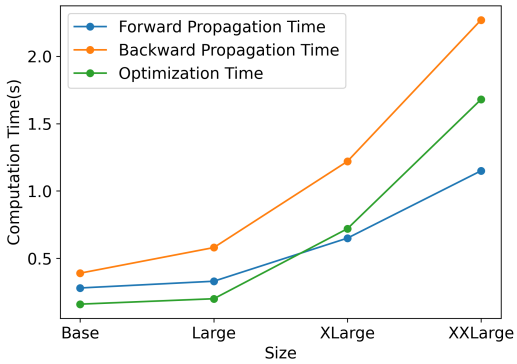

Figure 7. Time measurement for each proposed layer size calculation type.

In terms of computation, we observed that as the layer size increases, the time for forward computation, backward computation, and optimization all increase for the same input and output capacity. Specifically, the forward and backward computations substantially increase the computation time as the layer size grows. This is because the self-attention mechanism, a significant component of the transformer-based model, causes the computation volume to increase quadratically with the length of the input vector. In other words, if the length of the input vector doubles, the number of relationships that need to be processed increases fourfold. This is due to the necessity of computing the attention score for all pairs of input vectors. Therefore, we can observe that the computational complexity of the transformer models increases quadratically with the length of the input vector. In particular, the backward computation is more computationally intensive than the forward, thus increasing at a greater rate, while optimization is computed using multi-threading on the CPU side, it proceeds faster than computation on the multi-GPU. This is because its computational complexity is significantly lower compared to GPU computation, and there is no computational overhead due to communication. Additionally, we conducted computation experiments to predict the total computation time when composing various batch sizes, and we present the results. Figure 8 shows the calculation time measurement results for each batch size for the major layer sizes.

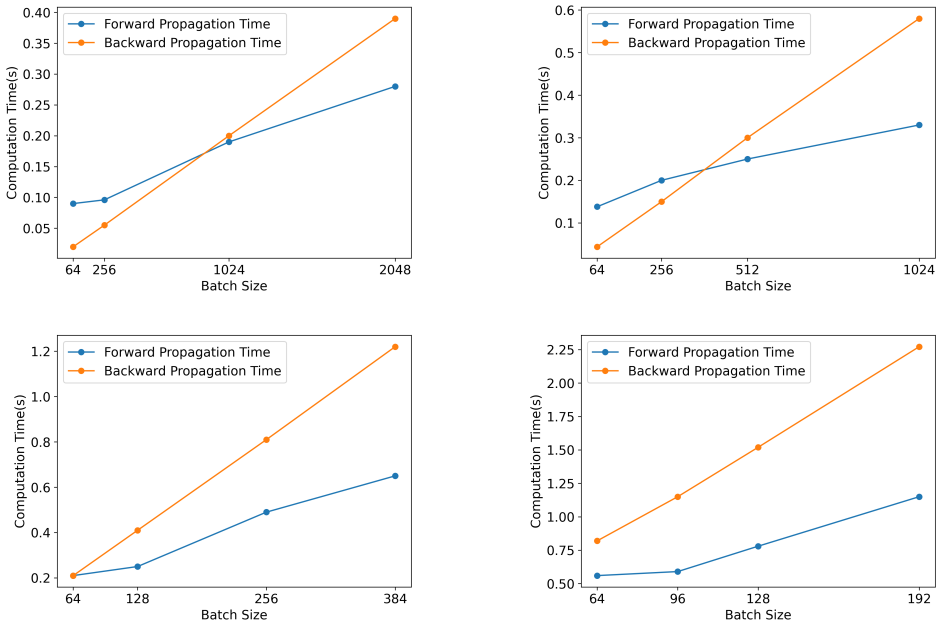

Figure 8. Time measurement for each batch size of the proposed layer size (Base (**top left**), Large (**top right**), XLarge (**bottom left**), and XXLarge (**bottom right**)).

The GPU computation task we propose can proceed once the context is prepared from the CPU to the GPU. The context preparation time mainly consists of three tasks: input transfer time, parameter transfer time, and buffer allocation time. We measured the total context preparation time and the times for the individual tasks that compose it. For the input transfer time, because the transfer time is very short in the forward operation, we measured it based on the hidden state transfer time required to construct the backward context.

According to Figure 9, the most time-consuming task in preparing the context is Buffer Allocation Time. This is realized using memory allocation operations in CUDA, which take a significant amount of time. Our proposed Management System is designed to reuse the existing memory when conducting the same type of computation as the previous layer. Additionally, the time taken to transfer parameters increases only slightly, even as the model size grows. This is because the bandwidth performance of the DGX-A100 System is high compared to the capacity of the parameters that need to be transferred, and the parameters to be transferred are distributed to multiple GPUs. Consequently, as the model size increases, the time to prepare the context does not significantly increase compared to computation time. Therefore, the time it takes to perform the forward and backward computations determines the system's overall performance.

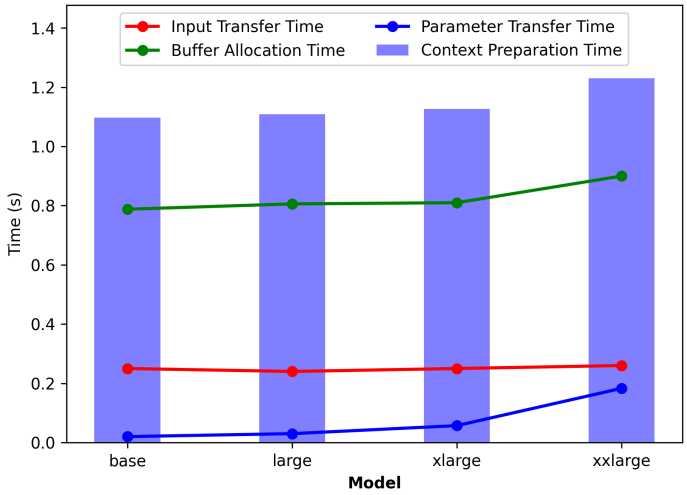

Figure 9. Time measurement for each proposed layer size communication type.

Considering the computation aspects of Figures 7–9 presented above and the communication time measurement results, our proposed schedule method hardly generates GPU idle time. The proposed training schedule batches Context Preparation and Result Transfer operations in the IO stream. It repeats Forward or Backward operations in the GPU Stream once it enters the Steady State. GPU idle time occurs when the sum of the execution times of Context Prepare and Result Transfer exceeds the sum of the execution times of Forward or Backward operations. The proposed layer and batch sizes hardly result in idle time during Forward and Backward operations. However, composing a massive batch in a base model or smaller leads to GPU idle time. Finally, under the conditions of implementing the existing 8-Way Tensor Parallelism, we can implement the existing training at a similar level, with several times fewer nodes than the existing ones.

5. Conclusions

In this study, we proposed a new training algorithm that partitions and parallelizes models using heterogeneous resources. This algorithm overcomes the GPU resource limitations of a single DGX-A100 system, contributing to the increase in the size of trainable language models and the growth of batch sizes. We also designed and implemented a man-

agement system to efficiently run this algorithm. This system can systematically manage the parallelized training process, resource usage, and batch tasks asynchronously. Lastly, we proposed a technique that enables effective job scheduling in a complex heterogeneous learning environment. This approach has been validated using performance measurements and demonstrates that training can be conducted with larger models and batch sizes even within limited resources.

As a result of this study, we showed that we can overcome the limitations of GPU resources by partitioning and parallelizing models using heterogeneous resources. Furthermore, the management system that we designed and implemented for this purpose has demonstrated its ability to systematically manage the parallelized learning process and resource usage and to batch tasks asynchronously.

Additionally, by introducing techniques that enable effective job scheduling in a complex heterogeneous learning environment, we confirmed that the efficiency and effectiveness of model training could be enhanced. These findings enable researchers to proceed with training tasks with larger models and batch sizes even with limited resources.

This study has been validated using performance measurements, and the results demonstrate how the management system and scheduling techniques effectively work. This shows how our algorithm and management system enhance large-scale language model training in a heterogeneous computing environment. Through this, the researchers confirmed the ability to effectively conduct training tasks with larger models and batch sizes with limited resources.

This study presents a new methodology that maximizes the use of heterogeneous computing resources to overcome the constraints of model training on a single DGX-A100 system. Our management system and scheduling techniques systematically manage the parallelized learning process and enable job scheduling in complex learning environments using heterogeneous resources. This methodology allows researchers to efficiently use limited resources to proceed with training larger models and batch sizes.

In conclusion, our research has pioneered the use of heterogeneous computing resources to significantly expand the feasible layer size within a single DGX-A100 system. As we look to the future, we intend to further our research by implementing a system capable of universally applying to various AI models, allowing automatic optimization for parallelization and scheduling. Additionally, we aim to explore additional learning techniques that enable the expansion of horizontal layer sizes, further unlocking scalability for more complex models. These forward-looking strategies mark a substantial step towards a more versatile and efficient paradigm for large-scale AI model training.

Author Contributions: Conceptualization, K.-H.K. and C.-S.J.; methodology, K.-H.K. and C.-S.J.; software, K.-H.K.; validation, K.-H.K. and C.-S.J.; formal analysis, K.-H.K. and C.-S.J.; investigation, K.-H.K.; resources, K.-H.K.; data curation, K.-H.K.; writing—original draft preparation, K.-H.K.; writing—review and editing, K.-H.K. and C.-S.J.; visualization, K.-H.K.; supervision, C.-S.J.; project administration, K.-H.K. and C.-S.J.; funding acquisition, C.-S.J. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available upon request from the corresponding author.

Acknowledgments: This work was supported by Artificial intelligence industrial convergence cluster development project funded by the Ministry of Science and ICT (MSIT, Korea) & Gwangju Metropolitan City.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. *Adv. Neural Inf. Process. Syst.* **2017**, *2017*, 5999–6009.
2. Mars, M. From Word Embeddings to Pre-Trained Language Models: A State-of-the-Art Walkthrough. *Appl. Sci.* **2022**, *12*, 8805. [CrossRef]
3. Garrido-Muñoz, I.; Montejo-Ráez, A.; Martínez-Santiago, F.; Ureña-López, L.A. A survey on bias in deep NLP. *Appl. Sci.* **2021**, *11*, 3184. [CrossRef]
4. Radford, A.; Narasimhan, K.; Salimans, T.; Sutskever, I. Improving language understanding by generative pre-training. *OpenAI*, 2018. Available online: https://www.mikecaptain.com/resources/pdf/GPT-1.pdf (accessed on 10 August 2023).
5. Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language models are few-shot learners. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 1877–1901.
6. Nvidia Corporation. *NVIDIA DGX A100 | DATA SHEET*; Nvidia Corporation: Santa Clara, CA, USA, 20 May 2020.
7. Rajbhandari, S.; Ruwase, O.; Rasley, J.; Smith, S.; He, Y. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, MO, USA, 14–19 November 2021; pp. 1–14.
8. Ren, J.; Rajbhandari, S.; Aminabadi, R.Y.; Ruwase, O.; Yang, S.; Zhang, M.; Li, D.; He, Y. {ZeRO-Offload}: Democratizing {Billion-Scale} model training. In Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC 21), Virtual, 14–16 July 2021; pp. 551–564.
9. Weng, J.; Lin, M.; Huang, S.; Liu, B.; Makoviichuk, D.; Makoviychuk, V.; Liu, Z.; Song, Y.; Luo, T.; Jiang, Y.; et al. Envpool: A highly parallel reinforcement learning environment execution engine. *Adv. Neural Inf. Process. Syst.* **2022**, *35*, 22409–22421.
10. Chen, T.; Xu, B.; Zhang, C.; Guestrin, C. Training deep nets with sublinear memory cost. *arXiv* **2016**, arXiv:1604.06174.
11. Gupta, A.; Berant, J. Gmat: Global memory augmentation for transformers. *arXiv* **2020**, arXiv:2006.03274.
12. Rajbhandari, S.; Rasley, J.; Ruwase, O.; He, Y. Zero: Memory optimizations toward training trillion parameter models. *IEEE Comput. Soc.* **2020**, *2020*, 11. [CrossRef]
13. Choi, H.; Lee, J. Efficient use of gpu memory for large-scale deep learning model training. *Appl. Sci.* **2021**, *11*, 377. [CrossRef]
14. Liu, Y.; Ott, M.; Goyal, N.; Du, J.; Joshi, M.; Chen, D.; Levy, O.; Lewis, M.; Zettlemoyer, L.; Stoyanov, V. Roberta: A robustly optimized bert pretraining approach. *arXiv* **2019**, arXiv:1907.11692.
15. Harlap, A.; Narayanan, D.; Phanishayee, A.; Seshadri, V.; Devanur, N.; Ganger, G.; Gibbons, P. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv* **2018**, arXiv:1806.03377
16. Narayanan, D.; Shoeybi, M.; Casper, J.; LeGresley, P.; Patwary, M.; Korthikanti, V.; Vainbrand, D.; Kashinkunti, P.; Bernauer, J.; Catanzaro, B.; et al. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. *IEEE Comput. Soc.* **2021**, *11*, 1–15. [CrossRef]
17. Huang, Y.; Cheng, Y.; Bapna, A.; Firat, O.; Chen, D.; Chen, M.; Lee, H.; Ngiam, J.; Le, Q.V.; Wu, Y.; et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In Proceedings of the 33rd International Conference on Neural Information Processing Systems (NIPS'19), Vancouver, BC, Canada, 8–14 December 2019.
18. Li, Z.; Zhuang, S.; Guo, S.; Zhuo, D.; Zhang, H.; Song, D.; Stoica, I. Terapipe: Token-level pipeline parallelism for training large-scale language models. In Proceedings of the International Conference on Machine Learning. PMLR, Online, 18–24 July 2021; pp. 6543–6552.
19. Bian, Z.; Xu, Q.; Wang, B.; You, Y. Maximizing parallelism in distributed training for huge neural networks. *arXiv* **2021**, arXiv:2105.14450.
20. Shazeer, N.; Cheng, Y.; Parmar, N.; Tran, D.; Vaswani, A.; Koanantakool, P.; Hawkins, P.; Lee, H.; Hong, M.; Young, C.; et al. Mesh-tensorflow: Deep learning for supercomputers. In Proceedings of the 32rd International Conference on Neural Information Processing Systems (NIPS'18), Montréal, BC, Canada, 13–16 December 2018.
21. Song, L.; Chen, F.; Zhuo, Y.; Qian, X.; Li, H.; Chen, Y. AccPar: Tensor partitioning for heterogeneous deep learning accelerators. In Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), San Diego, CA, USA, 22–26 February 2020. [CrossRef]
22. Liang, P.; Tang, Y.; Zhang, X.; Bai, Y.; Su, T.; Lai, Z.; Qiao, L.; Li, D. A Survey on Auto-Parallelism of Large-Scale Deep Learning Training. *IEEE Trans. Parallel Distrib. Syst.* **2023**, *34*, 2377–2390. [CrossRef]
23. Fu, Q.; Chukka, R.; Achorn, K.; Atta-fosu, T.; Canchi, D.R.; Teng, Z.; White, J.; Schmidt, D.C. Deep Learning Models on CPUs: A Methodology for Efficient Training. *arXiv* **2022**, arXiv:2206.10034.
24. Dean, J.; Corrado, G.; Monga, R.; Chen, K.; Devin, M.; Mao, M.; Ranzato, M.; Senior, A.; Tucker, P.; Yang, K.; et al. Large scale distributed deep networks. In Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS'12), New Orleans, LA, USA, 3–6 December 2012.
25. Zhang, H.; Huang, H.; Han, H. A novel heterogeneous parallel convolution Bi-LSTM for speech emotion recognition. *Appl. Sci.* **2021**, *11*, 9897. [CrossRef]
26. Shin, W.; Yoo, K.H.; Baek, N. Large-Scale data computing performance comparisons on sycl heterogeneous parallel processing layer implementations. *Appl. Sci.* **2020**, *10*, 1656. [CrossRef]

27. Choukse, E.; Sullivan, M.B.; O'connor, M.; Erez, M.; Pool, J.; Nellans, D.; Keckler, S.W. Buddy Compression: Enabling Larger Memory for Deep Learning and HPC Workloads on GPUs. In Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), Valencia, Spain, 30 May–3 June 2020; pp. 926–939. [CrossRef]
28. Li, Y.; Phanishayee, A.; Murray, D.; Tarnawski, J.; Kim, N.S. Harmony: Overcoming the Hurdles of GPU Memory Capacity to Train Massive DNN Models on Commodity Servers. *VLDB Endow.* **2022**, *15*, 2747–2760. [CrossRef]
29. Zou, D.; Jin, X.; Yu, X.; Zhang, H.; Demmel, J. Computron: Serving Distributed Deep Learning Models with Model Parallel Swapping. *arXiv* **2023**, arXiv:2306.13835.
30. Shoeybi, M.; Patwary, M.; Puri, R.; LeGresley, P.; Casper, J.; Catanzaro, B. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv* **2019**, arXiv:1909.08053.
31. Liu, J.; Wu, Z.; Feng, D.; Zhang, M.; Wu, X.; Yao, X.; Yu, D.; Ma, Y.; Zhao, F.; Dou, D. HeterPS: Distributed deep learning with reinforcement learning based scheduling in heterogeneous environments. *Future Gener. Comput. Syst.* **2023**, *148*, 106–117. [CrossRef]
32. Jain, A.; Moon, T.; Benson, T.; Subramoni, H.; Jacobs, S.A.; Panda, D.K.; Essen, B.V. SUPER: SUb-graph parallelism for TransformERs. In Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Portland, OR, USA, 17–21 May 2021; pp. 629–638. [CrossRef]
33. Zinkevich, M.; Weimer, M.; Li, L.; Smola, A. Parallelized stochastic gradient descent. In Proceedings of the 23th International Conference on Neural Information Processing Systems (NIPS'10), Vancouver, BC, Canada, 6–9 December 2010.
34. Kennedy, R.K.; Khoshgoftaar, T.M.; Villanustre, F.; Humphrey, T. A parallel and distributed stochastic gradient descent implementation using commodity clusters. *J. Big Data* **2019**, *6*. [CrossRef]
35. Kim, Y.K.; Kim, Y.; Jeong, C.S. RIDE: Real-time massive image processing platform on distributed environment. *EURASIP J. Image Video Process.* **2018**, *2018*, 39. [CrossRef]
36. Raffel, C.; Shazeer, N.; Roberts, A.; Lee, K.; Narang, S.; Matena, M.; Zhou, Y.; Li, W.; Liu, P.J. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.* **2020**, *21*, 5485–5551.