

NNTile: A Machine Learning Framework Capable of Training Extremely Large GPT Language Models on a Single Node

Aleksandr Mikhalev^[0000-0002-9274-7237], Aleksandr Katrutsa^[0000-0002-7050-1615],
Konstantin Sozykin^[0000-0003-3321-7399], and Ivan Oseledets^[0000-0003-2071-2163]

1 Introduction

Over the last few years, the global market of artificial intelligence (AI) has grown significantly. The trend is still going on: judging by [MarketsAndMarkets.com](#), global AI market value was estimated at \$150.2 billion in 2022 [1] with a projected value of \$1345.2 [1] billion by 2030 and a Compound Annual Growth Rate (CAGR) of 35.8%. The main reason for such a rise is a “Generative AI”, backed by transformers neural network architecture [16]. Corresponding to [10], one just puts more parameters into a transformer, feeds it more data, and spends more computing power to improve outcomes. The optimal amount of data and compute for a model of a given size is studied in [8].

Just like any other neural network architecture nowadays, transformers are pre-trained on thousands of GPUs, while CPUs are heavily underutilized, and CPU RAM is not taken into account at all. Here’s a simple example: a single DGX-A100 workstation contains 8 Nvidia A100 GPUs with 80GB VRAM and 2 TB of CPU RAM. Popular solutions for distributed training of neural networks, e.g., DeepSpeed [13, 4] or Megatron-LM [15, 11] or MegaScale [9] or PyTorch [12] FSDP [18], are limited by an aggregated 640GB of VRAM in such a case because offloading to CPU RAM causes synchronizations and, as a result, a significant loss of a training throughput. A comprehensive survey on the main approaches to distributed training of large models is presented in [7]. However, if training requires more than 640 GBs of aggregated

Aleksandr Mikhalev
Skoltech, Moscow, e-mail: al.mikhalev@skoltech.ru

Aleksandr Katrutsa
Skoltech, Moscow and AIRI, Moscow, e-mail: amkatrutsa@gmail.com

Konstantin Sozykin
Skoltech, Moscow, e-mail: ko.sozykin@skoltech.ru

Ivan Oseledets
AIRI, Moscow, and Skoltech, Moscow, e-mail: oseledets@airi.net

memory, then either training does not run at all or overall performance drops drastically. In the case of a cluster of GPUs, if aggregated VRAM is not enough, then there is only one solution – to get more GPUs to avoid the performance drop. Without a doubt, getting more GPUs for the pretraining phase to digest terabytes of data is the way to go, and at the first glance, there is no trouble at all. However, such a memory wall defines a minimal number of GPUs for all other phases related to training: searching for training hyperparameters and fine-tuning for downstream tasks, especially if a model supports very long sequences of data. Acquiring thousands of GPUs just to check training loss behaviour at another value of a learning rate might postpone a model release date heavily.

Although pretraining a transformer model from a scratch or a checkpoint does not suffer the aforementioned memory wall, average model floating point operations (model FLOPs) per GPU barely reach 55% of the theoretical peak. For example, Megatron-LM reports training efficiency for a model with 530 billion parameters at 40.4% on 2240 GPUs, at 38.8% on 2800 GPUs, and at 36.2% on 4480 GPUs in [15] and at 56% on 280 GPUs in [11]. Technical report [3] on training a Falcon model with 40 billion parameters along with its model card on HuggingFace ¹ states that 2800 PFlops-days were digested on 384 Nvidia A100 GPUs through around two months, which translates into approximately 40% efficiency. Recent paper [9] claims to get 54.3% efficiency on training 530-billion parameter model on 11200 GPUs and 55.2% efficiency on training 175-billion parameter model on 12288 GPUs, while Megatron-LM achieves only 48.2% and 41.2% respectively.

In this paper, we present a non-standard approach to train neural networks. To the best of our knowledge, this is the first study of a task-based parallel programming paradigm applied to train transformer-based models. Such a paradigm distributes computations among available resources dynamically and prefetches and offloads data asynchronously. It is described in more details section 2. Such a paradigm shift allows us to train custom GPT2 models with up to 50 billion parameters on a single server. In the meantime, a popular FullyShardedDataParallel (FSDP) approach of PyTorch allows only training of models up to 8 billion parameters while all other settings are equal.

The main contributions are listed below.

1. The brief introduction to the task-based parallelism and StarPU [5] library is presented.
2. The implementation details of the key elements of deep neural networks in a task-based paradigm are discussed.
3. The first experimental results on using task-based parallelism for training the GPT2 model in the single node are shown.

¹ <https://huggingface.co/tiiuae/falcon-40b>

2 Task-based parallelism

We consider a computation process as a sequence of operations and the necessary data for it. It is sometimes convenient to split large operations into smaller tasks that operate on pieces of data. For example, it is natural to split linear algebra operations, like matrix multiplication or factorization, into their corresponding block versions. Block versions split the entire data into chunks, referred to as *tiles*. The entire operations are correspondingly split into a set of suboperations such that each suboperation accesses only several tiles instead of entire tensors. Such suboperations with a limited data access are referred to as *tasks*. The simple example is a matrix multiplication operation that can be split into multiple tasks such that every task performs matrix multiplication on the corresponding submatrices. Then the results of tasks are aggregated in the result of the original matrix multiplication operation. The considered tasks are combined into a directed acyclic graph (DAG), where nodes represent tasks and directed edges correspond to dependencies. The smaller the tiles are, the more tasks are present in the DAG. As a consequence, more tasks can be executed in parallel, but the arithmetic intensity, along with the performance of each task, may drop. Once the DAG of tasks is formed, a scheduler comes into play and maps each task onto available computational resources dynamically during runtime. The scheduler also transfers necessary tiles to and from designated devices asynchronously. Further, we cover all the main ingredients of the task-based parallel programming paradigm: task submission, memory management and task scheduler.

Task submission. Our work, described in the current paper, is based on StarPU [5] runtime system with its scheduling library. The StarPU relies on a Sequential Task Flow (STF) approach, which inserts tasks into the graph (DAG) one by one during runtime. This approach emulates the *sequential* code execution, while the tasks are actually executed *asynchronously* due to their data dependencies. Such an emulation significantly simplifies the reading and modifying of the source code.

Memory management. There are two ways to work with the data: offload it to a GPU from the main memory for every single task or hold it on a GPU while it is still needed. The offloading approach proved to be easy to implement and useful in a case of standard dense linear algebra applications. However, neural networks training workflow suffers from a much low arithmetic intensity. It bottlenecks the offloading approach by a CPU-GPU communication bus. Therefore, we use a memory manager that preserves tiles on a GPU if needed for further tasks. Tiles can be asynchronously copied to another GPU if necessary. The StarPU library supports such a memory management system, which we rely on in our numerical experiments.

Schedulers. The next ingredient of the framework for task-based parallelism is the scheduler, which decides which device runs the current task and how to minimize the total runtime based on data communication costs, task complexity, and device availability. Simple yet effective greedy heuristics proved to fit standard dense linear algebra applications through the StarPU and Chameleon [2] software. Such heuristics schedule tasks one by one only taking into account already scheduled tasks without looking into the forthcoming remaining graph. The main opportunity for such heuristics to shine is within an incredibly high arithmetic intensity of the

overall problem. Although training neural networks is mostly memory bound, the entire graph of tasks is made of many repeated training iteration. Such an assumption allows to develop a so-called graph scheduling policy, which takes into account entire graph to make scheduling decisions.

3 GPT2 model training with NNTile.

To illustrate the efficiency of our approach, we train custom GPT2 models with an increasing number of parameters and compare the maximum feasible model size with PyTorch in FSDP mode. We re-implement all the necessary operations to train the GPT2 model in the tile-based form within the *NNTile* machine learning framework and list them below. Due to page limitation, we describe only forward passes through layers of the GPT2 model, although operations for gradient propagation from the last layer to the first one are also implemented in NNTile.

Custom GPT2 models configuration. Unfortunately, training standard pre-trained GPT2 models of a small size, available at the Huggingface portal ², did not scale from a single GPU to 8 GPUs on a single node in a data parallel training regime. Calculation of gradients of model parameters becomes a bottleneck in such a case. An obvious solution is to support a reduction: gradients of the same matrix of parameters from different devices are accumulated. However, enabling reduction with a help of StarPU data access mode STARPU_REDUX made things even worse. Greedy scheduling policies fail to get good scaling in such a scenario. Therefore, it was decided to train custom GPT2 models with just few transformer blocks but with large embedding size to split entire data into tiles across embedding dimension. Our numerical examples proved that standard StarPU scheduling policies are more or less good fit for such a *tensor* parallelism.

Embedding layer. This layer processes the input matrix of size $N_s \times N_b$, where N_b denotes batch size and N_s denotes the sequence length and equips every tokenized item from a sequence with an embedding vector. The tokenized item of the sequence corresponds to the map of a word from a sequence given by the selected tokenizer. The resulting three-dimensional tensor is of the size $N_e \times N_s \times N_b$, where N_e denotes the embedding dimension. So, this layer has $N_v \times N_e$ parameters, and N_v is the vocabulary size aligned with the tokenizer.

Nonlinear activations. Most nonlinear activation functions used in deep neural networks are elementwise transformations of the input data. Therefore, they perfectly fit the tile representation of the tensor used in our framework. Low-level kernels, corresponding to activations, apply directly to every tile.

Linear layer. This layer in the GPT2 model transforms the embedding dimension of the input data \mathbf{X} from N_e to N_d via a linear transformation. Formally, this transformation is done through a matrix multiplication and an addition of a bias over unchanged dimensions: $\mathbf{Y} = \mathbf{W}\mathbf{X} + \mathbf{b}$, where \mathbf{Y} is of size $N_d \times N_s \times N_b$, \mathbf{W}

² <https://huggingface.co/openai-community/>

is of size $N_d \times N_e$ and $+$ denotes the summation of the same data over unchanged dimension. Tile-based implementation of this layer is also straightforward and based on the multiplications and additions of elementary tiles.

Layer normalization layer. Normalization layers are designed to improve stability of the training process and avoid gradient vanishing/exploding phenomena. Since our primary focus is the transformer-based models, we consider the Layer Normalization layer [6]. This type of normalization shows superior performance for transformer models over alternative normalization layers [17]. The tile-based implementation of this layer consists of three steps: mean and variance of every tile accumulation, normalization of every tile and scaling with an addition of bias.

SoftMax. SoftMax transformation converts the samples' embeddings to probabilities. Formally, given embedding $\mathbf{t} \in \mathbb{R}^C$, where C is a number of classes, for example, the SoftMax estimates probabilities \hat{p}_i as $\hat{p}_i = \frac{e^{t_i}}{\sum_{k=1}^C e^{t_k}} = \frac{e^{t_i - t_{\max}}}{\sum_{k=1}^C e^{t_k - t_{\max}}}$, where $t_{\max} = \max_i t_i$. The latter equality is an implementation detail to avoid infinities while calculating sum of exponents. Therefore, the computing of SoftMax for tensor decomposed in tiles consists of two subroutines. The first one computes t_{\max} and denominator per tile, and the second one aggregates them with entire values stored in tiles and computes the target probabilities estimate. In the GPT2 model, SoftMax is used inside the Attention layer (see below) and in the cross-entropy loss function.

Attention layer. This layer is crucial in constructing efficient embeddings for solving NLP tasks. The attention layer with a head size h contains three matrix multiplications to construct embeddings of queries \mathbf{Q} , keys \mathbf{K} and values \mathbf{V} . These embeddings are of size $h \times N_s \times N_b \times N_h$, where N_s denotes the number of sequences, N_b is the number of batches, and N_h is the number of heads. Then the SoftMax is applied: $\mathbf{B} = \mathbf{V} \left(\text{SoftMax} \left(\frac{1}{\sqrt{h}} \mathbf{K}^T \mathbf{Q} \right) \right)$ and the resulting embeddings \mathbf{Y} are computed via linear projection $\mathbf{Y} = \mathbf{W} \mathbf{B}^T$, where $\mathbf{W} \in \mathbb{R}^{N_e \times N_h \times h}$. The resulting embedding tensor \mathbf{Y} is of the size $N_e \times N_s \times N_b$. We implement these transformations based on tile splitting of input data and parameters.

Cross-Entropy loss is a multiclass generalization of the binary logistic loss, which measures the prediction quality in the training stage. It is computed via following equation $L = - \sum_{i=1}^{N_b} - \log \left(\frac{e^{x_{ic}}}{\sum_{j=1}^C e^{x_{ij}}} \right)$, where c is a correct class label and x_{ij} is proportional to the probability of the i -th sample to be assigned to the j -th class. Computing this loss is also feasible for input embeddings split on tiles.

Optimizers. The standard optimizers in training deep neural networks [14] are SGD with momentum, Adam, and AdamW. They minimize the loss function \mathcal{L} . In particular, an update step for SGD is just a weighted sum of the input vector, gradient, and momentum term. This update step perfectly fits a tile decomposition of the used tensors. Although the update steps for the Adam and AdamW optimizers are more tricky and require twice as much memory compared to SGD with momentum, they are also consistent with the tile decomposition.

NNTile distribution. The NNTile is an open source project. It is under development right now, with very limited functionality. The NNTile is available as the GitHub repository at <https://github.com/nntile/nntile>.

4 Numerical experiments

In the numerical experiments, we compare the performance and the scalability of the GPT2 models with 4 and 8 layers of various embedding sizes trained with NNTile and PyTorch. All the experiments were conducted with an NNTile version 1.0.0 (<https://github.com/nntile/nntile/tree/1.0.0>) on a single node with 8 A100 GPUs. Figure 1 shows that NNTile performs on par with PyTorch FSDP, while being able to train much larger models on the same hardware. This is due to a built-in automatic offloading of data from VRAM of GPUs to CPU RAM while CPU cores are idle. A similar observation holds for GPT2 with 8 layers; see Figure 2. However, the 8-layer model suffers from performance degradation due to far from optimal decisions of the StarPU greedy scheduling policy *dmdasd*. Unfortunately, the more layers there are in a neural network, the less the performance we get with the NNTile. At the same time, the presented plots demonstrate that the task-based parallelism approach, which is a core of the NNTile framework, is capable of training much larger GPT2 models rather than the straightforward PyTorch FSDP mechanism. Our experiments highlight the potential of task-based parallelism to bridge the gap between the theoretical and practical performance of GPUs’ utilization in training large language models.

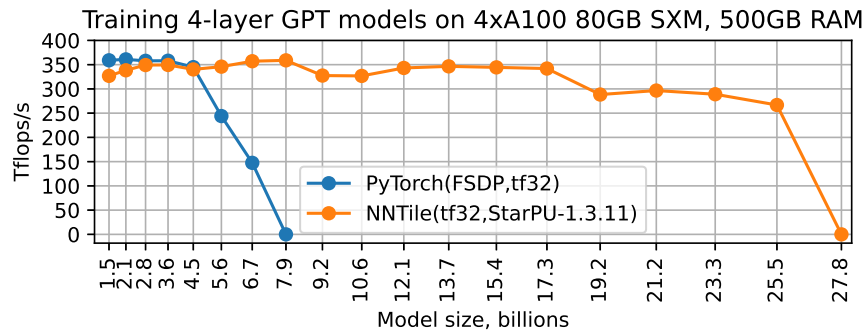


Fig. 1: GPT2 model with 4 layers trained on the four A100 GPUs in tf32 format of floating point numbers. NNTile can train significantly larger models than PyTorch FSDP (25.5B vs. 6.7B). Zero Tflops/s indicates Out-Of-Memory error.

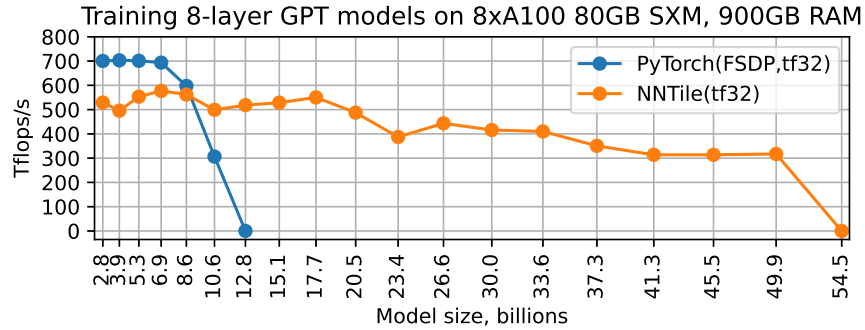


Fig. 2: GPT2 model with 8 layers trained on eight A100 GPUs in tf32 format of floating point numbers. NNTile can train significantly larger models than PyTorch FSDP (49.9B vs. 10.6B). Zero Tflops/s indicates Out-Of-Memory error.

5 Conclusion

This paper introduces the task-based parallelism paradigm for training large language models in a heterogeneous environment and the NNTile machine learning framework built on top of the StarPU library. The NNTile framework supports basic ingredients for neural network training. We implement forward and backward passes for the most popular layers (e.g., linear layer, attention layer, layer normalization layer, etc.), optimization methods based on the stochastic gradient estimate (e.g., Adam, SGD), and loss functions (e.g., cross-entropy and MSE losses). The implemented functions support data splitting into tiles across all possible axes. The StarPU backend processes tiles according to the underlying task scheduling policies. Numerical experiments demonstrate that NNTile can train the GPT2 model with 49.9 billion of parameters on a single node with 8 A100 GPUs. In comparison, PyTorch can train a similar model with only 10.6 billion of parameters on the same node within the same TensorFlow32 precision. Thus, we confirm the efficiency of task-based parallelism for training large language models with limited computational resources.

References

1. Artificial Intelligence (AI) Market by Offering (Hardware, Software), Technology (ML (Deep Learning (LLM, Transformers (GPT 1, 2, 3, 4)), NLP, Computer Vision), Business Function, Vertical, and Region - Global Forecast to 2030. URL <https://www.marketsandmarkets.com/Market-Reports/artificial-intelligence-market-74851580.html>
2. Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Thibault, S., Tomov, S.: A hybridization methodology for high-performance linear algebra software for GPUs. In: GPU Computing Gems Jade Edition, pp. 473–484. Elsevier (2012)

3. Almazrouei, E., Alobeidli, H., Alshamsi, A., Cappelli, A., Cojocaru, R., Debbah, M., Goffinet, É., Hesslow, D., Launay, J., Malartic, Q., et al.: The Falcon series of open language models. arXiv preprint arXiv:2311.16867 (2023)
4. Aminabadi, R.Y., Rajbhandari, S., Awan, A.A., Li, C., Li, D., Zheng, E., Ruwase, O., Smith, S., Zhang, M., Rasley, J., et al.: Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In: SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–15. IEEE (2022)
5. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009 **23**, 187–198 (2011). DOI 10.1002/cpe.1631
6. Ba, J.L., Kiros, J.R., Hinton, G.E.: Layer normalization. arXiv preprint arXiv:1607.06450 (2016)
7. Gusak, J., Cherniuk, D., Shilova, A., Katrutsa, A., Bershatsky, D., Zhao, X., Eyraud-Dubois, L., Shliazhko, O., Dimitrov, D., Oseledets, I., et al.: Survey on efficient training of large neural networks. In: Proceedings of the 31st International Joint Conference on Artificial Intelligence IJCAI-22, Vienna, Austria, pp. 23–29 (2022)
8. Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., de Las Casas, D., Hendricks, L., Welbl, J., Clark, A., et al.: An empirical analysis of compute-optimal large language model training. *Advances in Neural Information Processing Systems* **35**, 30016–30030 (2022)
9. Jiang, Z., Lin, H., Zhong, Y., Huang, Q., Chen, Y., Zhang, Z., Peng, Y., Li, X., Xie, C., Nong, S., et al.: MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In: 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), pp. 745–760 (2024)
10. Kaplan, J., McCandlish, S., Henighan, T., Brown, T., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., Amodei, D.: Scaling laws for neural language models. arXiv preprint arXiv:2001.08361 (2020)
11. Korthikanti, V., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., Catanzaro, B.: Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems* **5** (2023)
12. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al.: Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* **32** (2019)
13. Rasley, J., Rajbhandari, S., Ruwase, O., He, Y.: Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 3505–3506 (2020)
14. Ruder, S.: An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747 (2016)
15. Smith, S., Patwary, M., Norick, B., LeGresley, P., Rajbhandari, S., Casper, J., Liu, Z., Prabh-moye, S., Zerveas, G., Korthikanti, V., et al.: Using Deepspeed and Megatron to train Megatron-Turing NLG 530b, a large-scale generative language model. arXiv preprint arXiv:2201.11990 (2022)
16. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A., Kaiser, L., Polosukhin, I.: Attention is all you need. *Advances in neural information processing systems* **30** (2017)
17. Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C., Zhang, H., Lan, Y., Wang, L., Liu, T.: On layer normalization in the transformer architecture. In: International Conference on Machine Learning, pp. 10524–10533. PMLR (2020)
18. Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C.C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., et al.: Pytorch FSDP: experiences on scaling fully sharded data parallel. arXiv preprint arXiv:2304.11277 (2023)