École Normale Supérieure de Lyon

Projet Long de Recherche

Internship report

Efficient scheduling of Transformer models with StarPU

Enrique Galvez

Under supervision of
Olivier Beaumont
Lionel Eyraud Dubois
Julia Gusak
Laércio Lima Pilla
Philippe Swartvagher

Laboratory: Inria Bordeaux

TOPAL team

16^{th} of September 2024 – 31^{st} of January 2025

Contents

Introduction

AlexNet [1] was the first deep neural network to win the prestigious image recognition contest ImageNet in 2012. Since then, deep learning techniques became ubiquitous in almost every domain where traditional techniques were failing to provide a satisfying solution. Among these domains, natural language processing is a domain where machine learning can be used, but with poor results at the beginning due to the difficulty for standard deep neural networks to consider large context dependencies. The invention of Transformer models [2] in 2017 solved this issue by introducing the attention layer and an encoder-decoder structure. Many applications using Transformer-based models has shown impressive results in domains such as natural language understanding [3], text generation [4] or even dialogue systems [5]. However, these models are often very large and require a long training time, making their optimization critical in many applications.

Moreover, training large-scale models such as transformers often relies on accelerators like GPUs or TPUs to handle their significant amount of computations. These accelerators, along with CPUs and other specialized hardware, form heterogeneous systems that require an efficient coordination of computations and data movement. The modern way to address this concern is to delegate scheduling and data movement to a specialized framework such as StarPU [6], developped at Inria. Indeed, StarPU is a runtime system designed to manage parallel task execution across diverse hardware resources, ensuring seamless operation across CPUs, GPUs, and other accelerators. Its ability to dynamically adapt to available resources and workloads makes it particularly suitable for optimizing the execution of transformer models, where efficient utilization of heterogeneous hardware is crucial for performance and scalability.

The main purpose of this internship is to study how to use StarPU to efficiently execute Transformer-based models. For this purpose, we suggest optimizations for the deep learning framework NNTile [7], that provides implementations for well-used models such as GPT-2 [8] and Llama [9], based on StarPU for task scheduling. However, NNTile has several points that can be improved in its management of tasks. Firstly, we describe in this report several methods to improve task granularity in order to benefit from better performance on behalf of StarPU's scheduling. Secondly, we determine which tiling parameters are the most suitable to allow the training of big models with a good performance. Lastly, we compare our solutions implemented in NNTile with default implementations in a multi-GPU context.

1 Training of deep learning models

1.1 The Transformer architecture

Transformer models were first introduced by Vaswani et al. in 2017 [2]. In their article, they invent the self-attention mechanism and design the Transformer architecture around it. This mechanism greatly improves the accuracy of transformers compared to Recurrent Neural Networks, in particular for capturing long-term dependencies. As a result, transformers brought significant improvements in the domains of Natural Language Processing, Speech Recognition and more. The original design of a transformer consists of an encoder-decoder structure. In this architecture, the encoder processes the input sequence to generate contextualized representations, and the decoder generates the output sequence based on these representations. Both encoder and decoder are composed of Transformer blocks that are themselves composed of a multi-head self-attention layer and feed-forward layer.

In this work, we consider decoder-only networks, such as GPT [8, 10] and Llama [9]. Their specificity is to only use the decoder part of the Transformer architecture, described in Figure 1. The first part of such a network aims at transforming the input into numerical data on which the next parts will operate. The first element of this block is the tokenizer, which transforms the text sequence into small pieces of information called tokens. Then, an embedding is computed to transform the text into numerical vectors. The embedding usually has good properties: if two words have a close meaning, the distance between their corresponding vector representation should be small. Modern networks also include at this step a *positionnal encoding* to keep track of the position of the token inside the sequence, considering that the meaning of a piece of text depends on its position in the text. The next part of the transformer structure is the Multi-Head Self-Atention layer. It splits the input into multiple "attention heads", each performing scaled dot-product attention independently, enabling the model to capture diverse relationships and patterns in the data. Each attention head interprets its input vector as a query, key, value (Q, K, V) tuple in order to extract context-specific information by weighting V by an attention score computed using Q and K. This attention block is followed by a feed-forward network. The combination of the attention block and the feed-forward network is

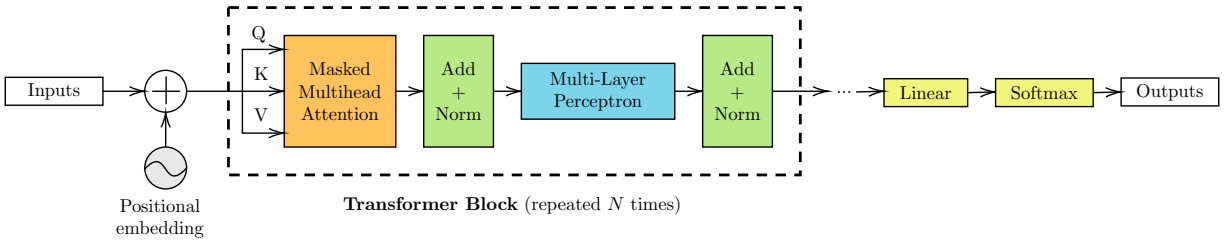


Figure 1: Architecture of a decoder-only Transformer [10]

a Transformer Block, which is repeated N times to constitute the whole network. At the end of the network, linear and softmax layers are used to extract the output, which is usually a probability vector.

1.2 Efficient execution of the attention layer

The attention layer performs complex and computationally-intensive operations, making its optimization critical when aiming at an efficient execution of transformers. The structure of the multi-head attention block is composed of several parts, described by below formulas. Firstly, the input is splitted accross the several attention heads, and each head divide again its input into a query, key, value (Q, K, V) tuple. Then, each head applies to the (Q, K, V) tuple an operation called scaled dot-product attention. This operation consists in matrix multiplications and a masked softmax. The goal of scaled dot-product attention is to weight V tensor by an attention score computed using Q and K.

By writing W_O, W_Q, W_K, W_V the learnable weights, the formula for a forward pass through multi-head attention layer can be written as:

$$MultiHead(X_Q, X_K, X_V) = Concat(head_1, ..., head_h)W_O \tag{1}$$

Where

$$head_i = Atention(X_Q W_Q^{(i)}, X_K W_K^{(i)}, X_V W_V^{(i)}) \tag{2}$$

And

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V \tag{3}$$

With softmax applied element-wise along each row of the matrix $\left(\frac{1}{\sqrt{d_k}}\right)QK^T$, to produce attention weights across the sequence. For a given element z_i in a row, softmax formula is:

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \tag{4}$$

This ensures that the attention weights for each query vector sum to 1, providing a probability distribution that reflects the relative importance of each key-value pair with respect to the query [2].

Algorithm 1: Multi-head attention forward pass

Inputs: X_Q, X_K, X_V
Weights: W_Q, W_K, W_V, W_O
$Q^T \leftarrow$ gemm(X_Q, W_Q) ; $K^T \leftarrow$ gemm(X_K, W_K); $V^T \leftarrow$ gemm(X_V, W_V)
$Q \leftarrow$ transpose(Q^T) + $bias_Q$; $K \leftarrow$ transpose(K^T) + $bias_K$; $V \leftarrow$ transpose(V^T) + $bias_V$
$A \leftarrow (1/\sqrt{hd\_size})\times$ batched_gemm(K^T,Q)
$A \leftarrow$ mask(A,$mask$)
$A\_mse \leftarrow$ maxsumexp(A)
$A \leftarrow$ softmax_inplace(A, A_mse)
$B \leftarrow$ batched_gemm(A,V)
$B^T \leftarrow$ transpose(B)
$Y \leftarrow$ gemm(W_O, B^T) + $bias_O$
Return Y

According to these formulas, the pseudo code for the forward pass through multi-head attention layer can be described in Algorithm 1. Note that multiple heads are managed by adding a dimension to the tensors. In this pseudo-code, line 3 and 4 shows the preparation of the inputs before computing the attention score. Line 5 is the most expensive in terms of computations because it computes the matrix multiplication between K^T and Q, creating the $N \times N$ attention matrix, N being the size of the sequence. The *maxsumexp* operation at line 7 iterates through the attention matrix in order to pre-compute statistics required for the inplace computation of softmax at line 8. Result of softmax is then multiplied by V at line 9 and output weights are added to the output matrix Y at line 11.

Algorithm 2: Multi-head attention forward pass using FlashAttention

1 **Inputs:** X_Q, X_K, X_V
2 **Weights:** W_Q, W_K, W_V, W_O
3 $Q^T \leftarrow \text{gemm}(W_Q, X_Q)$; $K^T \leftarrow \text{gemm}(W_K, X_K)$; $V^T \leftarrow \text{gemm}(W_V, X_V)$
4 $Q \leftarrow \text{transpose}(Q^T) + bias_Q$; $K \leftarrow \text{transpose}(K^T) + bias_K$; $V \leftarrow \text{transpose}(V^T) + bias_V$
5 $A\_mse \leftarrow \text{flash\_maxsumexp}(Q, K, mask)$
6 $B \leftarrow \text{flash\_softmax\_gemm}(A\_mse, Q, K, V, mask)$
7 $B^T \leftarrow \text{transpose}(B)$
8 $Y \leftarrow \text{gemm}(W_O, B^T) + bias_O$
9 **Return** Y

Even if the first algorithm algorithm is straight-forward from the mathematical definition of multi-head attention block, its execution suffers from poor performance. Indeed, several works [11, 12] criticized the quadratic memory requirement introduced by the materialization of the attention matrix A. The solution implemented in most of State-of-the-art frameworks like Pytorch is FlashAttention [12]. This method consists in computing rows of the B matrix directly using corresponding rows of Q and columns of K^T, without materializing the entire attention matrix A. This optimization decreases the memory requirement of the attention block as well as it optimizes the I/O complexity [13]. FlashAttention algorithm, presented in Algorithm 2 has a structure similar to Algorithm 1. However, the difference resides within the implementation of the softmax operation at line 6, where the tensor B is built without materializing the matrix A.

1.3 Parallel training of DNNs and Transformers

Over past few years, transformer models have known a significant increase of their number of parameters. For instance, GPT-3 is a model composed of 175×10^9 parameters [14] while the number of parameters in GPT-4 [10] is estimated around 1.8×10^{12}. As a consequence, training recent models require an important computational power and several high performance computational units are often required. However, the sequential design of these models with data dependencies between successive layers make their parallelization a difficult challenge. Some approaches in State-of-the-art propose to modify the design of the model to fit machine parallelism [15, 16], while other propose generic methods to execute any Transformer in a parallel context [11, 17].

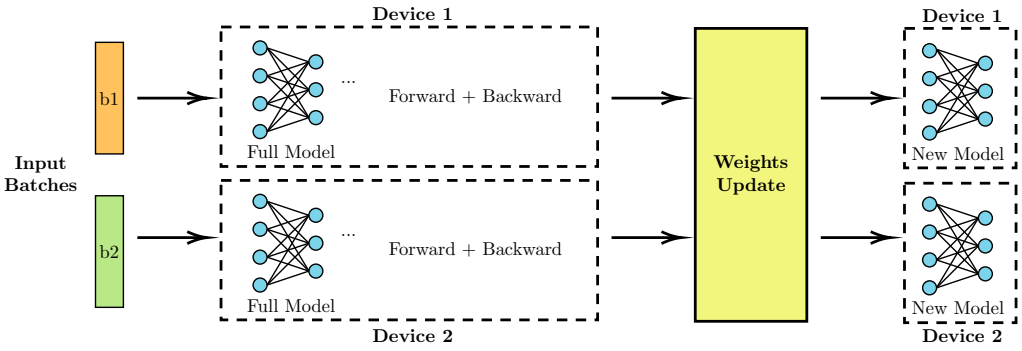


Figure 2: Data parallelism

A first method to train deep learning models in a parallel context is data parallelism [18]. Described in Figure 2, data parallelism consists in training the model in parallel across several batches of data. In such a context, model is replicated across the computational units and synchronizations are performed between weights during their update. The main advantage of this approach is its simplicity, because model is stored entirely on each device, this method can be easily adapted to different models. However, its main drawback is the fact that devices should be able to store the entire model in its memory, limiting the generalization of this method for very large models.

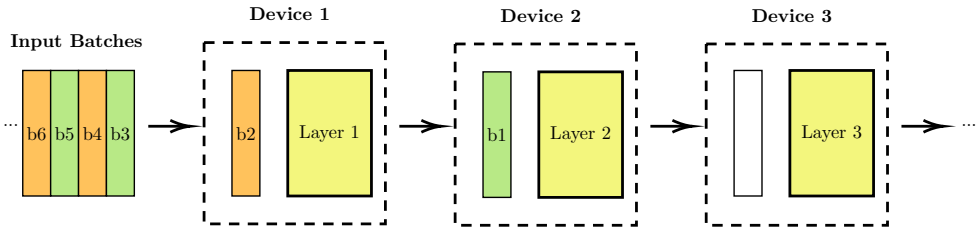


Figure 3: Pipeline parallelism

A second approach is pipeline parallelism [19]. In this approach, each computing unit is specialized in a specific layer of the model, and executes the forward pass and the backward pass for a batch of inputs. As described in Figure 3 , in order to perform a forward pass through the network, some dependencies should be met. For example, before performing forward pass for input batch b through layer n (on device n), device $n-1$ should have computed forward pass for input batch b on layer $n-1$. Dependencies are reversed for backward pass: before performing backward pass for input batch b through layer n (on device n), device $n+1$ should have computed backward pass for input batch b on layer $n+1$. These dependencies are the main drawback of this approach, as they introduce barriers, often preventing an optimal scheduling of the computations.

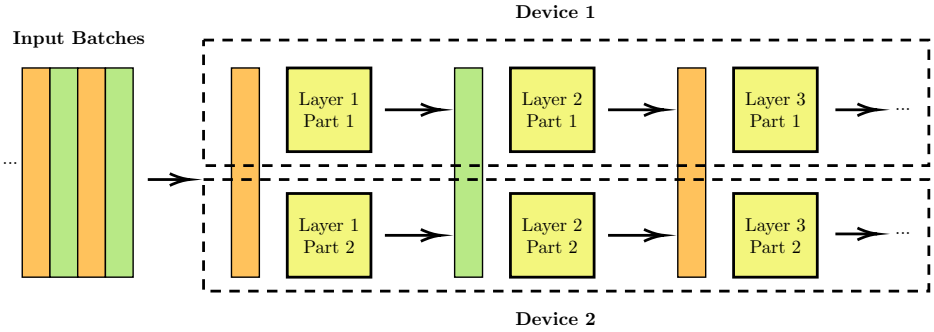


Figure 4: Tensor parallelism

Another approach is Tensor-parallelism [20]. This approach, described in Figure 4 consists in developing parallel kernels for the different deep learning operations. This approach requires specific development for the parallel kernels but it offers significative performances. In this method, parallelism is relative to the tensors used in the model, with several implementations possible depending on the dimensions of the tensors.

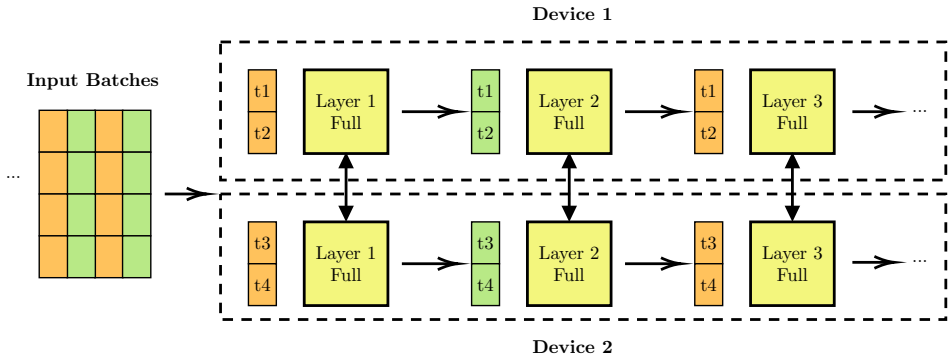


Figure 5: Sequence parallelism for Transformers

A last approach, specific to Transformer models is Sequence parallelism [11]. In a similar way to tensor parallelism, it lays on specific parallel kernels as described in Figure 5. However, in this approach, the tokens of each sequence batch are split across the computing devices. This requires specific optimizations and algorithmic modifications to the original Transformer implementation, in particular for the attention layer.

Each of the previous approaches can be mixed with others in order to add levels of parallelism. Such an approach is generally called block-wise parallelism, and relies on an efficient scheduling of tasks working on small pieces of the whole operation, called tiles. This is the approach leveraged by StarPU and NNTile, both described in the following section.

2 Programming framework

2.1 StarPU

Very large Transformer-based models are ubiquitous in most of current deep learning applications such as natural language processing, text generation or even image processing. However, the training of models with that much parameters require a large number of computing units including efficient accelerators. The conception of deep learning implementations for large models on such hardware require a specific abstraction. Indeed, pieces of code executed on different hardwares should be optimized for each particular platform, and communications should be managed efficiently between the computing units. StarPU [6] is a framework that provides this abstraction, by introducing a task-based programming paradigm, separating the implementation of the architecture-specific kernels from the scheduling of the calls to these kernels and the management of the communications. Figure 6 describes where StarPU position itself, between the application and the hardware.

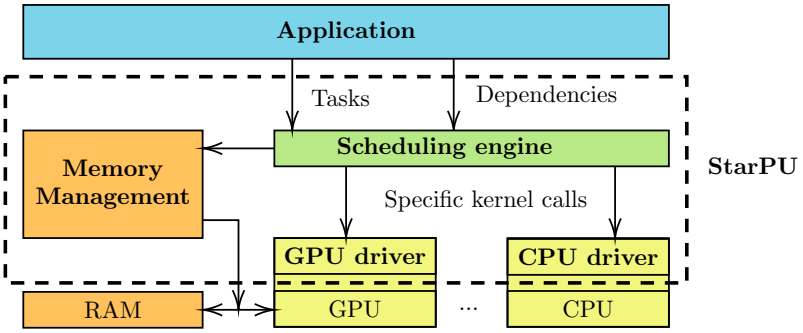


Figure 6: StarPU structure [6]

In order to provide such an abstraction, StarPU uses a programming paradigm based on the creation of tasks that can be scheduled on any computing unit if the task has an implementation for it, called kernel. For example, to create a matrix multiplication kernel with StarPU, one can write a version for this multiplication using CUDA and another one that just calls an optimized CPU library. To execute the matrix multiplication, the user of StarPU will call the method `starpu_task_insert` to request the creation of the matrix multiplication task. Then, it is StarPU's job to determine on which device to execute the task, using the appropriate kernel for the device selected. In addition to that, StarPU also manages scheduling of the tasks in the case where many tasks are created. In practice, a lot of tasks should be considered for a StarPU application, to allow scheduling to fill all the computing resources with tasks at any moment.

An important thing to note is that StarPU tasks are executed in an asynchronous way, but respecting data dependencies between the tasks. Indeed, depending on the lecture modes (read/write) of the data used by each kernel, StarPU generates a dependency graph and uses scheduling algorithms to determine an efficient schedule. This approach allows portability in terms of performance for StarPU-based applications between different distributed architectures, eventually heterogeneous. Moreover, because scheduling is isolated from application's development, StarPU allows easy scalability while increasing the number of computing units.

2.2 NNtile

NNTile [7] is a deep learning framework based on StarPU and developped by specialists from Skolkovo Institute of Science and Technology (Skoltech) and Artifical Intelligence Research Institute (AIRI) from Moscow. The motivation for NNTile is to leverage StarPU's scalability in a deep learning framework for training large models. NNTile supports recent models such as GPT-2, LLaMa and Bert, and allows the user to load weights from a pretrained PyTorch model. As shown in Figure 7, NNTile allow to train models up to 4 times larger than what is possible with PyTorch FSDP module (Fully Shared Data Parallel) while conserving a decent performance in terms of Tflop/s.

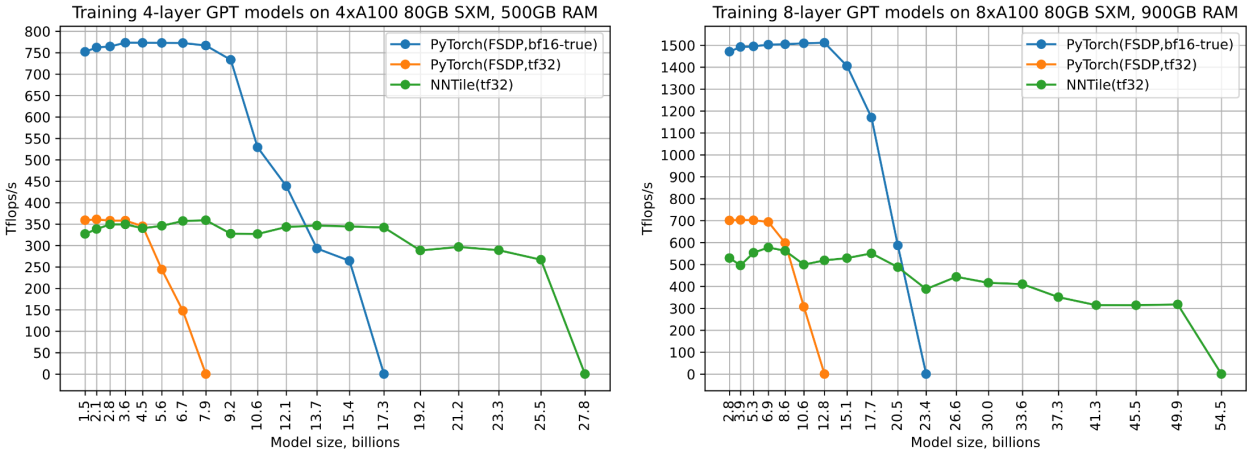


Figure 7: NNTile performance for training big models (from [7])

To achieve that, NNTile implements asynchronous operators based on kernels executed with StarPU. In concrete terms, when the user runs an inference for a given model, all the tasks are first submitted to StarPU which schedules them according to their data dependencies on the computing units available, and then the actual computations can begin. The main advantage of NNTile is to implement Blockwise paralellism [17] accross model's execution. This means that each dimension of the tensors are divided in multi-dimensional tiles that can be passed as input to StarPU tasks. Supporting tiling in this context requires specific implementations for most of the deep learning operators of the transformer architecture. An overview on how these implementations work is provided in the following section.

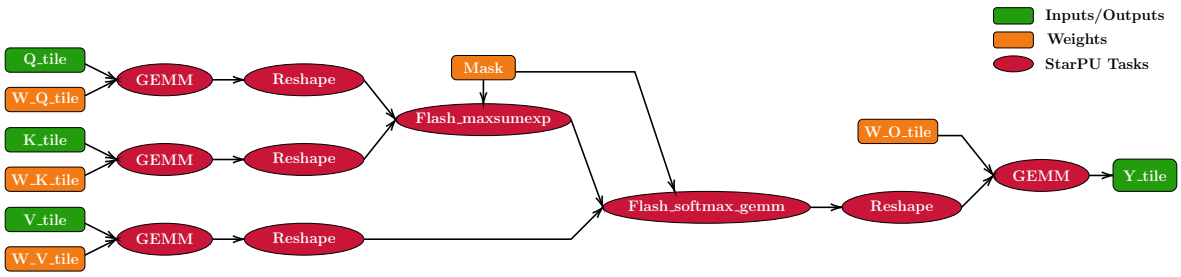


Figure 8: FlashAttention task graph with only 1 tile, no bias

An example of task graph for a deep learning operator used in transformers is shown in Figure 8. The nodes in the task graph are the asynchronous tasks submitted to StarPU and the edges represent the data dependencies between two tasks. As an example, if a task B require reading a data that is written by a task A, then an edge will be added from A to B. The dependency graph shown in Figure 8 corresponds to the FlashAttention layer, with a single attention head and without tiling. Input and output tiles are represented with green rectangles while the weights are in orange rectangles and tasks are represented in red ovals. The arrows represent the dependencies between the tasks and the data, meaning that a child task should ever be executed after its parent in the task graph. It is the responsibility of StarPU to schedule efficiently these tasks on the computing units available with respect to their dependencies.

2.3 Limitations of NNTile's implementations

Despite the good performance results that NNTile can provide, some articles in State-of-the-Art define guidelines that should be respected when developping StarPU applications to benefit from correct performance. Indeed, in the case of a bad use of StarPU, the performances can be very bad, especially if data tiling is badly handled. A first well-known issue occurs when data dependencies are too strong between successive tasks, preventing an optimal parallel execution. The consequence is that computing units will often have to wait for the execution of tasks in other units to finish before being able to execute the next tasks. Another issue, documented in Figure 7 of [21], resides in the link between task granularity and efficiency. Indeed, I StarPU tasks are too small, the IDLE time caused by scheduling overhead will become significant and slow down the execution. To solve this issue, it has been stated in [21] that efficiency is nearly optimal when tasks last more than 3ms.

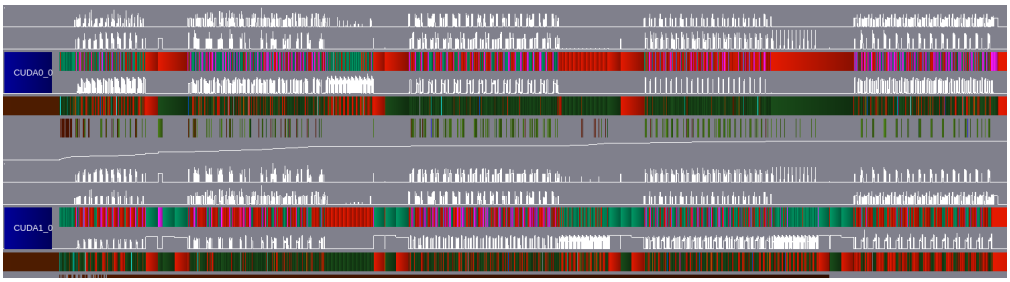


Figure 9: NNTile profile for GPT-2 with bad tiling parameters

Figure 9 shows the scheduling graph while executing GPT-2 inference with NNTile's default tiling parameters on 2 NVIDIA Tesla V100 GPUs. In Figure 9, green parts shows the moments when computing units are working and red parts shows IDLE state of computing units. We observe in these graphs that without a special attention on how tiling is handled, scheduling may be highly sub-optimal, with computing units that are more in IDLE state than in working state during the whole execution. These first observations motivated the optimization efforts on NNTile's kernels described in this report.

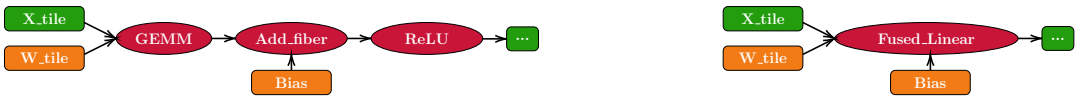
A second limitation of NNTile's default implementations resides in the way tasks are divided. Indeed, NNTile divides all the elementary deep learning operations as StarPU kernels, without considering their different computational cost. As a consequence, small operations such as bias addition can be considered as a task as well as bigger operations such as matrix multiplications. This results in the presence of small tasks in the schedule, causing the issue previously mentionned. To address this issue, we propose to fuse kernels, as described in the following section.

3 Improving division of tasks

3.1 Fusing GEMM with activation and bias

As explained in previous section, NNTile's implementations may suffer from sub-optimal performances due to the presence of small tasks in the schedule, associated with much bigger ones. These small tasks may have a duration smaller than 3ms, which can be considered as the threshold to benefit from optimal StarPU performance [21]. In order to fix this issue, we propose to fuse the smallest operators with the bigger ones. An example for two operators that can be fused are the **activation** layer which is an element-wise operation (ReLU, GeLU...) and the **linear** layer which compute a matrix multiplication. An important thing to note is that we will consider two levels of fusion: the task-wise fusion consists in simply regrouping several tasks to a single one and the operator-wise fusion consists in fusing the kernels to compute directly the result of both operators.

In the case of Linear+ReLU fusion, Both operator-wise and task-wise fusion can be applied. In this case, operator-wise fusion would consist in applying the ReLU transformation inside the GEMM kernel. In more concrete terms, inside the GPU kernel, we can replace the cuBLAS call used to compute matrix multiplication with a cuBLASLt call [22] computing the result of Linear+ReLU operation using a fused kernel. In addition to that, task-wise fusion is natural from this point, since kernels are fused the fused kernel is called in a single task instead of two. Figure 10 shows the evolution of the task graph when a Linear layer is fused with the application of biases and an element-wise activation operation such as ReLU in this case.



(a) Without fused kernels (b) Using fused kernels

Figure 10: Task graph for the operation `ReLU(Add_fiber(GEMM(X,W),bias))`

Besides reducing the task graph, as shown in Figure 10, Linear+Bias+ReLU fusion also reduces scheduling overhead as it groups several tasks into a single one. The evolution of the scheduling profile can be observed in Figures 11 and 12, where green parts represent WORKING state and red parts represent IDLE state for the computing units considered (here 2 NVIDIA Tesla V100 GPUs).

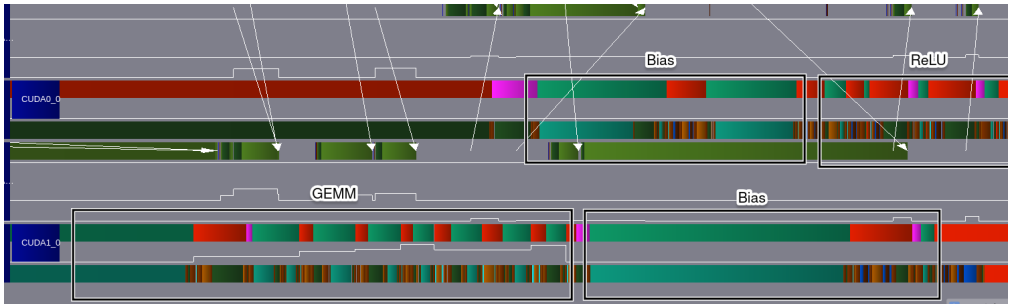


Figure 11: Tasks execution for `ReLU(Add_fiber(GEMM(X,W),bias))` without fused kernels

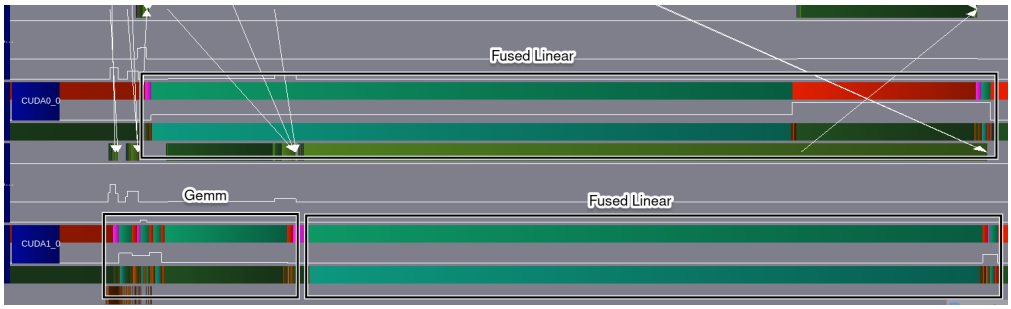
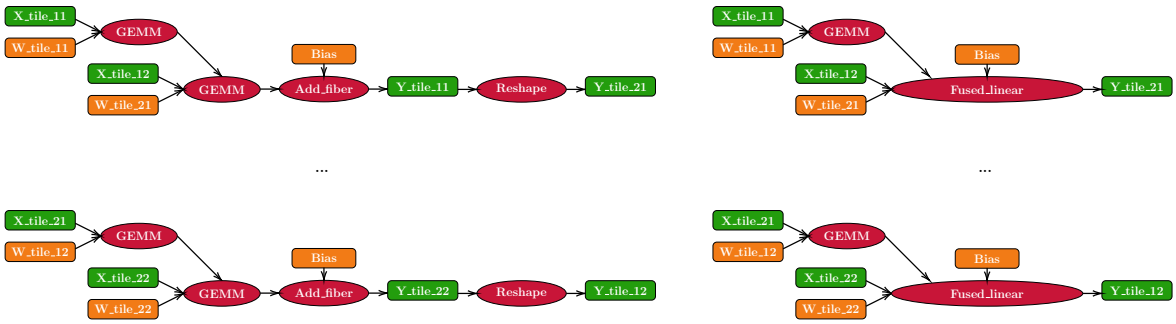


Figure 12: Tasks execution for `ReLU(Add_fiber(GEMM(X,W),bias))` using fused kernels

In addition to the benefits already described, fusing kernels allow the system to reuse data loaded in the memory of the same computing unit instead of requiring data transfers between several units. Moreover, operator-wise fusion is the most efficient way to compute the succession of GEMM and ReLU operations on a GPU because it optimizes memory accesses by computing both operations at once.

3.2 Fusing GEMM with reshape

In previous subsection, we focused on the fusion of kernels of a transformer's multi-layer perceptron. However, these fusion techniques can also be applied in the transformer's attention block. Indeed, attention block is composed of many different operators with unequal durations. FlashAttention's task graph displayed in Figure 8 shows that GEMM operations that are costly are often followed by reshape operations, usually smaller. However, splitting these tasks is not optimal since the reshape operator should require an additional iteration through the matrix after the GEMM. Moreover, reshape kernels will result in short tasks since nothing is computed and data is just moved. For all these reasons, fusing GEMM and reshape operators appear to be a good optimization. To illustrate the consequences of this fusion, Figure 13 shows the task graph of the attention layer before and after applying the fusion.



(a) Without fused kernels (b) Using fused kernels

Figure 13: Task graph for the operation `Reshape(GEMM(X,W))`

In order to understand how we can fuse GEMM and reshape kernels in a blockwise-parallel context, we need to first understand how a blockwise-parallel GEMM works. Algorithm 3 computes the multiplication between matrices A and B that are both divided into blocks of fixed size. Loops at lines 5 and 6 performs an iterations through blocks of the output matrix C while loop at line 7 iterates through the inner dimension of the matrix product. This loop calls an asynchronous kernel `gemm_async` which correspond to a StarPU task submission. An important thing to note is that `gemm_async` updates the value in the C block. As a consequence, the execution of line 8 for each k depends on the previous execution of line 8 for $1, \ldots, k-1$. In distributed terms, we say that loop 7 performs a reduction through k, which can be optimized by setting `STARPU_REDUX` access mode for C blocks, letting StarPU's scheduler anticipate the dependency pattern created by the reduction.

Algorithm 3: Distributed GEMM

1 **Input:** Matrices A (size $M \times K$), B (size $K \times N$)
2 Matrix A is divided into $P \times Q$ blocks $A_{i,j}$
3 Matrix B is divided into $Q \times R$ blocks $B_{i,j}$.
4 Initialize: $C_{i,j} \leftarrow 0$ for all $i < P$ and $j < R$
5 **for** $i = 0$ *to* P **do**
6 **for** $j = 0$ *to* R **do**
7 **for** $k = 0$ *to* Q **do**
8 | gemm_async($C_{i,j} \leftarrow C_{i,j} + A_{i,k} \times B_{k,j}$)
9 **Return** C

Algorithm 4: Distributed GEMM+reshape

1 **Input:** Matrices A (size $M \times K$), B (size $K \times N$)
2 Matrix A is divided into $P \times Q$ blocks $A_{i,j}$
3 Matrix B is divided into $Q \times R$ blocks $B_{i,j}$.
4 Initialize: $C_{i,j} \leftarrow 0$ for all $i < P$ and $j < R$
5 **for** $i = 0$ *to* P **do**
6 **for** $j = 0$ *to* R **do**
7 | $\tilde{i}, \tilde{j} \leftarrow$ reshape_indices(i,j)
8 | **for** $k = 0$ *to* Q **do**
9 | | gemm_reshape_async($C_{\tilde{i},\tilde{j}} \leftarrow C_{\tilde{i},\tilde{j}} + \text{reshape}(A_{i,k} \times B_{k,j})$)
10 **Return** C

This distribued GEMM algorithm can be fused with a reshape operator, as described in Algorithm 4. The idea of this algorithm is to apply the reshape after the GEMM for each block in the StarPU task and reorder the blocks to correspond to the reshaped output. This algorithm has the same structure as Algorithm 3. However, in this version, post-reshape indices $\tilde{i}$ and $\tilde{j}$ are computed at line 7 and used to select the output block where matrix products are summed at line 9. Moreover, the StarPU kernel called at line 9 applies the reshape after computing the GEMM operation. These modifications ensures that Algorithm 4 computes the output of a GEMM followed by a reshape.

3.3 Implementing fused kernels in GPT-2

In previous subsections, we improved separately some operators used in a transformer model such as GPT-2. However, to benefit from good performance of these optimized operators in the context of GPT-2 training or inference, it is necessary to consider the whole execution. In GPT-2 inference task graph, each layer strongly depends from the previous one so we do not expect scheduling to be optimal between the layers. However, thanks to the tiling, we expect to observe a good parallel scheduling of tasks inside each layer. According to previous subsections, we can fuse GEMM with small following operations when GEMM is followed by an activation, reshape or bias operator. To perform this fusion, we developed a new NNTile kernel called `FusedLinear` which computes a GEMM fused with a selection among these operations.

Moreover, in order to benefit from optimal performance on behalf of the kernels, it is necessary to choose appropriate tiling parameters. Indeed, tiles should be big enough to allow StarPU tasks to last more than

3ms [21] but tensors should be divided in enough tiles to allow the scheduler to fill each computing unit with enough tasks to maximize occupancy. In other words, there is a trade-off between parallelism opportunities and StarPU-overhead. This tradeoff depends on the number of computing units available and on the size of the model considered. This tradeoff is explored in practice in the next section.

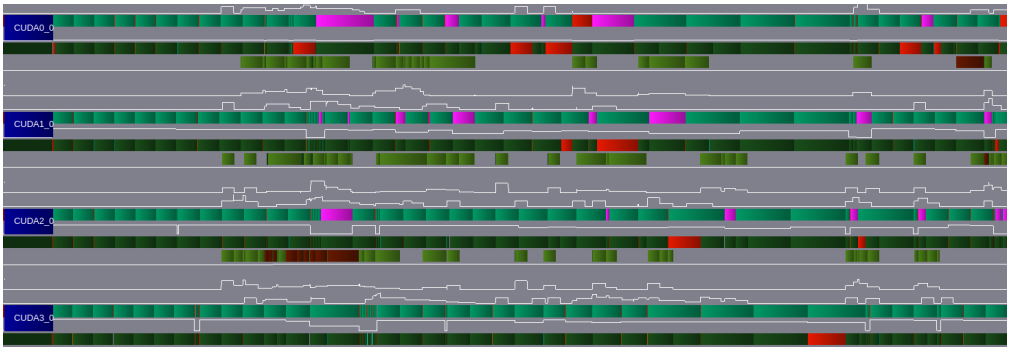


Figure 14: NNTile profile for GPT-2 with good tiling parameters

The benefits of using good tiling parameters can be seen in Figure 14. We observe that IDLE time has been significantly decreased compared to what we observed in Figure 9 with bad tiling parameters. In this profile, purple represents the communication state, meaning that the GPU is waiting for a memory transfer to finish. Minimizing the cost of these memory transfers is a difficult challenge, but in our case we consider that StarPU scheduling handle it with an efficient enough solution.

4 Performance results

4.1 Performance evaluation methodology

To run our experiments, we use the high performance computing cluster Grid5000 [23]. This cluster provides nodes with powerful GPUs such as NVIDIA Tesla P100, V100 or A100. Since NNTile's GPU kernels require a CUDA version higher than 7.0 and because of the presence of nodes accelerated with 8× V100 GPUs, we choose these nodes to run our experiments. This environment is representative of a GPU cluster being between 5 and 7 years old. In order to explore the multi-GPU aspect of transformer execution, we run experiments on these machines with up to the maximum number of 8 NVIDIA Tesla V100 GPUs on a single node. The fact that GPUs are directly part of a single node has the advantage to provide a multi-GPU architecture with fast communications.

All the experiments were tested by running our modified version of NNTile inside docker containers in Grid5000 cluster. We used NNTile's feature that allows us to import PyTorch model to use PyTorch's GPT-2 weights and model structure. Corresponding code is available in the gitlab repository `https://gitlab.inria.fr/egalvez/nntile-attention.git` and instructions to run it are available at `https://stage-plr-docs-f9fc50.gitlab.io/content/eff-att/info/`.

4.2 Performance of layer fusion on toy models

In this section, we evaluate separately the fused kernels described in previous sections. The main purpose of this study is to determine whether fused kernels are providing performance improvements, and if these improvements depends on a specific experimental context. Indeed, depending on model dimensions, StarPU tasks may be too short to provide decent performance, due to the high relative cost of StarPU scheduling for very small tasks [21]. For this reason, the choice of dimensional parameters is a very important part of the experimental process.

The choice of the tensor sizes is particularly important for evaluating a simple model performing a matrix multiplication followed by a ReLU operation and a sum with biases. Indeed, ReLU and bias tasks are much faster than GEMM task, resulting in a high relative cost of StarPU overhead if tensor sizes are chosen too small. Both graphs in Figure 15 show the execution time for a simple model composed of the successive layers Linear, ReLU and bias, depending on the number of GPUs used. In the first graph, tensor sizes are taken from default GPT-2 sizes, and we observe very small execution times, meaning that tasks are too small.

As a result, the use of fused kernels provides a significant speedup in this case because tasks will all last at least as long as the execution of a GEMM task, thus reducing relative cost of StarPU overhead. Another interesting observation is that when the number of GPUs increase, the execution may be slower because of the lack of parallelism opportunities to fill all the GPUs with work. This issue will be solved automatically when infering through a whole GPT network, thanks to the higher number of tasks submitted to StarPU, without very strong dependencies.

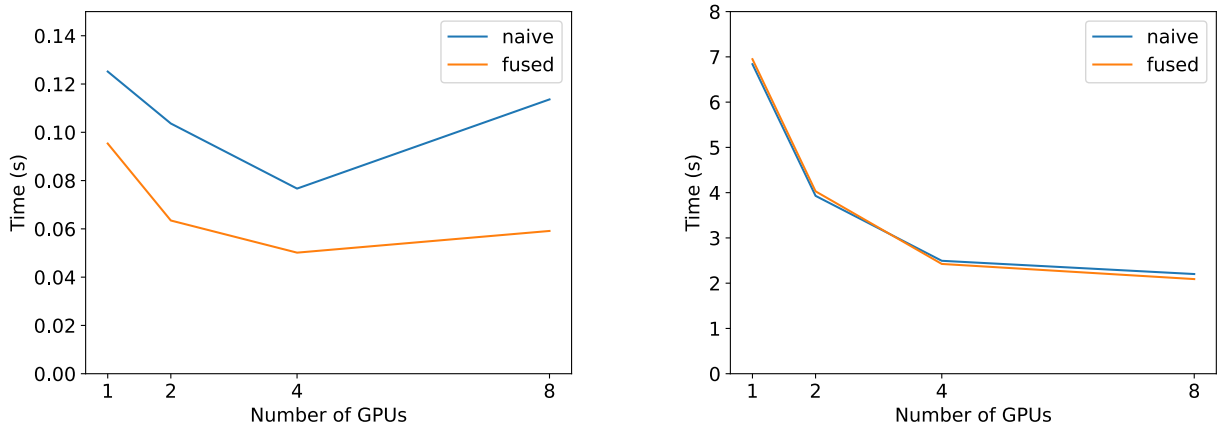


Figure 15: Multi-GPU performance for Linear+ReLU+bias

The second graph of Figure 15 also shows the execution times depending on the number of GPUs for a simple model composed of the successive layers Linear, ReLU and bias. The difference with the first graph is that we fixed tensor sizes bigger than the sizes used in the first graph, in order to ensure that the smaller task ReLU last more than the threshold of 3ms. To increase the size of the model, we increased the size of sequence and embedding, allowing us to increase the size of the tiles for the corresponding tensors, and thus the duration of the tasks. The multiplication we consider, used in GPT-2's multi-layer perceptron, can be described as in Equation 5, considering that emb, seq and hid represent embedding size, sequence size and hidden size, the latter being usually equals to $4 \times emb$. In this second graph, we observe that tasks are slower and resources are well used, but the implementation with fused kernels is no longer providing much speedup compared to the naive version.

$$\underbrace{batch \times emb \times seq}_{Inputs} \overset{GEMM}{\otimes} \underbrace{seq \times hid}_{Weights} = batch \times emb \times hid \tag{5}$$

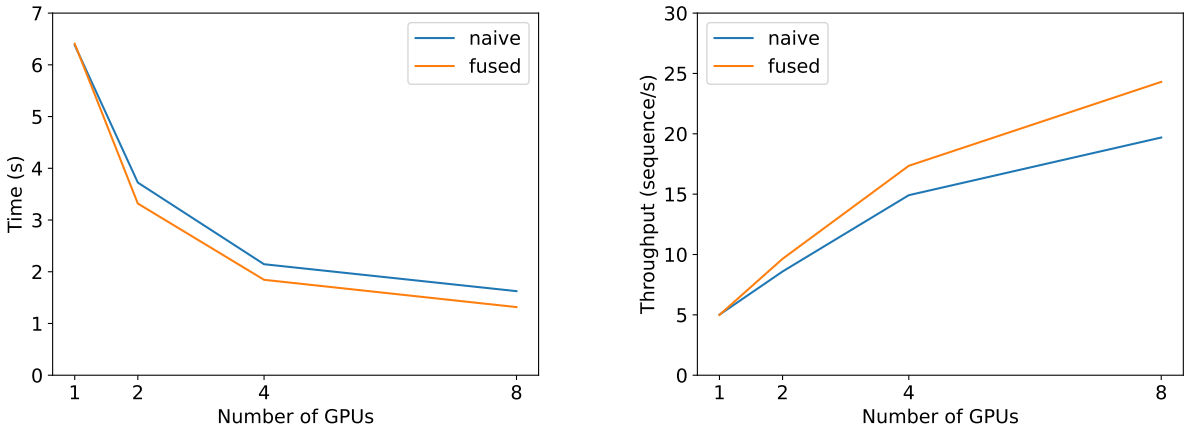


Figure 16: Multi-GPU performance for Linear+Reshape

In a similar way, Figure 16 shows the time for the multi-GPU execution of the succession of Linear and Reshape layers, consisting in computing the operation: `Reshape(GEMM(X,W))`. However, this succession is part of GPT-2's Attention block, in which tensor sizes are bigger than in the multi-layer perceptron. As a consequence, tasks already last long enough to provide decent performance with GPT-2 sizes. In such a context, we observe a performance improvement while using fused kernels instead of naive ones. To conclude, we expect that fused kernels for both Linear+ReLU and Linear+Reshape will provide performance improvement to GPT-2 inference due to the high number of tiles, allowing a good scheduling of the operations.

4.3 Performance of layer fusion on GPT-2 inference

In the following experiments, we implemented kernel fusion in several levels in GPT-2 inference. Figure 17 shows the performance obtained by using fused kernels in GPT-2 inference. In the first graph of Figure 17, we observe that the use of fused kernels provide a speedup compared to naive implementation, and that this speedup decreases when the number of GPUs increases. An explanation to this is the fact that the cost of small non-fused tasks such as ReLU, Reshape or bias become very small for many GPUs because they can be scheduled in a way that do not block computing units for more expensive operations such as GEMM or softmax.

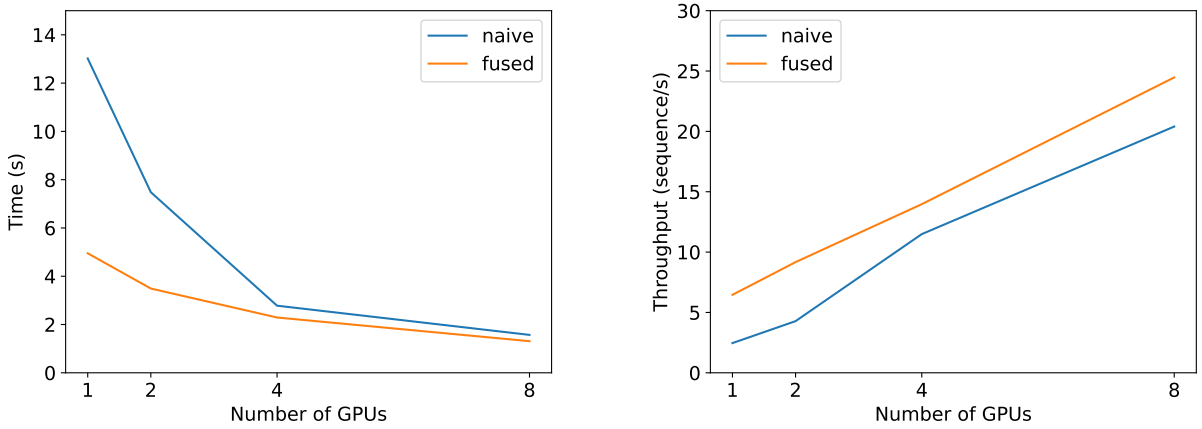


Figure 17: Multi-GPU performance for GPT-2 inference

Another observation in Figure 17 is that throughput scales almost linearly with respect to the number of GPUs. Moreover, the second graph in Figure 17 shows that the fused implementation allows a better scaling of the throughput accross the number of GPUs compared to the naive version. This shows that the tiling parameters chosen do not restrict parallelism opportunities for the fused version.

4.4 Performance tradeoff between parallelism and occupancy

As explained previously, tensor sizes and more particularly tile sizes have a critical impact on the performance of GPT-2 execution. Indeed, tasks too small result in a high cost of StarPU overhead while tiles too big result in a lack of parallelism opportunities. Figure 18 shows the impact of several tile sizes for the embedding dimension of the tensors of GPT-2 during its inference. The first graph shows this evolution for naive kernels and the second one shows it for fused kernels. A first observation that we can make is that increasing tile size seem to increase performances. However, for the naive implementation a value of 768 or 1536 is optimal while 1536 is optimal for all the setups when we use fused kernels. A second observation is that in both cases, using a tile size equal to 3072 (*i.e.* no tiling for the embedding dimension) result in a slight degradation of the performances.

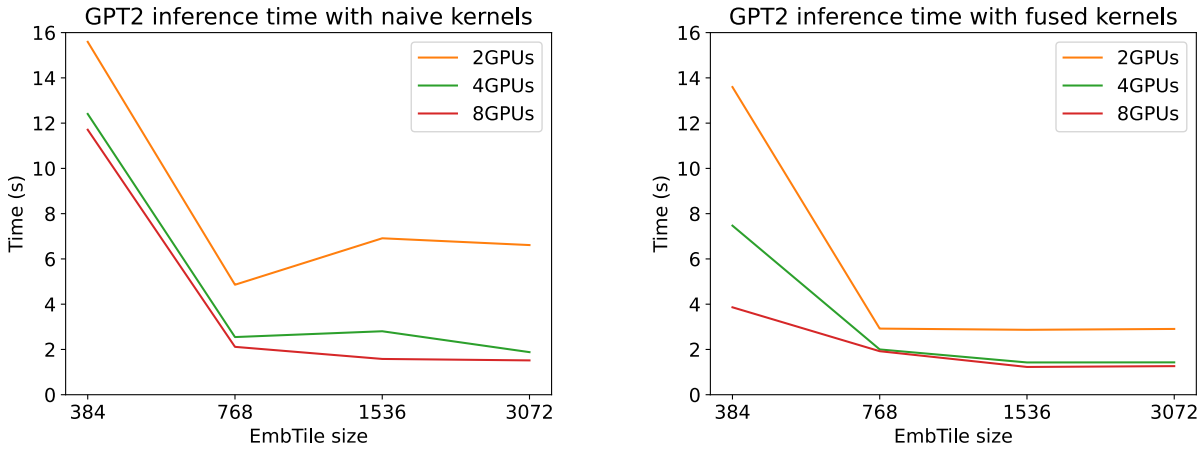


Figure 18: Multi-GPU performance for GPT-2 inference

As a conclusion, fused kernels allow us to use bigger tiles in GPT-2 inference, resulting in better performance. A possible explanation is that despite having bigger tasks to schedule, the reduction of their number allow StarPU scheduler to handle them in a better way. Moreover, having big tiles also reduce the communications between several GPUs as the data used for the computations over one tile can be stored only in one GPU.

Conclusion

In this report, we explored how to leverage StarPU to efficiently train Transformer-based models within the NNTile framework. Our work focused on improving NNTile's task management by addressing key challenges such as task granularity and tiling parameter selection. Firstly, we studied how to parametrize task selection, ensuring that StarPU's scheduler could better optimize resource utilization. By implementing kernel fusion at several levels, we obtained performance improvements. Secondly, we investigated the impact of different tiling sizes on inference latency and throughput, identifying configurations that enable scalable training of large models without excessive computational overhead.

Our experimental evaluations demonstrated that the proposed enhancements led to performance gains compared to NNTile's default implementations. Moreover, we achieved very good performance by using 8 GPUs for computing GPT-2 inference. These improvements highlight the importance of tiling and task decomposition in achieving efficient deep learning training on multi-GPU architectures.

Future work could extend these optimizations to Transformers training by developing specific backward kernels, explore backward tiling strategies, and further integration with advanced StarPU features. The insights gained from this internship contribute to ongoing efforts to optimize Transformer training and could serve for further work in the use of block-wise parallelism in deep learning frameworks.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017.

[3] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018.

[4] H. Zhang, H. Song, S. Li, M. Zhou, and D. Song, "A survey of controllable text generation using transformer-based pre-trained language models," *ACM Comput. Surv.*, vol. 56, Oct. 2023.

[5] J. Ni, T. Young, V. Pandelea, F. Xue, and E. Cambria, "Recent advances in deep learning based dialogue systems: a systematic survey," *Artificial Intelligence Review*, vol. 56, pp. 3055–3155, Apr. 2023.

[6] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[7] A. Mikhalev, A. Katrutsa, K. Sozykin, G. Karpov, and D. Bershatsky, "Nntile." `https://github.com/nntile/nntile`.

[8] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[9] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023.

[10] OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, R. Avila, I. Babuschkin, S. Balaji, V. Balcom, P. Baltescu, H. Bao, M. Bavarian, J. Belgum, I. Bello, J. Berdine, G. Bernadett-Shapiro, C. Berner, L. Bogdonoff, O. Boiko, M. Boyd, A.-L. Brakman, G. Brockman, T. Brooks, M. Brundage, K. Button, T. Cai, R. Campbell, A. Cann, B. Carey, C. Carlson, R. Carmichael, B. Chan, C. Chang, F. Chantzis, D. Chen, S. Chen, R. Chen, J. Chen, M. Chen, B. Chess, C. Cho, C. Chu, H. W. Chung, D. Cummings, J. Currier, Y. Dai, C. Decareaux, T. Degry, N. Deutsch, D. Deville, A. Dhar, D. Dohan, S. Dowling, S. Dunning, A. Ecoffet, A. Eleti, T. Eloundou, D. Farhi, L. Fedus, N. Felix, S. P. Fishman, J. Forte, I. Fulford, L. Gao, E. Georges, C. Gibson, V. Goel, T. Gogineni, G. Goh, R. Gontijo-Lopes, J. Gordon, M. Grafstein, S. Gray, R. Greene, J. Gross, S. S. Gu, Y. Guo, C. Hallacy, J. Han, J. Harris, Y. He, M. Heaton, J. Heidecke, C. Hesse, A. Hickey, W. Hickey, P. Hoeschele, B. Houghton, K. Hsu, S. Hu, X. Hu, J. Huizinga, S. Jain, S. Jain, J. Jang, A. Jiang, R. Jiang, H. Jin, D. Jin, S. Jomoto, B. Jonn, H. Jun, T. Kaftan, Łukasz Kaiser, A. Kamali, I. Kanitscheider, N. S. Keskar, T. Khan, L. Kilpatrick, J. W. Kim, C. Kim, Y. Kim, J. H. Kirchner, J. Kiros, M. Knight, D. Kokotajlo, Łukasz Kondraciuk, A. Kondrich, A. Konstantinidis, K. Kosic, G. Krueger, V. Kuo, M. Lampe, I. Lan, T. Lee, J. Leike, J. Leung, D. Levy, C. M. Li, R. Lim, M. Lin, S. Lin, M. Litwin, T. Lopez, R. Lowe, P. Lue, A. Makanju, K. Malfacini, S. Manning, T. Markov, Y. Markovski, B. Martin, K. Mayer, A. Mayne, B. McGrew, S. M. McKinney, C. McLeavey, P. McMillan, J. McNeil, D. Medina, A. Mehta, J. Menick, L. Metz, A. Mishchenko, P. Mishkin, V. Monaco, E. Morikawa, D. Mossing, T. Mu, M. Murati, O. Murk, D. Mély, A. Nair, R. Nakano, R. Nayak, A. Neelakantan, R. Ngo, H. Noh, L. Ouyang, C. O'Keefe, J. Pachocki, A. Paino, J. Palermo, A. Pantuliano, G. Parascandolo, J. Parish, E. Parparita, A. Passos, M. Pavlov, A. Peng, A. Perelman, F. de Avila Belbute Peres, M. Petrov, H. P. de Oliveira Pinto, Michael, Pokorny, M. Pokrass, V. H. Pong, T. Powell, A. Power, B. Power, E. Proehl, R. Puri, A. Radford, J. Rae, A. Ramesh, C. Raymond, F. Real, K. Rimbach, C. Ross, B. Rotsted, H. Roussez, N. Ryder, M. Saltarelli, T. Sanders, S. Santurkar, G. Sastry, H. Schmidt, D. Schnurr, J. Schulman, D. Selsam, K. Sheppard, T. Sherbakov, J. Shieh, S. Shoker, P. Shyam, S. Sidor, E. Sigler, M. Simens, J. Sitkin, K. Slama, I. Sohl, B. Sokolowsky, Y. Song, N. Staudacher, F. P. Such, N. Summers, I. Sutskever, J. Tang, N. Tezak, M. B. Thompson, P. Tillet, A. Tootoonchian, E. Tseng, P. Tuggle, N. Turley, J. Tworek, J. F. C. Uribe, A. Vallone, A. Vijayvergiya,

C. Voss, C. Wainwright, J. J. Wang, A. Wang, B. Wang, J. Ward, J. Wei, C. Weinmann, A. Welihinda, P. Welinder, J. Weng, L. Weng, M. Wiethoff, D. Willner, C. Winter, S. Wolrich, H. Wong, L. Workman, S. Wu, J. Wu, M. Wu, K. Xiao, T. Xu, S. Yoo, K. Yu, Q. Yuan, W. Zaremba, R. Zellers, C. Zhang, M. Zhang, S. Zhao, T. Zheng, J. Zhuang, W. Zhuk, and B. Zoph, "Gpt-4 technical report," 2024.

[11] S. Li, F. Xue, C. Baranwal, Y. Li, and Y. You, "Sequence parallelism: Long sequence training from system perspective," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (A. Rogers, J. Boyd-Graber, and N. Okazaki, eds.), (Toronto, Canada), pp. 2391–2404, Association for Computational Linguistics, July 2023.

[12] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," in *Advances in Neural Information Processing Systems* (S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, eds.), vol. 35, pp. 16344–16359, Curran Associates, Inc., 2022.

[13] B. Saha and C. Ye, "The i/o complexity of attention, or how optimal is flash attention?," 2024.

[14] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[15] F. Gu, J. Lu, and C. Cai, "Rpformer: A robust parallel transformer for visual tracking in complex scenes," *IEEE Transactions on Instrumentation and Measurement*, vol. 71, pp. 1–14, 2022.

[16] Z. Gao, S. Zhang, I. McLoughlin, and Z. Yan, "Paraformer: Fast and accurate parallel transformer for non-autoregressive end-to-end speech recognition," 2023.

[17] H. Liu and P. Abbeel, "Blockwise parallel transformer for large context models," 2023.

[18] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl, "Measuring the effects of data parallelism on neural network training," *Journal of Machine Learning Research*, vol. 20, no. 112, pp. 1–49, 2019.

[19] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," 2019.

[20] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," 2020.

[21] E. Slaughter, W. Wu, Y. Fu, L. Brandenburg, N. Garcia, W. Kautz, E. Marx, K. S. Morris, Q. Cao, G. Bosilca, S. Mirchandaney, W. Leek, S. Treichlerk, P. McCormick, and A. Aiken, "Task bench: A parameterized benchmark for evaluating parallel runtime performance," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2020.

[22] NVIDIA, "cublaslt." `https://docs.nvidia.com/cuda/cublas/#using-the-cublaslt-api`.

[23] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science* (I. I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan, eds.), vol. 367 of *Communications in Computer and Information Science*, pp. 3–20, Springer International Publishing, 2013.