Towards EXtreme scaleTechnologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale

http://textarossa.eu

WP4 Tool chain for heterogeneous multi-node HPC platform

D4.2 Efficient Memory Management strategies for DNNs at node level

Revised version

This project has received funding from the European Union's Horizon 2020 research and innovation programme, EuroHPC JU, grant agreement No

TEXTAROSSA

Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale

Grant Agreement No.: 956831

Deliverable: D4.2 Efficient Memory Management strategies for DNNs at node level

Project Start Date: 01/04/2021 **Duration**: 36 months
Coordinator: *AGENZIA NAZIONALE PER LE NUOVE TECNOLOGIE, L'ENERGIA E LO SVILUPPO ECONOMICO SOSTENIBILE - ENEA, Italy.*

Deliverable No	D4.2 (revised)
WP No:	WP4
WP Leader:	INRIA
Due date:	M18 (October 31, 2022)
Revision date:	M27 (revised)

Dissemination Level:

PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

This project has received funding from the European Union's Horizon 2020 research and innovation programme, EuroHPC JU, grant agreement No

DOCUMENT SUMMARY INFORMATION

Project title:	Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale
Short project name:	TEXTAROSSA
Project No:	956831
Call Identifier:	H2020-JTI-EuroHPC-2019-1
Unit:	EuroHPC
Type of Action:	EuroHPC - Research and Innovation Action (RIA)
Start date of the project:	01/04/2021
Duration of the project:	36 months
Project website:	textarossa.eu

WP4 Tool chain for heterogeneous multi-node HPC platform

Deliverable number:	D4.2
Due date:	M18
Actual submission date:	M27 (revised)
Editor:	Bérenger Bramas
Authors:	Olivier Beaumont, Lionel Eyraud-Dubois, Samuel Thibault
Work package:	4
Dissemination Level:	Public
No. pages:	20
Authorized (date):	20/05/2023
Responsible person:	Bérenger Bramas
Status:	Plan Draft Working Final **[Submitted]** Approved

Revision history:

Version	Date	Author	Comment
0.1	2022-10-18	Olivier Beaumont	Draft structure + V0
0.2	2022-19-19	Lionel Eyraud-Dubois	Details Section 7
0.3	2022-20-10	Samuel Thibault	Proofreading
0.3	2022-20-10	Olivier Beaumont	Proofreading
1.1	2023-12-04	Olivier Beaumont	Revised version

Quality Control:

Checking process	Who	Date
Checked by internal reviewer		
Checked by Task Leader	-	-
Checked by WP Leader	Bérenger Bramas	2023-04-28
Checked by Project Coordinator	Massimo Celino	2023-05-20

COPYRIGHT

ACKNOWLEDGEMENTS

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement no 956831. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Italy, Germany, France, Spain, Poland.
Please see http://textarossa.eu for more information on the TEXTAROSSA project.
The partners in the project are AGENZIA NAZIONALE PER LE NUOVE TECNOLOGIE, L'ENERGIA E LO SVILUPPO ECONOMICO SOSTENIBILE (ENEA), FRAUNHOFER GESELLSCHAFT ZUR FOERDERUNG DER ANGEWANDTEN FORSCHUNG E.V. (FHG), CONSORZIO INTERUNIVERSITARIO NAZIONALE PER L'INFORMATICA (CINI), INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE (INRIA), BULL SAS (BULL), E4 COMPUTER ENGINEERING SPA (E4), BARCELONA SUPERCOMPUTING CENTER-CENTRO NACIONAL DE SUPERCOMPUTACION (BSC), INSTYTUT CHEMII BIOORGANICZNEJ POLSKIEJ AKADEMII NAUK (PSNC), ISTITUTO NAZIONALE DI FISICA NUCLEARE (INFN), CONSIGLIO NAZIONALE DELLE RICERCHE (CNR), IN QUATTRO SRL (in4). Linked third parties of CINI are POLITECNICO DI MILANO (CINI-POLIMI), Università di Torino (CINI-UNITO) and Università di Pisa (CINI-UNIPI); linked third party of INRIA is Université de Bordeaux; in-kind third party of ENEA is Consorzio CINECA (CINECA); in-kind third party of BSC is Universitat Politècnica de Catalunya (UPC).
The content of this document is the result of extensive discussions within the TEXTAROSSA © Consortium as a whole.

DISCLAIMER

Table of contents

Executive Summary

This deliverable provides a survey of the literature on the optimization of memory management in the context of training on deep neural networks and the use of parallelism for DNN inference to maximize throughput and minimize latency. Our goal is to broaden the scope of applications related to the use of dynamic runtime schedulers to inference in DNNs, as well as to explore a context where a large number of independent tasks need to be processed, and where the use of heterogeneous resources allows to improve performance, both in terms of throughput and latency.

In the context of training, due to the increase of the size of the models, relying on a single resource (either a multicore CPU, a GPU, or an FPGA) is not possible without implementing specific memory saving techniques. This is especially true for the newer transformer-based models. These memory-saving techniques either induce (i) more computations (re-materialization or checkpointing), or (ii) more data exchanges between the (limited) memory of the accelerators and the memory attached to the CPU (offloading), or (iii) a distribution of the model over several resources which in turn induces extra communications. We distinguish between approaches that reduce the memory associated with the storage of intermediate data (activations) and those that reduce the memory associated with the storage of network weights (and optimizer states). For each approach, we detail the possible approaches and their current limitations in terms of the shape of the networks, and we analyze their overhead in terms of data transfer.

In the context of inference, we consider the possible use of dynamic runtimes (StarPU and OmpSs), the possibility of relying on a task-based model to process a stream of inferences. A case of special interest is the one where the complete model is too large to be stored in the memory of a single resource, but small enough to be replicated on a TEXTAROSSA node. The parallelism can also be used to decrease latency (relying to internal parallelism inside models) and it can be combined with compression and mixed precision techniques.

1 Introduction

WP4 focuses on the middle layer of the TEXTAROSSA project. It is tied to hardware and software, making the bridge between hardware features and software interfaces. Consequently, most WP4 activities are highly coupled with other WPs. The main activities of WP4 include the improvement of the streaming and task-based programming models to computing nodes with FPGAs. In this report we concentrate on the use of runtime systems in the context of the training of Deep Neural Networks (DNNs). The current document surveys the literature and summarizes state of the art memory saving techniques. It also explores the possible use of task-based systems to dynamically handle the communications and the allocation of inference and training tasks in the context of a single heterogeneous node.

Among the techniques covered in this paper to minimize memory consumption, we will consider re-materialization, offloading and model parallelism.

Re-materialization and offloading are techniques used to limit the memory occupied by activations. The re-materialization consists in deleting during the forward phase some of the computed activations, and then recomputing them (during the backward phase). Offloading consists in transferring some activations from the accelerator's memory (generally limited in size) to the memory of the GPP and in prefetching them when they are needed during the backward phase.

Model parallelism is a technique used to limit memory occupied by the model weights and by distributing them over several accelerators. In this case, each input data is processed on several computation resources and activations are transferred from one resource to another.

In general, the techniques considered in this deliverable allow to save memory, at the cost of redundant computations (in the case of re-materialization) or data exchanges (offloading and model parallelism).

2 Re-materialization based Strategies in Automatic Differentiation

Adjoints computation is a numerical method for computing the gradient of a function, which may be complex. This method is at the core of many scientific applications, from climate and ocean modeling [2] to oil refinery [14]. In addition, the structure of the underlying dependence graph is also at the basis of the backpropagation step of machine learning [35], and thus the models considered in this report are based on it.

Storage has been one of the key issues with the computation of adjoints: it is required to keep all the intermediate data to compute the final gradient, but it is possible to recompute them.

Therefore, the computation of adjoints has always been a trade-off between re-computations and memory requirements [23].

When one type of limited memory is available, authors of [26] showed the optimality of a binomial approach that was later implemented under the name Revolve [24]. In the latter paper, closed form formulas providing the exact position of saved data have even been proposed for homogeneous chains (each operation has the same duration and memory cost). When computation times are heterogeneous, but data sizes are identical, an optimal checkpointing strategy can be obtained with Dynamic Programming [25]. The problem of adjoint computations has received an increasing attention in the recent years with the introduction of a second level of storage of infinite capacity but with access (write and read) costs [60, 5, 4, 58, 52]. Indeed, with the increase in the problem size, the memory was not sufficient anymore to solve the problems in a reasonable time. Hence solutions have started considering the usage of disks to store some of the intermediary data. Several works have considered this problem. In [60], a first heuristic was presented that applies the schedule provided by Revolve, where the checkpoints that stay idle for the longest period are stored on disk (level 2 storage). Some implementations (for example [52]) are based on two-levels checkpointing strategy: the first pass (forward mode) of the adjoint graph checkpoints periodically to disk (level 2), then the second pass (reverse mode) reads those disk checkpoints one after the other and uses Revolve with only memory (level 1) checkpoints. The main parameter (period used for the forward checkpointing) can be chosen by the user. The algorithm designed in [5] can solve this problem optimally. In a subsequent work, authors of [4] showed that the optimal solution is weakly periodic, meaning that the number of forward computations performed between two consecutive checkpoints into the second level of storage is always the same except for a bounded number of them. More recently, they extended this result for a hierarchical memory architecture with an arbitrary number of storage levels [29].

In [7], the classical results of [24] are extended to meet the requirements of more complex adjoint computations emerging in DNN graphs of Siamese Neural Networks [13, 18, 45] and Cross-Modal-Embeddings [44, 47]. They represent a class of graphs that has a shape of multiple chains joint together at the end by the last operation, which in case of Deep Learning is a loss function. This work shows that dynamic programming is still applicable for the new type of graph, but its complexity grows exponentially with several chains.

3 Re-materialization for DNNs

When a DNN represents a single chain of layers, the computation of the gradients in the training phase is similar to Automatic Differentiation (AD). The checkpointing strategies used in AD to reduce memory consumption are known in AI as Re-materialization or gradient checkpointing strategies. Re-materialization is the method that relies on re-computations to reduce the memory footprint of a given fixed model or architecture, while obtaining the exact same output of the training phase.

For example, the authors of [51] show, for a popular neural network like DenseNet, that using shared memory storages and recomputing concatenation and batch normalization operations during backpropagation help to go from quadratic memory cost to linear memory cost for storing feature maps. Along the same idea, re-implementations of some commonly used layers like batch normalization have been proposed [57]. In the latter case, memory usage is reduced

by rewriting the gradient calculation for this layer so that it does not depend on certain activation values (so that it is no longer necessary to store them). A generic divide-and-conquer approach based on compiler techniques can perform automatic differentiation for arbitrary programs [59].

The use of re-materialization strategies inspired by AD has recently been advocated for DNNs in several papers [27, 16, 35, 36, 21, 33]. A direct adaptation of the results on homogeneous chains was proposed for the case of Recurrent Neural Networks (RNNs) in [27] but cannot be extended to other DNNs. Apart from this, for practical usage, an implementation of re-materialization exists in PyTorch [1], based on a simple periodic and single-pass re-materialization strategy that exploits the ideas presented in [16]. In this strategy, the chain is divided in equal-length segments, and only the input of each segment is materialized during the forward phase. This strategy provides non-optimal solutions in terms of throughput and memory usage, because it does not benefit from the fact that more memory is available when computing the backward phase of the first segment (since values materialized for later segments have already been used). This implementation was nevertheless used to process significantly larger models [22].

Some research attempted to adapt re-materialization strategies to Arbitrary Computation Graphs (ACG). On the one hand, a polynomial algorithm is provided in [21] that finds the re-materialization strategy for the forward propagation that minimizes memory used to execute ACG, under the assumption that activation deletion is not allowed during the backward phase (activations can be recomputed only once), which is a very strong and restrictive assumption in practice, especially in the case of deep networks. On the other hand, in the AD literature, the process is fully recursive, allowing the full memory usage throughout the entire training since the released memory can be used later. In what follows, we refer to such solutions as single-pass re-materialization strategies.

A similar problem is considered in [36], where activation deletion during backward propagation is possible, though similarly the framework is restrictive on several points that are crucial in terms of practical performance and applicability. First, the study is limited to unit costs for data. More importantly, the approach described in [36] is based on the computation of a tree-width decomposition of the graph and only derives the minimum computational cost associated with the minimum memory footprint. The minimum memory footprint then depends on the quality of the decomposition, which is an NP-complete problem for which (large) constant approximation algorithms exist. In practice, the problem to be solved is rather to minimize the computational cost while meeting a given memory constraint. Indeed, limiting the search to the smallest possible memory size obviously leads to a significant additional computational cost.

Another closely related approach is Checkmate [33] in which an Integer Linear Program is proposed to solve the re-materialization problem. This program can handle arbitrary graphs by assuming a fixed ordering of the execution and can provide a solution of minimum runtime given a memory limit. However, solving this ILP is very computationally expensive and does not converge in a reasonable time as soon as the network exceeds a few dozen layers. Its rational approximation, however, can be easily found, but may push memory usage above memory limits.

At last, other approaches finely control the tradeoff between memory and computation. In [37], the authors also consider a general ACG framework. Their work can be seen as a generalization

of [16] algorithm to ACGs. More specifically, their goal is to decompose the ACG into groups of nodes and during the forward phase, only the boundaries between groups are materialized. Then, during the backward phase, to perform the gradient computations of a group, it is required to recompute all the activations of the group using its input saved boundary, and then the backward phase is performed without additional recomputing operations. On the one hand, the advantage of this approach is that it is tractable for ACGs using dynamic programming. On the other hand, as in [16] and [21], the search is restricted to single-pass re-materialization strategies.

In [34], the authors proposed Dynamic Tensor Re-materialization that dynamically choose which activations should be discarded and then recomputed at runtime. Still, it is based on a heuristic approach that encourages to discard tensors that have large memory and staleness costs and that can be easily recomputed while allowing cheap re-computations of other tensors as well. This heuristic showed good results, though optimal static re-materialization methods remain more reliable, considering that execution times and memory costs of layers normally do not change much over iterations.

Rotor (https://gitlab.inria.fr/hiepacs/rotor) is the first attempt to precisely model heterogeneity and more importantly the ability, offered in DNN frameworks, to combine two types of activation savings, by either storing only the layer inputs (as done in AD literature), or by recording the complete history of operations that produced the outputs (as available in autograd tools). For this model, a static algorithm is proposed with an optimality proof, based on dynamic programming. This algorithm manages to find the best schedule in polynomial time.

4 Offloading

Offloading is a potentially complementary approach to re-materialization that consists in offloading some of the forward activations from the memory of the GPU to the memory of the CPU, which is expected to be much larger [55, 6]. In [55], the authors propose a simple and effective mechanism of Memory Virtualization, that nevertheless introduces unnecessary idle time by enforcing some synchronization between data transfers and computations of later forward activations. This approach has been later improved in [6] by the design of techniques to deal with memory fragmentation. Nevertheless, in both papers, the algorithmic strategies to decide which activations to offload into the main memory are relatively straightforward. Proposed strategies consist in trying to offload either all activations or only those that correspond to convolutional layers. Indeed, convolutional layers are known to induce a large computational time with respect to their input size, which make them good candidates to overlap offloading and processing.

Several frameworks offer improvements over this first attempt. In order to reduce the overhead induced by communications, some authors [56] recommend adding compression to decrease the communication time, while others [38] design a memory-centric architecture to help with data transfers. Memory Virtualization was further considered in [46, 39, 30, 65]. In [46, 39, 30], the authors implement memory virtualization by manipulating the computational graphs and inserting special operations called swap in and swap out that send the activations in and out of the device memory. Such an approach can be applied to any ACG that represent neural network training graphs. The authors of [39] improve the candidate selection and prefetching mechanisms by introducing thresholds to filter out different possibilities. Moreover, some

works try to combine Offloading with other memory optimizing techniques. Memory Swapping and Memory Pooling are implemented together in [65], where candidates for swapping are found by assigning priority scores to all activations.
As a complement to these practical approaches, in [8] a theoretical analysis of the underlying optimization problem is proposed: which data to offload and how to schedule transfers. An extension to perform weight offloading can be found in [12].

5 Combination of Re-materialization and Offloading

The works, combining both approaches, are relatively recent, though the idea comes naturally from the fact that they serve the same purpose, while they make use of different resources. The speed-centric re-materialization from [16] enhanced with memory-centric re-materialization (discards activations of every segment all the time) was combined with the simple offloading approach from [55] in [62]. Then, the authors in [54] also use the re-materialization scheme of [16] with a possibility of further offloading saved checkpoints to the CPU if re-materialization only is not enough to perform training under memory constraint.

Another approach that combines re-computations and data offloading from GPU memory to CPU memory was proposed in [53]. This approach is especially useful in the case where the size of the activations is small compared to the size of the model, which is the case in most recent NLP models. In this case, the network weights are offloaded to the CPU memory [12], that serves as a parameter-server host.

In [9], the goal is to find simultaneously optimal Re-materialization and Offloading strategies that could for the makespan in the case of DNNs represented by heterogeneous complex chains. In this context, it is assumed that the model weights stay in GPU and the only activations can be moved to the CPU memory.

6 Pipelined Model Parallelism

When using Model Parallelism [17], the different layers of a network are spread over different resources, so that the storage of DNN weights and activations is shared between the resources. In Model Parallelism, only activations should be communicated, and transfers take place just between layers assigned to different processors, which adds up to a low total amount of data movements with respect to other types of parallelisms. Despite that, the scalability of the method is poor because of chain connections in DNN computational graph that force a sequential execution of all the tasks.

The execution within Model Parallelism can be accelerated if several mini batches are pipelined, and thus several training iterations are active at the same time, helping to keep computing resources busy most part of the time. The practical use of Pipelined Model Parallelism is nevertheless a delicate issue, and the analysis of the induced memory needs is complex. In [31], it is proposed to split the training batch into several mini batches, which are then pipelined through the layers of the network (and the different computing resources). Once the forward and backward phases have been computed on all these mini batches, the weights are then updated. This approach is fairly simple to implement but has the disadvantage of leaving the computational resources largely idle (e.g., after the first resource has executed its

forward operations on the pipelined mini batches, it has to wait until the corresponding backward operations become available to complete the iteration). The Pipedream approach proposed in [48] improves this training process, by only enforcing that the forward and backward tasks use the same model weights for a given mini batch. Such a weakened constraint on the training process allows Pipedream to achieve a much better utilization of the processing resources, but the asynchronous updates affect badly the overall convergence of the training.

Despite its advantages, Pipedream has several issues: (i) degraded convergence because of weight staleness that is non-uniform with respect to different stages, (ii) poor memory management because of redundant weight and activation copies produced by non-optimal schedule, (iii) inferior load balancing being restricted to contiguous allocations, (iv) not suitable for heterogeneous GPUs.

The poor convergence of asynchronous methods has been addressed in several papers. It is caused by weight staleness when the delayed gradients are used to perform an update step. Some works [28, 15] propose to predict weights during forward and backward propagation using the momentum of the gradient. Performing the updates less regularly [28, 49] (in contrast in Pipedream they are done after each backward) helps limiting weight staleness as well. Alternatively, PipeMare [63] proposes to reschedule learning rate depending on the pipeline stage and adapt the model weights for backward so that they are defined by the most recent version of weights and the accumulated weighted difference between the model weights from successive iterations and the stage number. The last method achieves the same convergence rate as Gpipe, while having the same resource utilization as Pipedream without storing multiple copies of the weights.

Another important issue related to Pipedream is the need to keep many copies of the model parameters, which can potentially cancel the benefit of using Model Parallelism. To address this issue, the same methods that help with weight staleness can be used: in [49] the updates are done so that it is possible to keep only two versions of the weights; in [15] two versions of the weights are needed too, but also one gradient and momentum should be stored. The inefficient memory utilization by Pipedream has been also observed in [32]. Unlike other works, they offer another version of pipelining different from Gpipe and Pipedream. Its principle of work can be described in the following way: once all forward steps on one mini-batch are processed by all GPUs and the first backward of the last stage is done, the same GPU can proceed to the first stage of the next mini-batch by performing its forward and then the remaining forwards of the new mini-batch are executed on the other processors in the reverse order just after the backwards of the preceding mini-batch. This allows GPUs to use memory immediately after it is released during backward steps. In general, it uses memory more efficiently, though memory itself is not considered as a constraint.

Contiguous allocations can be also a bottleneck that hinders a throughput. The authors of [19] offer a method suitable for fine-tuning large models. They obtain non-contiguous allocations, by coarsely building stages that have a high ratio of computation time with respect to communication time. These stages can be further allocated to any device, allowing more than one stage per processor. To find non-contiguous allocation for ACGs, the authors of [61] propose two Integer Linear Programs (ILPs) (one minimizes latency, the other one maximizes throughput for a steady state situation) and a dynamic program. The obtained solutions are optimal for inference and can be adapted for training, though those methods do not consider the pipelining nature of model parallelism and scheduling, which have a significant influence on the peak memory usage.

Some researchers have worked on extending the results of Pipedream to heterogeneous computing clusters and heterogeneous communication links [64, 50, 43]. Finding the optimal load balance and schedule for heterogeneous settings is a difficult task, thus all of them rely on some simplifications and heuristics. To solve issues in the case of high communication costs and heterogeneous networking, the authors of [64] proposed an updated dynamic programming strategy that assumes no overlap between computations and communications. The HetPipe proposal [50] considers a different way of combining Data and Model Parallelism, in which nodes may contain different GPUs. The idea of HetPipe is to heuristically split the GPUs into virtual workers that may contain heterogeneous GPUs and use Data Parallelism between virtual workers. Model Parallelism is used inside the virtual workers, based on a simplified ILP that assumes no overlap between computation and communication. Pipelined Model Parallelism in [43] is done with a help of Deep Reinforcement Learning.

Other extensions of Pipedream explore different ways of combining Model Parallelism with other types of parallelism [50, 20, 40, 41]. In the DAPPLE framework [20], Model Parallelism is implemented alongside Data Parallelism. There, the focus is on the case of several nodes, each equipped with several GPUs. DAPPLE extends Pipedream by allowing more possibilities to map a stage of the DNN to GPUs located in several nodes. The assignment problem is solved without taking memory constraints into account. Furthermore, [40] does Hybrid Parallelism, using Tensor Slicing, Data and Model Parallelism, finding job allocation with dynamic programming. However, this method does not consider memory constraints.

Transformers offer a new dimension for pipelined parallelism. In [41], the pipelining is not performed through micro-batching. Instead, they pipeline the tokens in the input sequence. Such approach manages to significantly accelerate the training of GPT-3. Their solution is based on dynamic programming without memory considerations.

The work in [10] carefully investigates the limitations of Pipedream. It carefully estimates the effect of the chosen schedule on the peak memory usage of the pipeline. It also evaluates to which extent non-contiguous allocations can be advantageous with respect to contiguous ones. Therefore, in [10] an Integer Linear Program is described that finds simultaneously the optimal load balance based on non-contiguous allocations and the optimal schedule, considering all sources of memory consumption. The MadPipe heuristic, proposed in [11], is based on dynamic programming that also combines non-contiguous allocations with scheduling considerations to find the best load balancing. These methods can be combined with [28, 15, 49, 63] to improve the training convergence.

7 Use of Task-Based Systems for DNN Inference.

In this Section, we explore the possibility of using StarPU [3] (https://starpu.gitlabpages.inria.fr) to increase the throughput obtained in inference tasks. Indeed, the new networks used for the most recent applications, such as GPT-like networks, induce increasingly high resource costs. These resource costs encompass both the computational workload and the storage requirements. Such networks have been scaled to such many parameters that storing them uses a significant amount of memory, and that even performing the inference operation requires a large number of computations. In this context, it is essential to perform the inference in a parallel way, to increase the throughput of the operation, defined as the number of inputs processed per second.

This task raises challenges both at the system level and at the algorithmic level. At the system level, it is necessary to design a software architecture that allows a network (typically trained within PyTorch) to be exported as a task graph composed of StarPU tasks. To accomplish this conversion, we rely on the ONNX library [42] to retrieve tasks and dependencies, and on the ONNX Runtime library https://onnx.ai to provide efficient implementations of all computational kernels. The ONNX format provides a unified interface for all possible task types in a Deep Neural Network, and the ONNX Runtime provides implementations for a large set of computing devices. This first part of the work was done as part of Jean-François David's PhD thesis and is now mostly functional.

The second part is more algorithmic in nature and consists in designing optimization algorithms to solve the placement problem. To define the placement problem, we assume that we are given (i) a set of tasks with their internal dependencies, their associated computational costs and the volumes of input and output data and (ii) a set of potentially heterogeneous computational resources, characterized by their memory sizes and their computational speeds for the different tasks. The set of tasks represents all the layers of the Deep Neural Network, and the goal is to be able to process as many inference operations per second as possible. This is done by assigning layers to the computational resources, which incurs a storage requirement for all the parameters of that layer. In addition, data dependencies may incur communication delays to transfer the data from one resource to another. There is thus a tradeoff, where assigning layers to more computational resources allow the system to perform more operations in parallel, but is limited by the available memory on the resources and by the communication capabilities. Additionally, the assignment needs to consider the fact that some computational resources are more efficient for some computational tasks than others. The second part of Jean-François David's PhD thesis will be to design and analyze new and efficient assignment algorithms, which would lead to an optimized resource usage in a Textarossa node.

8 Positioning with respect to TEXTAROSSA KPIs.

Our objective in TEXTAROSSA is to develop data placement strategies and computational algorithms that make the best use of the heterogeneity of the computational resources developed in the project. SOTA solutions for inference consist in performing each complete inference on a single GPU, which induces several limitations. First, as models become larger and larger (especially transformer-based models), being constrained to store all network weights on a single accelerator is a major limitation. Second, from an energy point of view, using the most energy-efficient computation resource for each layer can lead to significant gains. In case of load imbalance, the possibility of using mechanisms like DVFS to improve energy consumption by slowing down some computations without affecting latency and throughput is also an opportunity. Finally, the use of multiple resources to perform parallel inference can also help minimize latency by exploiting the possibility of performing several network branches in parallel. We aim to contribute to the project objectives in terms of energy efficiency, sustained application performance and the use of Integrated Development Platforms.

This work is also part of several internal collaborations within TEXTAROSSA. First of all, in an HPC framework in a real environment, it is difficult to predict exactly the performance of the different resources and data transfers, and it appears essential to rely on a dynamic runtime system like StarPU or OmpSs to schedule and distribute tasks. As demonstrated in the case of linear algebra and also explored in WP6 (within MathLib Inria), it is nevertheless necessary to perform static data placement in addition to dynamic scheduling. This is especially true in the

multiple node case, but also as soon as there are strong memory constraints and constraints on the number of communications. If the system cannot reach the constraints in terms of throughput/latency, then it is also interesting to rely on techniques to reduce the precision of the computations. This implies using the possibility to generate kernels with different types of precision of the computations developed within the TEXTAROSSA project). More generally, the use of frameworks such as OmpSs@FPGA allows the generation of efficient kernels for FPGAs. Finally, the possibility of compressing data to reduce the cost of data exchanges and improve latency and throughput is also an interesting perspective. In the framework of TEXTAROSSA, we will first explore the use of dynamic runtimes and the trade-offs between dynamic and static placement on platforms consisting of CPUs and GPUs, in particular the IDV-A.

9 Conclusion

In the context of training, this survey shows that multiple strategies can be envisioned to save memory, at the cost of re-computations, data transfers at the node level, or the use of parallel resources. From the point of view of parallelism, multiple strategies can also be used (data parallelism, model parallelism, kernel parallelism), and they also have an influence on the memory requirements and on the induced communications. It is therefore necessary to know how to solve the allocation problem, for a given type of neural network and a given description of the computational node (computation speed of each resource on each task, speed of the different communication resources, memory sizes). This problem is naturally very complex and is likely to lead to very rigid scheduling solutions, difficult to implement in practice and very sensitive to the slightest modeling errors of the platform. However, we know that the interactions between communication threads, as well as thermal interactions, make accurate predictions extremely difficult. In TEXTAROSSA, our goal is to extract from the solution of this optimization problem a set of task and data placement directives, and then let task-based runtimes such as StarPU or OmpSs dynamically schedule computations and communications to take full advantage of the heterogeneous computational capabilities of TEXTAROSSA's resources and communications capabilities of the nodes. This will be covered during the second phase of Task 4.2 and Task 4.6.

10 References

[1] Periodic checkpointing in Pytorch, 2018. https://pytorch.org/docs/stable/checkpoint.html.

[2] A Adcroft, JM Campin, S Dutkiewicz, C Evangelinos, D Ferreira, G Forget, B Fox-Kemper, P Heimbach, C Hill, E Hill, et al. Mitgcm user manual, 2008.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architec- tures. In European Conference on Parallel Processing, pages 863–874. Springer, 2009.

[4] Guillaume Aupy and Julien Herrmann. Periodicity in optimal hierarchical checkpointing schemes for adjoint computations. Optimization Methods and Software, 32(3):594– 624, 2017.

[5] Guillaume Aupy, Julien Herrmann, Paul Hovland, and Yves Robert. Optimal multistage algorithm for adjoint computation. SIAM Journal on Scientific Computing, 38(3):232–255, 2016.

[6] Shriram S B, Anshuj Garg, and Purushottam Kulkarni. Dynamic memory management for GPU-based training of deep neural networks. In IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE Press, 2019.

[7] Olivier Beaumont, Lionel Eyraud-Dubois, Julien Herrmann, Alexis Joly, and Alena Shilova. Optimal checkpointing for heterogeneous chains: how to train deep neural networks with limited memory. Research Report RR-9302, Inria Bordeaux Sud-Ouest, under Minor Revision at ACM Transactions on Mathematical Software, ACM TOMS, 2022.

[8] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. Optimal GPU-CPU offloading strategies for deep neural network training. In European Conference on Parallel Processing, pages 151–166. Springer, 2020.

[9] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. Efficient Combination of Re-materialization and Offloading for Training DNNs. In NeurIPS 2021 - Thirty- fifth Conference on Neural Information Processing Systems, Virtual-only Conference, France, December 2021.

[10] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. Pipelined model parallelism: Complexity results and memory considerations. In European Conference on Parallel Processing, pages 183–198. Springer, 2021.

[11] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. MadPipe: Memory Aware Dynamic Programming Algorithm for Pipelined Model Parallelism. In ScaDL 2022 - Scalable Deep Learning over Parallel and Distributed Infrastructure - An IPDPS 2022 Workshop, Proceedings of IPDPS W'22, Lyon / Virtual, France, June 2022.

[12] Olivier Beaumont, Lionel Eyraud-Dubois, Alena Shilova, and Xunyi Zhao. Weight Offloading Strategies for Training Large DNN Models. working paper or preprint, February 2022.

[13] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard S¨ackinger, and Roopak Shah. Signature verification using a" siamese" time delay neural network. In Advances in neural information processing systems, pages 737–744, 1994.

[14] Phil Brubaker. Engineering Design Optimization using Calculus Level Methods. 2016.

[15] Chi-Chung Chen, Chia-Lin Yang, and Hsiang-Yun Cheng. Efficient and robust parallel DNN training through model parallelism on multi-GPU platform. arXiv preprint arXiv:1809.02839, 2018.

[16] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. arXiv preprint arXiv:1604.06174, 2016.

[17] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In Advances in neural information processing systems, pages 1223–1231, 2012.

[18] William Du, Michael Fang, and Margaret Shen. Siamese convolutional neural networks for authorship verification. Proceedings, 2017.

[19] Saar Eliad, Ido Hakimi, Alon De Jagger, Mark Silberstein, and Assaf Schuster. Fine-tuning giant neural networks on commodity hardware with automatic pipeline model parallelism. In 2021 USENIX Annual Technical Conference (USENIX ATC 21), pages 381–396. USENIX Association, 2021.

[20] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. Dapple: A pipelined data parallel approach for training large models, 2020.

[21] Jianwei Feng and Dong Huang. Optimal gradient checkpoint search for arbitrary computation graphs, 2018.

[22] Priya Goyal. Pytorch memory optimizations via gradient checkpointing, 2018.

[23] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. Optimization Methods and software, 1(1):35–54, 1992.

[24] Andreas Griewank and Andrea Walther. Algorithm 799: Revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. ACM Transactions on Mathematical Software (TOMS), 26(1):19–45, 2000.

[25] Andreas Griewank and Andrea Walther. Evaluating derivatives: principles and techniques of algorithmic differentiation, volume 105. Siam, 2008.

[26] José Grimm, Lo¨ic Pottier, and Nicole Rostaing-Schmidt. Optimal time and minimum space-time product for reversing a certain class of programs. In Martin Berz, Christian H. Bischof, George F. Corliss, and Andreas Griewank, editors, Computational Differentiation: Techniques, Applications, and Tools, pages 95–106. SIAM, Philadelphia, PA, 1996.

[27] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. In Advances in Neural Information Processing Systems, pages 4125–4133, 2016.

[28] Lei Guan, Wotao Yin, Dongsheng Li, and Xicheng Lu. Xpipe: Efficient pipeline model parallelism for multi-GPU DNN training. arXiv preprint arXiv:1911.04610, 2019.

[29] Julien Herrmann. H-revolve: a framework for adjoint computation on synchronous hierarchical platforms. ACM Transactions on Mathematical Software (TOMS), 46(2):1– 25, 2020.

[30] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the GPU memory limit via smart swapping. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 1341–1355, 2020.

[31] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In Advances in Neural Information Processing Systems, pages 103–112, 2019.

[32] Arpan Jain, Ammar Ahmad Awan, Asmaa M Aljuhani, Jahanzeb Maqbool Hashmi, Quentin G Anthony, Hari Subramoni, Dhableswar K Panda, Raghu Machiraju, and Anil Parwani. Gems: GPU-enabled memory-aware model-parallelism system for distributed DNN training. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–15. IEEE, 2020.

[33] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E. Gonzalez. Checkmate: Breaking the memory wall with optimal tensor re-materialization, 2019.

[34] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic tensor re-materialization. arXiv preprint arXiv:2006.09616, 2020.

[35] Navjot Kukreja, Jan Hu¨ckelheim, and Gerard J Gorman. Backpropagation for long sequences: beyond memory constraints with constant overheads. arXiv preprint arXiv:1806.01117, 2018.

[36] Ravi Kumar, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua Wang. Efficient re-materialization for deep networks. In Advances in Neural Information Processing Systems, pages 15146–15155, 2019.

[37] Mitsuru Kusumoto, Takuya Inoue, Gentaro Watanabe, Takuya Akiba, and Masanori Koyama. A graph theoretic framework of re-computation algorithms for memory-efficient backpropagation. arXiv preprint arXiv:1905.11722, 2019.

[38] Youngeun Kwon and Minsoo Rhu. Beyond the memory wall: A case for memory-centric hpc system for deep learning. In 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 148–161. IEEE, 2018.

[39] Tung D Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. Tflms: Large model support in tensorflow by graph rewriting. arXiv preprint arXiv:1807.02037, 2018.

[40] Jiange Li, Yuchen Wang, Jinghui Zhang, Jiahui Jin, Fang Dong, and Lei Qian. Pipepar: A pipelined hybrid parallel approach for accelerating distributed DNN training. In 2021 IEEE 24th International Conference on Computer Supported Cooperative Work in Design (CSCWD), pages 470–475. IEEE, 2021.

[41] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. arXiv preprint arXiv:2102.07988, 2021.

[42] Wei-Fen Lin, Der-Yu Tsai, Luba Tang, Cheng-Tao Hsieh, Cheng-Yi Chou, Ping-Hao Chang, and Luis Hsu. Onnc: A compilation framework connecting onnx to proprietary deep learning accelerators. In 2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS), pages 214–218. IEEE, 2019.

[43] Yingchi Mao, Zijian Tu, Fagang Xi, Qingyong Wang, and Shufang Xu. Tapp: DNN training for task allocation through pipeline parallelism based on distributed deep reinforcement learning. Applied Sciences, 11(11):4785, 2021.

[44] Javier Marin, Aritro Biswas, Ferda Ofli, Nicholas Hynes, Amaia Salvador, Yusuf Aytar, Ingmar Weber, and Antonio Torralba. Recipe1m: A dataset for learning cross-modal embeddings for cooking recipes and food images. arXiv preprint arXiv:1810.06553, 2018.

[45] Jonathan Masci, Davide Migliore, Michael M Bronstein, and Ju¨rgen Schmidhuber. Descriptor learning for omnidirectional image matching. In Registration and Recognition in Images and Videos, pages 49–62. Springer, 2014.

[46] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. Training deeper models by GPU memory optimization on tensorflow. In Proc. of ML Systems Workshop in NIPS, 2017.

[47] M. Mueller, A. Arzt, S. Balke, M. Dorfer, and G. Widmer. Cross-modal music re-trieval and applications: An overview of key methodologies. IEEE Signal Processing Magazine, 36(1):52–62, Jan 2019.

[48] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: generalized pipeline parallelism for DNN training. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, pages 1–15, 2019.

[49] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel DNN training. In Marina Meila and Tong Zhang, editors, Proceedings of the 38th International Conference on Machine Learning, volume 139 of Proceedings of Machine Learning Research, pages 7937–7947. PMLR, 18–24 Jul 2021.

[50] Jay H. Park, Gyeongchan Yun, Chang M. Yi, Nguyen T. Nguyen, Seungmin Lee, Jaesik Choi, Sam H. Noh, and Young ri Choi. Hetpipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism, 2020.

[51] Geoff Pleiss, Danlu Chen, Gao Huang, Tongcheng Li, Laurens van der Maaten, and Kilian Q Weinberger. Memory-efficient implementation of densenets. arXiv preprint arXiv:1707.06990, 2017.

[52] GC Pringle, DC Jones, S Goswami, SHK Narayanan, and D Goldberg. Providing the archer community with adjoint modelling tools for high-performance oceanographic and cryospheric computation. 2016.

[53] Bharadwaj Pudipeddi, Maral Mesmakhosroshahi, Jinwen Xi, and Sujeeth Bharadwaj. Training large neural networks with constant memory using a new execution algorithm. arXiv preprint arXiv:2002.05645, 2020.

[54] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–16. IEEE, 2020.

[55] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In The 49th Annual IEEE/ACM International Symposium on Microarchitecture, page 18. IEEE Press, 2016.

[56] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W Keckler. Compressing dma engine: Leveraging activation sparsity for training deep neural networks. In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 78–91. IEEE, 2018.

[57] Samuel Rota Bul`o, Lorenzo Porzi, and Peter Kontschieder. In-place activated batch-norm for memory-optimized training of DNNs. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 5639–5647, 2018.

[58] Michel Schanen, Oana Marin, Hong Zhang, and Mihai Anitescu. Asynchronous two-level checkpointing scheme for large-scale adjoints in the spectral-element solver nek5000. Procedia Computer Science, 80:1147–1158, 2016.

[59] Jeffrey Mark Siskind and Barak A. Pearlmutter. Divide-and-conquer checkpointing for arbitrary programs with no user annotation. Optimization Methods and Software, 33(4-6):1288–1330, 2018.

[60] Philipp Stumm and Andrea Walther. Multistage approaches for optimal offline check- pointing. SIAM Journal on Scientific Computing, 31(3):1946–1967, 2009.

[61] Jakub Tarnawski, Amar Phanishayee, Nikhil R Devanur, Divya Mahajan, and Fanny Nina Paravecino. Efficient algorithms for device placement of DNN graph operators. arXiv preprint arXiv:2006.16423, 2020.

[62] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic GPU memory management for training deep neural networks. SIGPLAN Not., 53(1):41–53, February 2018.

[63] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher Aberger, and Christopher De Sa. Pipemare: Asynchronous pipeline parallel DNN training. Proceedings of Machine Learning and Systems, 3, 2021.

[64] Jun Zhan and Jinghui Zhang. Pipe-torch: Pipeline-based distributed deep learning in a GPU cluster with heterogeneous networking. In 2019 Seventh International Conference on Advanced Cloud and Big Data (CBD), pages 55–60. IEEE, 2019.

[65] Junzhe Zhang, Sai Ho Yeung, Yao Shu, Bingsheng He, and Wei Wang. Efficient memory management for GPU-based deep learning systems. arXiv preprint arXiv:1903.06631, 2019.