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

WP4 Tool chain for heterogeneous multi-node HPC platform

D4.8 Framework for efficient CNNs inference on a TEXTAROSSA node

http://textarossa.eu

This project has received funding from the European Union's Horizon 2020 research and innovation programme, EuroHPC JU, grant agreement No 956831

TEXTAROSSA

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

Grant Agreement No.: 956831

Deliverable: D4.8 Framework for efficient CNNs inference on a TEXTAROSSA node

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	D4.8
WP No:	WP4
WP Leader:	INRIA
Due date:	M30
Delivery date:	30/11/2023

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 956831

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:	D4.8					
Deliverable title:	Framework for efficient CNNs inference on a TEXTAROSSA node					
Due date:	M30					
Actual submission date:	2 December 2023					
Editor:						
Authors:	L. Eyraud-Dubois, Antonio Filgueras, Alessandro Lonardo					
Work package:	WP4					
Dissemination Level:	Public					
No. pages:	40					
Authorized (date):	30/11/2023					
Responsible person:	L. Eyraud-Dubois					
Status:	Plan	Draft	Working	Final	Submitted	Approved

Revision history:

Version	Date	Author	Comment
0.1	2023-09-15	Lionel Eyraud-Dubois	Draft structure
0.2	2023-10-20	Lionel Eyraud-Dubois	Section Inference Efficiency
0.3	2023-10-27	Lionel Eyraud-Dubois	Section Training Efficiency
0.4	2023-10-28	Alessandro Lonardo	Section Workflow in RAIDER application
0.5	2023-11-03	Antonio Filgueras	Section Efficient CNN kernels for FPGA
0.6	2023-11-04	Lionel Eyraud-Dubois	Introduction, Conclusion, Executive Summary
0.7	2023-11-17	Carlos Alvarez	Llvm clarifications
0.8	2023-11-29	Lionel Eyraud-Dubois	List of acronyms

Quality Control:

Checking process	Who	Date
Checked by internal reviewer	Michal Kulczewski	27 nov. 2023
Checked by Task Leader	Lionel Eyraud-Dubois	29 nov. 2023

Checked by WP Leader	Bérenger Bramas	30 nov. 2023
Checked by Project Coordinator	Massimo Celino	30 nov. 2023

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

List of Acronyms

Acronym	Definition
AI	Artificial Intelligence
AST	Abstract Syntax Tree
CNN	Convolutional Neural Network
DAG	Directed Acyclic Graph
DNN	Deep Neural Network
GPU	Graphic Processing Unit
ILP	Integer Linear Program
RICH	Ring Imaging CHerenkov
TF	TensorFlow
TPU	Tensor Processing Unit

Executive Summary

This deliverable presents the progress on supporting Deep Neural Network applications within the Textarossa project. We show developments in two different and complementary directions.

In the first part, we present algorithms for resource optimization when executing Convolutional Neural Networks on a heterogeneous server. We study memory optimization when training large neural networks, where we propose an improved re-materialization algorithm with significantly smaller overhead, and we show how combining clever re-materialization strategies with model parallelism can improve the parallel efficiency. We also present on-going work on optimizing the resource usage for the inference process, with progress both on allocation algorithms and on a prototype task-based implementation based on the StarPU runtime.

In the second part, we present developments within the OmpSs@FPGA framework to better support the DNN use-case. We show an optimized implementation of a convolution kernel within the framework which outperforms state-of-the-art CPU implementations. We also introduce support within the OmpSs@FPGA framework for mixed precision data types and for the newer C++ 11 standard. Thanks to this new support, we developed an initial porting of the RAIDER inference application to the OmpSs task-based programming model.

1. Introduction

The unprecedented availability of data, computation and algorithms have enabled a revolution of Artificial Intelligence (AI), as seen in Convolutional Neural Networks (CNNs) for vision, more recently Transformers and Large Language Models, resulting in revolutionary applications such as automatic computer vision, ChatGPT and generative AI. Deep Learning approaches have enabled a large spectrum of new applications in many businesses with their ability to process vast amounts of data and extract meaningful patterns. In healthcare, deep learning is used for disease diagnosis, drug discovery, and personalized medicine. In finance, it aids in fraud detection, algorithmic trading, and risk management. Deep learning also powers advancements in autonomous vehicles, enabling them to perceive the environment and make real-time decisions. In natural language processing, it drives chatbots, language translation, and sentiment analysis, enhancing human-computer interaction. Additionally, it plays a crucial role in image and speech recognition, powering applications like facial recognition systems and voice assistants. Furthermore, deep learning is utilized in industrial automation for predictive maintenance and quality control. Its versatile applications continue to grow, shaping the future of technology across diverse sectors.

Training large neural network models is usually performed at a large scale, using a large number of computing nodes to process the vast amount of data samples required to obtain good accuracy. However, optimizing at the scale of a single server is relevant for several reasons. First, the efficiency of a large-scale execution relies on having efficient kernels and algorithms running on each of the nodes, so that the gains obtained by optimizing at the server level will result in similar gains when running at scale. Secondly, many other use-cases exist beyond training new neural network models from scratch. Once a large model is trained, it is very common to specialize it for more specific tasks, by providing a smaller set of data tailored to the needs of a particular application. This process is called fine-tuning, and typically requires significantly less computing power than the main training procedure. In addition, another important aspect of deep learning is in the inference part. Once the model has been trained for the required task, the inference process consists in answering requests, where new samples are fed into the network to obtain a prediction according to the current model parameters. This process also requires typically much fewer resources than training. Both use cases, fine-tuning and inference, are less resource-heavy than training from scratch but happen typically much more frequently. Efficiency gains from optimizing these use-cases on a single server can thus be very beneficial.

Experts from the Deep Learning community typically use python frameworks such as PyTorch [10] to quickly develop and evaluate new models in a rapidly evolving field. These frameworks often need to trade some resource efficiency to obtain the inter-operability and ease-of-use required for such rapid prototyping. In this deliverable, we try to provide expertise from the High-Performance Computing community to typical deep learning use-cases.

When designing a TEXTAROSSA node, it is thus very natural to consider supporting Deep Learning applications as a possible use-case. In this document, we present some contributions in this direction, with two very different but complementary approaches. In Section 2, we explore algorithmic techniques to improve the efficiency of resource usage when using Convolutional Neural Network applications at the server level. In Section 3, we discuss the technical aspect of developing a set of efficient CNN kernels for heterogeneous architectures enhanced with FPGA. In Section 4, we present a workflow for the deployment of CNNs on FPGA that is used in the RAIDER application and was presented in Deliverable 6.2.

The contributions of this Deliverable are:

- An improved re-materialization strategy has been designed for reducing memory usage when training CNNs on a single GPU.
- A combination of re-materialization with model parallelism has been proposed for a better parallel efficiency when training on multiple GPUs.
- An ongoing work is underway for increasing the efficiency of performing inference requests on a heterogeneous server made of a multicore CPU and GPUs.
- A 2d convolution kernel has been ported to FPGA devices using the OmpSs task-based programming model, with successive performance improvements, reaching competitiveness with CPU state-of-the-art implementations (like PyTorch).
- The OmpSs@FPGA framework has been evolved to support mixed precision data types usually present in CNN implementations.
- An improvement of the framework to support the newer C++ 11 standard has been reached, allowing an initial porting of the RAIDER inference application to the OmpSs task-based programming model.

This deliverable is related to the following project objectives as stated in the DoA:

Objective 1 - Energy efficiency. DNN executions are power hungry and one of the main sources of computing power usage nowadays. Executing in FPGA has been demonstrated to be competitive with other computing platforms in terms of energy efficiency. Improving resource usage for DNN applications and being able to easily port them to FPGAs using the OmpSs@FPGA programming model will help to improve inference and training energy efficiency.

Objective 2 - Sustained application performance. The aim of the work presented in this deliverable is to improve the performance obtained when executing DNN applications over the IDV-E platform by using better resource allocation algorithms, by improving the framework and also by improving the task scheduling through the use of the Fast Task Scheduler developed in Task 2.5. In this deliverable it is also shown how CNN kernels performance can be improved in FPGAs.

Objective 4 - Seamless integration of reconfigurable accelerators. One of the main concepts behind OmpSs@FPGA is to allow the seamless integration of reconfigurable accelerators. In this deliverable we study how to integrate accelerators for CNNs.

Objective 5 - Development of new IPs. Although the IPs developed here are designed in HLS the CNN kernel developed by BSC will be fully available to the public and the source code is even listed in this deliverable. All code developed by Inria will also be fully available to the public.

The work developed here is also related to the strategic goals of the project as follows:

Strategic Goal #2: Supporting the objectives of EuroHPC as reported in ETP4HPC's Strategic Research Agenda (SRA) for open HW and SW architecture. The StarPU and OmpSs@FPGA software frameworks are open source and publicly available as all the developments done in this project.

Strategic Goal #3: Opening of new usage domains. The work developed in this section is strongly aligned with this strategic goal as although GPUs and FPGAs are already used to perform both CNN training and inference the usual approach doesn't involve the use of a task-based programming model to do so. The advantages of task-based programming models in general and StarPU and OmpSs@FPGA in particular (mainly programmability, portability and performance) would pave the way to new developments in the DNN world if the research is successful.

2. Efficiency of CNNs at the server level

We separate this section into two parts: we first discuss improving efficiency during the training process and then present ideas for improving the efficiency of handling inference requests on an already trained neural network.

2.1. Training Efficiency

The landscape of Neural Networks has evolved very quickly, from the first vision networks like ResNet-50 to Natural Language Processing transformer-based models like GPT. This evolution has led to increasingly better results at the cost of tremendous resource requirements. One of these resource requirements is memory usage during training, which comes both from the number of parameters of the models and the size of the activations that must be kept in memory to perform back-propagation. Since training is usually performed on computing resources such as GPUs or TPUs on which memory is limited; these memory requirements often become a limitation that needs to be addressed.

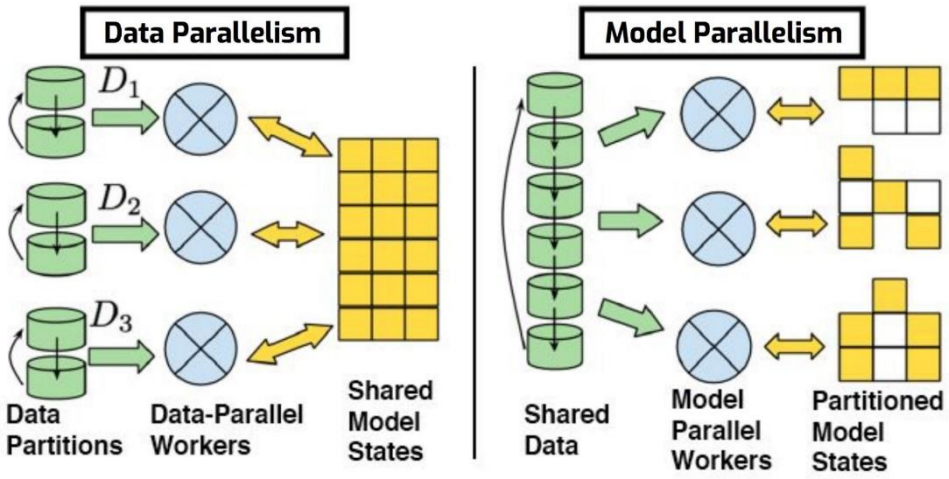


Figure 2.1: Data versus model parallelism (source: Data vs Model Parallelism in TensorFlow, Illia Polosukhin)

The first category of solutions consists in relying on parallelism (see Figure 2.1). Data parallelism [16] refers to the distribution of the memory related to the activations, at the cost of exchanging the network weights between the different resources using collective communications which can be expensive for networks with very large weights such as GPT for example. On the contrary, model parallelism [9] consists in distributing the weights of the network, at the cost of the communication of activations. Because of the sequential nature of most neural network models, it is necessary to pipeline the computations to use model parallelism efficiently. This incurs memory overheads for the activations, and significantly limits the scalability.

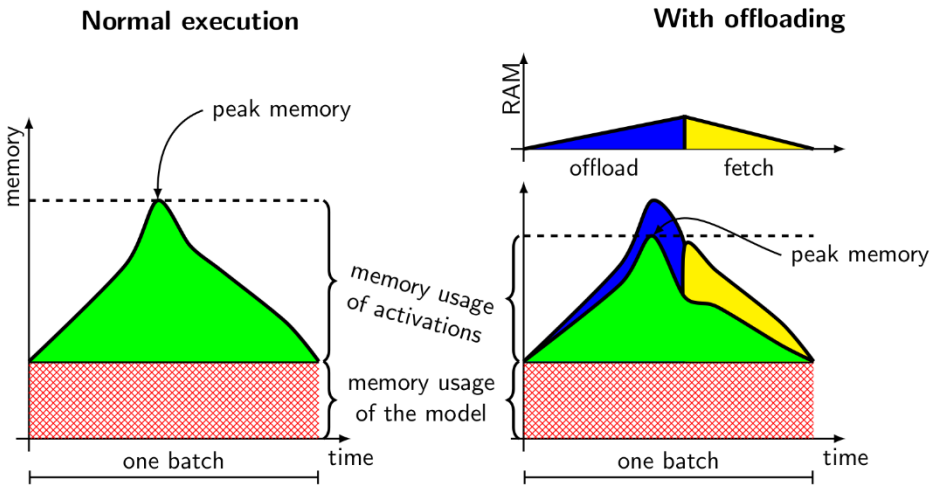


Figure 2.2: Reducing peak memory with activation offloading

The second category of solutions is purely sequential. Offloading [15] makes it possible to move some activations computed during the forward phase from the memory of the accelerator (GPU or TPU) to the memory of the CPU, and then to fetch them back at the appropriate moment into the memory of the GPU during the backward phase (see Figure 2.2). This solution therefore consumes bandwidth on the PCI-e bus between the CPU and the accelerator, which is also used to load training data. Another solution, called re-materialization [6], consists in deleting from accelerator memory some activations computed during the forward phase and then recomputing them during the backward phase. This approach does not consume communication resources, but it does induce a computational overhead (see Figure 2.3).

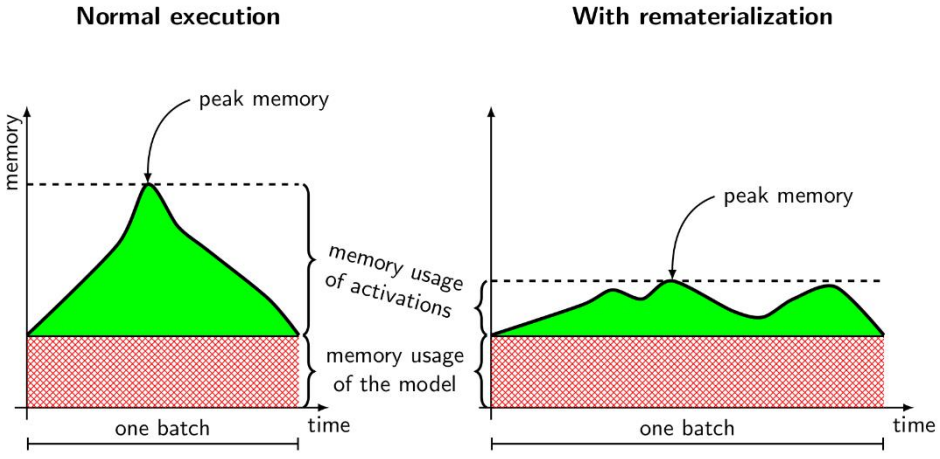


Figure 2.3: reducing peak memory with re-materialization

In this section, we present two contributions for improving the efficiency of training at the server level. The first one proposes an improved algorithm for re-materialization and obtains significant results on the well-known GPT networks. These results have been published in the ICML conference in 2023 [7]. The second contribution discusses the combination of model parallelism and re-materialization.

2.2. Re-materialization

A large part of the memory requirements during training come from the storage of the activations associated with gradient descent, since (almost) all the results computed during the forward phase must be kept in memory until they are used by the gradient computation during the backward phase.

In this part, we focus on the re-materialization approach on a single GPU or TPU. This is sufficient in practice for a large variety of neural networks, especially for performing fine-tuning of large networks with limited resources. Furthermore, re-materialization can be trivially combined with data parallelism to accelerate training if necessary. In this framework, for a given memory constraint, the optimization problem consists in finding a sequence of computing, forgetting and recomputing actions which allow one to perform the training for given inputs and batch sizes, while fulfilling the memory constraint and minimizing the computational overhead.

Previous solutions

To find the optimal sequence, different approaches have been proposed. In the first approach, like in Rotor [6], it is assumed that the dependencies within the model have a particular structure, typically a sequence of operations. In this case, using dynamic programming, it is possible to find the optimal order of computations in reasonable time. On the other hand, in the case where the computations performed by the model do not naturally consist in a sequence of operations, this approach requires to aggregate elementary operations into complex blocks to make the chain structure emerge. In this case, re-materialization decisions must be made at the level of blocks, which reduces optimization opportunities.

In the case of general graphs that are not structured as a sequence of elementary operations, another approach has been proposed in Checkmate [8]. It consists in describing the operations corresponding to both forward and backward phases as a Directed Acyclic Graph (DAG) and to find the optimal solution through solving an Integer Linear Program (ILP). The number of integer variables is proportional to $V \times E$, where V is the number of operations and E is the number of arcs of the DAG. Hence, a major shortcoming of this approach is the computational time induced by solving the ILP. Typically, even using commercial solvers such as CPLEX or Gurobi, it is not possible (in one day of computation) to consider a GPT2 models with more than 10 transformer blocks, while classical instances include several dozens.

Such GPT neural networks are not completely sequential, but can be decomposed in a sequence of blocks, where each block contains several operations. It is a typical example where using Rotor requires one to aggregate all the operations of the same block together. Rotor therefore decides at the scale of the whole block whether to keep all the data or to delete them all during the forward phase. Checkmate, on the other hand, sees the whole graph describing the model and can therefore decide, independently and at the level of each operation, whether to keep its data or not.

The Rockmate algorithm

To improve over both solutions, we have proposed a new re-materialization strategy called Rockmate, which combines the ideas of (i) Checkmate, which finds good solutions in the case of general graphs but is slow, and (ii) Rotor, which finds the optimal solution only in the case of sequential networks but is fast. In Rockmate, models are seen as a sequence of blocks (in the sense of Rotor), but where several optimal strategies are pre-computed for each block (using a Checkmate-like approach). The main idea is to apply Checkmate inside each block and to apply Rotor on the complete sequence of blocks.

As discussed above, Rotor fails to compute very good re-materialization strategies because it can only choose between two options: keep all or delete all activations in the block. In Rockmate, we use a refined version of Checkmate to generate a larger set of re-materialization strategies for each block. A re-materialization strategy is characterized by (i) the memory peak during the execution of the block (either during forward or backward) and (ii) the total size of the internal activations of the block that are kept between the forward phase and the backward phase. The first one ensures that this strategy can be executed within a given memory limit. The second one allows the dynamic program to know how much memory will be left for the next blocks. The number of different options to consider is a parameter of

Rockmate, and we have observed that using at most 400 different strategies in total for each block is enough to get good solutions in practice. Since we apply Checkmate at the level of a block (and not on the whole network), the corresponding graph is small enough that the runtime remains small, even for generating the whole family of strategies.

In summary, we have made the following contributions:

- We have developed a graph-building tool that automatically extracts the Directed Acyclic Graph (DAG) of the model, divides it into a sequence of blocks and identify all blocks with identical structures to avoid applying Checkmate multiple times on similar blocks.
- We have designed an improved Checkmate formulation that can express a limit over the size of activations which are kept in memory between the forward and backward phases of a block (and thus, during the execution of the following blocks).
- We have proposed an improved Rotor algorithm which instead of having two solutions per block, can exploit the different re-materialization strategies computed during the second phase. The output of this algorithm therefore consists in a schedule which describes which block should be computed, in which order, and with which re-materialization strategy.
- We have implemented all these algorithms into a Python framework, which can be used directly with a large variety of PyTorch models. This implementation is open-source and available at https://github.com/topal-team/rockmate

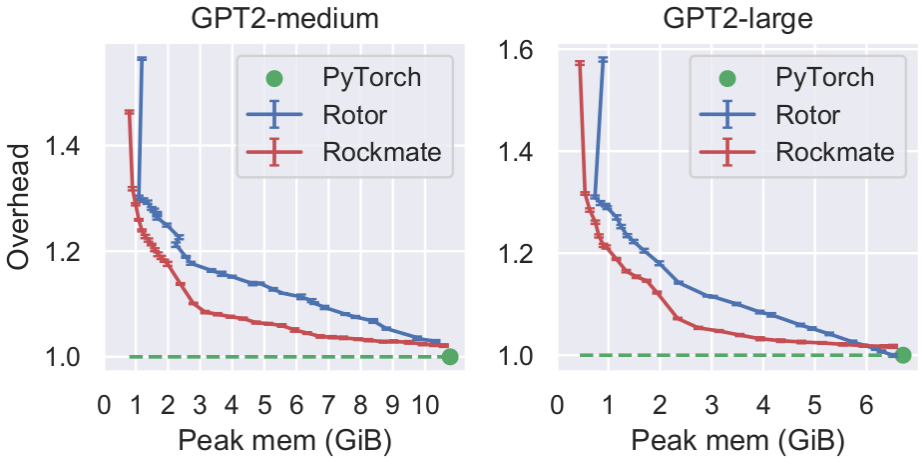


Figure 2.4: performance results of Rockmate on GPT2

An example of the performance results that can be achieved with this software is provided in Figure 2.4 for two variants of the GPT network (medium and large). This figure shows the computational overhead resulting from limiting the peak memory usage. The green dot denotes the performance achieved without any re-materialization. For any peak memory below this threshold, the red and blue lines provide the overhead in terms of computation time for the Rockmate and Rotor strategies respectively. We can see that Rockmate can reduce the memory usage by a factor of three, for an overhead limited to about 10%, twice smaller than the overhead of Rotor for the same memory budget.

2.3. Combining model parallelism and re-materialization

We now consider the (very common) case of a single server which contains several GPU devices. In such a context, it is natural to perform the training process in parallel on all the available GPUs of the server. The

most standard approach is called data parallelism, where the training samples are spread across all available GPUs so that each GPU processes a subset of the samples. When using data parallelism, the parameters of the model are replicated on all the GPUs to allow them to process the samples. To update the weights after each iteration, it is necessary to reduce all the gradients computed on all GPUs on a single resource, update the weights and then broadcast the resulting weights to all GPUs. For models with a very large number of parameters, data parallelism has two main drawbacks: memory usage is very high because of the replication of the parameters, and the associated communication costs become prohibitive and hinder the performance.

An alternative approach is model parallelism, where the model parameters are distributed across the GPUs instead of being replicated: each GPU is in charge of a different part (or stage) of the neural network. In that case, each sample is processed successively by all the GPUs, so that we replace the communication of the parameters by communications of the activations between the different stages. Parallel execution is achieved by pipelining the samples on the different GPUs (see Figure 2.5): the total batch size of an iteration is divided in several micro batches, whose number is denoted by m . The first micro-batch goes through the first stage of the neural network on the first GPU and the resulting activation is sent to the next GPU. While the second GPU processes this first micro-batch, the first GPU is available to process the next micro-batch, and so on. In the GPipe model parallelism strategy [9], once all micro-batches have been processed, the backward pass is performed in reverse order. The PyTorch framework includes an implementation of this GPipe model parallelism strategy [11].

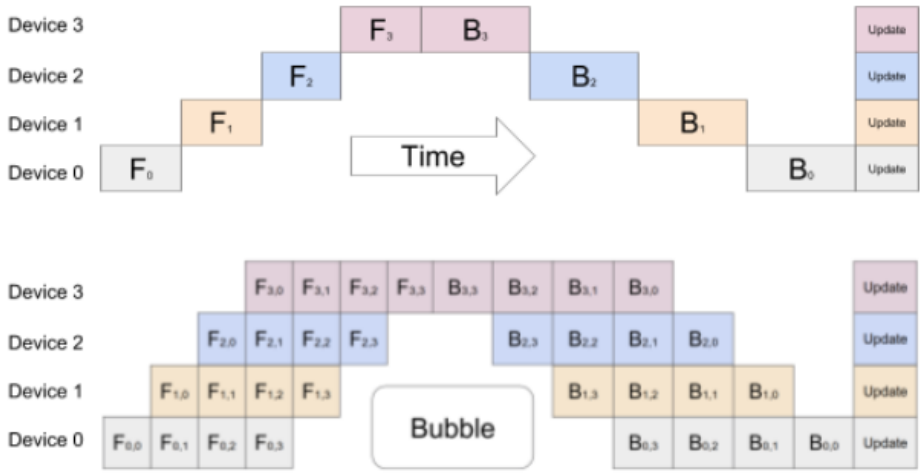


Figure 2.5: Model parallelism without pipeline (top), and with pipeline (bottom). Source: GPipe [9]

A deeper pipeline allows for better parallel efficiency because the pipeline remains full for a longer time. However, deepening the pipeline increases the memory used by the activations, which becomes a limiting factor for efficiency. It is thus natural to combine model parallelism with re-materialization: by storing fewer activations, one can reduce the memory pressure and in turn increase the total batch size and use an even deeper pipeline. The usual approach in most implementations is to only keep the first activation of each stage, forget all other activations, and recompute all of them during the backward pass. This very aggressive solution involves a large amount of recomputation which reduces, or even negates, the efficiency gains. This basic re-materialization approach is included in the GPipe implementation within the PyTorch framework.

In the following, we present our work on exploring the possibility of using more clever re-materialization approaches. For simplicity, we chose to use the Rotor algorithm for this exploration, and we selected the

GPT neural network examples, on a small number $N = 2$ or $N = 4$ GPUs. We first explored a direct approach where the re-materialization optimization is performed independently on each micro-batch, and then a more global approach where all the micro-batches are optimized together.

Direct approach: combining Rotor and Pipe

In the model parallelism setting, the neural network is divided in N successive stages of equal sizes. We consider the example of the GPT neural network, which is made of several dozens of transformer blocks, all with the same computational workload. This makes it quite easy to divide it into stages of roughly equivalent workload, by assigning a similar number of transformer blocks to each GPU. We divide the batch size into m micro-batches, where both the batch size and m can vary depending on the performance of the re-materialization strategy.

Our first approach is the direct combination of Rotor with the Pipe implementation included in the PyTorch framework. For a given stage, we decide to optimize the re-materialization of each micro-batch separately. Since all micro-batches of the same stage perform the same computations, it is not necessary to perform the optimization several times: we use the same optimized strategy for all micro-batches. If the total available memory for activations is M and the total batch size is B, we use the Rotor algorithm with an available memory $\frac{M}{m}$ and a batch size $\frac{B}{m}$.

The Rotor framework provides a new PyTorch module that implements this optimized strategy, and we can provide this new module to the Pipe framework to obtain a pipeline over all the micro-batches. This direct implementation ensures that the peak memory usage remains below M, discards a limited subset of the activations and optimizes the recomputation overhead of each micro-batch.

Global approach

The direct approach presented above has a drawback: to ensure that the total peak memory usage remains below M, it arbitrarily divides it equally between all micro-batches. However, this might be improved, in two different ways. First, there might be a more efficient way of distributing the available memory than such an equal repartition. Second, once the backward pass of one micro-batch has been computed, the corresponding activations are freed from memory. This additional memory capacity can be used to improve the re-materialization strategy of the next micro-batches and to further reduce the computational overhead.

The most natural way of addressing both of these concerns is to optimize the whole sequence of micro-batches with the Rotor algorithm. To do that, for each stage we create a sequence of micro-batches B_1 through B_m, separated by dummy operations that represent receiving the input of each micro-batch. We modify the Rotor algorithm to ensure that these dummy operations are not allowed to be recomputed, and we use this modified algorithm to optimize the whole sequence $B_1 \dots B_m$.

After the optimization, we divide the re-materialization strategy into subsequences, one for each micro-batch. We modify the Rotor runtime to ensure that it uses the correct subsequence at each micro-batch when interfaced with the Pipe framework. This results in a more optimized execution and further reduces the computational overhead.

2.4. Inference Efficiency

We now consider the inference phase of Convolutional Neural Networks. In other words, the model has been previously trained, so that the weights of the model are now constant throughout the inference

phase. The goal of that phase is to process successive inference *requests*, where each request consists of one or several input(s), which need to be classified according to the Neural Network.

We are interested in performing this phase at the server level, on a single node made of heterogeneous processing elements. These processing elements are usually a multi-core CPU on one side, and one or several GPU accelerator(s), or an FPGA component on the other side. To efficiently process requests on such an architecture, the goal is to optimize the usage of both kinds of resources so that they process parts of the networks adapted to their computational capabilities.

This optimization can be measured by two different criteria: one can strive to optimize the *throughput* of the system, defined as the number of requests processed per time unit; one can also be interested in minimizing the *latency* of requests, defined as the time between the arrival of a new request and the time at which the request is completed. Depending on the assumptions and on the choices made in the optimization decisions, these criteria are sometimes equivalent (optimizing one also optimizes the other), and sometimes antagonistic (optimizing one implies worse performance for the other).

The following sections present several directions of optimization that can be explored in such a context. We first discuss the possibility of partitioning the CNN across the heterogeneous hardware and the expected benefits. Then we present a task-based implementation of CNN inference that allows us to easily implement such partitioning solutions.

2.5. Partitioning the CNN on heterogeneous hardware

We consider a heterogeneous server, that typically contains one or several accelerator(s) and a multi-core CPU. This server must process requests for inference on a fixed CNN.

The basic approach, typically used in software like Triton [14], is to use each kind of processing element independently. In the most basic scheme, only the accelerators are used, with the idea that they are significantly more efficient at the convolutional operations required for the CNN. This approach relies on replicating the parameters of the model over all the processing elements (for example, in the main memory of the CPU, and in the memory of each GPU or each FPGA). This replication ensures that requests can be processed independently on any processing unit. Whether the CPU is used or not, in both cases requests are load-balanced among all the processing elements to ensure that all computing resources are used equally.

The main advantages of this approach are its simplicity: it is very easy to implement and does not require communication between the different computing elements. However, with the increase in the size of the models, the replication of the parameters of the model can induce a high memory pressure for large models. The most recent models will not even fit in the memory of only one computing element.

Another issue with the basic approach is that all processing elements perform all parts of the computational workload of the CNN, whereas with the heterogeneity of the platform, some elements might have different affinities with different parts of the CNN. For HPC applications, it has indeed been observed [12] that this can result in superlinear speedup if each processing element performs the computations for which it is more suited. For example, in a typical CNN, the tasks near the end of the computation are performed on significantly smaller data, for which the GPU is less suited because there is less parallelism.

For both reasons (decrease the memory usage and take advantage of the heterogeneity of the hardware), it is very relevant to partition the CNN across the different processing elements. In such a solution, each request would go through the processing elements in sequence: for example, the first GPU would process this request on the first part of the CNN, send the result to the second GPU which would process the second

part of the CNN, and the CPU would process the end of the request to obtain the final result. Several successive requests can be pipelined to ensure that processing elements do not remain idle.

In terms of memory usage, with such a solution, each processing element only needs to store the parameters of the model corresponding to the part of the network that it will compute. The parameters are thus *distributed* over the computing elements instead of being *replicated*. However, ensuring a correct load-balance between all computing elements is more challenging than in the fully replicated case. Careful optimization is needed to decide which part of the model is assigned to which computing element.

In the rest of the section, we present an example of a linear programming formulation that can help perform this decision.

2.6. Linear Programming Formulation

We start by providing some notation. For ease of presentation, we assume that the CNN consists in a sequence of tasks T_i for $1 \leq i \leq n$, such that the output of task $T_i$is the input of task T_{i+1}. The input of task T_1 is the input data of an inference request, and the output of task T_n is the result of this request. We denote by w_i the size of the model parameters needed to compute task T_i, and by o_i the size of the output of task T_i.

We consider a server with m heterogeneous processing elements and denote by $p_{i,j}$ the processing time of task T_i on the computing resource j. We denote by M_j the available memory on resource j, and by β_j the communication bandwidth out of resource j.

The goal of the optimization procedure is to assign tasks of the CNN to the computing resources to optimize the resulting throughput, while satisfying the memory constraint on each computing resource. We assume a steady-state execution of requests and express the execution of tasks as an average over an execution period.

Our formulation uses the following variables:

- for all $i \leq n$ and $j \leq m$, $x_{i,j}$ represents the (fractional) number of tasks of type T_i performed on resource j during one unit of time.
- for all $i \leq n$ and $j \leq m$, $s_{i,j}$ is equal to one if the parameters of task T_i are stored on resource j, and equal to zero otherwise.
- for all $i \leq n$ and $j \leq m$, $c_{i,j}$ represents the (fractional) number of data produced by task T_i and sent by resource j.
- ρ represents the (fractional) throughput of the solution, equal to the number of inference requests that can be processed during one time unit.

Constraints:

- The time spent computing on a given resource j during one unit of time is bounded by 1:

  $\forall j,\ \sum_{i=1}^{n} x_{i,j}\, p_{i,j} \leq 1$ (1)

- Computing a task T_i on resource j requires the parameters to be stored in memory:

$$\forall i,j,\ p_{i,j}x_{i,j} \leq s_{i,j} \quad (2)$$

This constraint ensures that if $s_{i,j}$ is 0, then $x_{i,j}$ is also 0: it is not possible to compute any task $s_{i,j}$ without the parameters of the model. If $s_{i,j}$ is 1, then this constraint becomes $p_{i,j}x_{i,j} \leq 1$. Since $x_{i,j}$ is already constrained by (1), this means that when $s_{i,j}$ is 1, $x_{i,j}$ can take any feasible value and is not limited by constraint (2).

- The total memory size of model parameters stored on resource j should fit within the available memory:

$$\forall j,\ \sum_{i=1}^{n} s_{i,j}\, w_i \leq M_j \quad (3)$$

- For a given task T_i and resource j, the amount of data sent out of resource j for that task can be evaluated based on the number of tasks T_i and T_{i+1} computed by resource j. Indeed, if resource j computes more tasks of type T_i than tasks of type T_{i+1}, the results produced in excess on resource j have to be sent out of resource j. This gives the following constraints:

$$\forall i,j,\ c_{i,j} \geq x_{i,j}o_i - x_{i+1,j}o_i \quad (4)$$

$$\forall i,j,\ c_{i,j} \geq 0 \quad (5)$$

- Since the link bandwidth is β_j, the total data that can be sent out of resource j during one time unit is bounded by β_j:

$$\forall j,\ \sum_{i=1}^{n} c_{i,j} \leq \beta_j \quad (6)$$

- For any task T_i, the throughput cannot be larger than the total number of tasks of type T_i processed across all resources:

$$\forall i,\ \sum_{j=1}^{m} x_{i,j} \geq \rho \quad (7)$$

We obtain the following linear programming formulation:

maximize ρ subject to

Constraints (1-7)

$x_{i,j} \geq 0, s_{i,j} \in \{0,1\}$

The discussion above shows that solving this formulation provides a way to assign the different layers of the model to the computing resources that allows for an optimal throughput. This assignment considers both the memory constraints on each resource (with constraint (3)) and the heterogeneity of the computing efficiency thanks to the $p_{i,j}$ values. The assumption of a steady state periodic schedule may not be realistic in practice, but it allows us to obtain a clear and efficient formulation. For a practical implementation, it seems more relevant to use dynamic scheduling approaches to make decisions at runtime for each individual request: for a given task T_i, we can choose the least loaded computing resource among all those that have the necessary model parameters to compute this task.

In the next section we describe how to build a task-based implementation for CNN inference that can perform such dynamic scheduling decisions.

2.7. Task-based implementation

This section presents an ongoing work for a task-based implementation of inference for Convolutional Neural Networks. The goal is to be able to efficiently perform inference computations on a heterogeneous server, with the possibility to finely control which computing resources perform which part of the computation. This implementation is based on the StarPU runtime which manages task scheduling and communications, and on the ONNX framework which provides efficient implementations of the neural network operators. Our implementation provides a link between both frameworks.

Since this implementation is still in development, this deliverable does not contain experimental results. A more detailed performance analysis will be provided later in another deliverable.

StarPU library

StarPU [12] is a high-performance task programming library for hybrid architectures. It is a software tool aiming to allow programmers to exploit the computing power of the available CPUs and accelerators, while relieving them from the need to specially adapt their programs to the target machine and processing units.

At the core of StarPU is its runtime support library, which is responsible for scheduling application-provided tasks on heterogeneous machines. In addition, StarPU comes with programming language support, in the form of an OpenCL front-end.

StarPU's runtime and programming language extensions support a task-based programming model. Applications submit computational tasks, with CPU and/or GPU implementations, and StarPU schedules these tasks and associated data transfers on available CPUs and GPUs. The data that a task manipulates are automatically transferred among accelerators and the main memory, so that programmers are freed from the scheduling issues and technical details associated with these transfers.

StarPU takes particular care of scheduling tasks efficiently, using well-known algorithms from the literature. In addition, it allows scheduling experts, such as compiler or computational library developers, to implement custom scheduling policies in a portable fashion.

The main concepts of StarPU are:

- A codelet describes a computational kernel that can possibly be implemented on multiple architectures such as a CPU, a CUDA device or an OpenCL device.
- A data handle is a StarPU structure that describes a piece of data that can be accessed by the different computational tasks.
- A computational task: executing a StarPU task consists in applying a codelet on a data set, on one of the architectures on which the codelet is implemented. A task thus describes the codelet that it uses, but also which data are accessed, and how they are accessed during the computation (read and/or write).
- A worker is any computing resource that can process tasks. Each worker is associated with a memory node; processing a task on a worker might involve transferring the necessary data to the corresponding memory node if it is not already present. These transfers are performed automatically by StarPU with no user involvement.

ONNX framework

ONNX Runtime [13] is a cross-platform machine-learning model accelerator, with a flexible interface to integrate hardware-specific libraries. ONNX Runtime works with different hardware acceleration libraries through its extensible Execution Providers framework to optimally execute the ONNX models on the

hardware platform. These execution providers provide efficient implementation on different accelerators of most of the basic computing kernels used in neural networks.

Our current implementation uses the default CPU and CUDA providers and is thus able to use both kinds of devices. The genericity of both StarPU and ONNX make it very extensible, so that supporting other devices would only require minimal changes to the code.

Software architecture

We start by providing a high-level view of the process of our task-based implementation. It can start from any neural network model trained with PyTorch. We use ONNX utility functions to obtain the computational graph of the neural network. We divide this graph into subgraphs, so that the workload of each subgraph is large enough to represent a StarPU task. These subgraphs contain several nodes of the computational graph of the neural network but will be seen in our implementation as a single StarPU task. These subgraphs are exported with the ONNX export utility function, each in a separate export file. This first part of the process takes place in the user's PyTorch application.

The ONNX export files are opened by our task-based implementation. There is a StarPU data handle for each input or output tensor of these export files, identified by their name in the original computational graph. These data handles are registered at initialization time and will allow StarPU to identify the data dependencies between the different subgraphs. An ONNX session is initialized for each subgraph on each computing resource that should run this subgraph; this loads the model parameters into the memory of the device.

When performing an inference for a given input data, the corresponding StarPU data handle for the input tensor is filled with the data, and all StarPU tasks corresponding to all subgraphs are submitted to the StarPU runtime. StarPU identifies which task can run on which computing resource, automatically schedules tasks on the most efficient and/or least loaded resource, and automatically manages communication between all the devices. Once all tasks have been processed, the output tensor contains the result, which can be transferred to the user.

Specific implementation details

- Our solution makes use of the concept of combined workers in the StarPU runtime system: several CPU cores can be merged to work as a single StarPU worker. It is thus possible to use the multi-threaded execution providers of ONNX to perform multi-core executions of tasks on the CPU. This requires us to carefully pin the threads on the correct CPU cores, to avoid interference between the different ONNX sessions.
- For the input and output tensors of the complete neural network, the corresponding StarPU data handles use statically pinned memory buffers. This ensures that the transfers to and from the computing resources (the GPUs for example) can be performed as fast as possible, and in a completely asynchronous fashion.
- Similarly, for the other tensors which carry data dependencies between the subgraphs, the corresponding StarPU data handles are registered with `memory_node=-1`. This means that StarPU can automatically allocate memory for the result of a task on the computing resource that processes this task. The cache management system of StarPU ensures that buffers are not freed after use, but reused for other similar data, so that the overhead cost of memory allocation is negligible after the first few inferences.
- All our implementation is completely asynchronous and uses both the asynchronous API of ONNX and StarPU. This allows StarPU to monitor device status and communication progress while the

computation is performed on the CPU and/or GPU. We can thus achieve a very efficient overlap between computations and communications, avoiding unnecessary synchronizations.

3. Efficient CNN kernels for FPGA

3.1. Introduction

This section explains how the OmpSs@FPGA programming model has been used in order to improve the execution of CNN kernels in FPGA devices. From the technical and research points of view, the following contributions have been carried out:

- A 2d convolution kernel has been ported to FPGA devices using the OmpSs task-based programming model
- A study on how to improve performance of a 2d convolution using OmpSs@FPGA features has been done
- The 2d convolution kernel has been transformed following the findings in the previous point to increase its performance until it is competitive with CPU state-of-the-art implementations (like Pytorch)
- The framework has been evolved in order to support mixed precision data types usually present in CNN implementations
- An improvement of the framework to support newer C++ standards (as C++ 11 needed by RAIDER) has been reached.
- An initial porting of the RAIDER inference application to the OmpSs task-based programming model has been achieved

The developments explained in this deliverable section are also related to the ones reported in deliverables 2.10 IP for fast task scheduling, part 1; 2.11 IP for fast task scheduling, part 2; 4.6 Task-based runtime systems and 4.7 HLS Flow. In particular, the IP for Fast Task Scheduling developed in task 2.5 and the OmpSs@FPGA task-based model developed in task 4.2 have been used here/have been developed with inputs from the work reported in this deliverable.

3.2. Relationship with the project objectives and strategic goals

This deliverable is related to the following project objectives as stated in the DoA:

- Objective 1 - Energy efficiency. Executing in FPGA has been demonstrated to be competitive with other computing platforms in terms of energy efficiency. At the same time, DNN executions are power hungry and one of the main sources of computing power usage nowadays. To be able to easily port DNN applications to FPGAs using OmpSs@FPGA programming model will help to improve inference and training energy efficiency.
- Objective 2 - Sustained application performance. As explained in the next sections, we aim to improve the performance obtained when executing applications over the IDV-E platform both by improving the framework and also by improving the task scheduling through the use of the Fast Task Scheduler developed in Task 2.5. In this deliverable it is shown how CNN kernels performance can be improved in FPGAs.
- Objective 4 - Seamless integration of reconfigurable accelerators. One of the main concepts behind OmpSs@FPGA is to allow the seamless integration of reconfigurable accelerators. In this deliverable we study how to integrate accelerators for CNNs.

- Objective 5 - Development of new IPs. Although the IPs developed here are designed in HLS the CNN kernel developed by BSC will be fully available to the public and the source code is even listed in this deliverable.

The work developed here is also related to the strategic goals of the project as follows:

- Strategic Goal #2: Supporting the objectives of EuroHPC as reported in ETP4HPC's Strategic Research Agenda (SRA) for open HW and SW architecture. The OmpSs@FPGA SW framework is open source and publicly available as all the developments done in this project.
- Strategic Goal #3: Opening of new usage domains. The work developed in this section is strongly aligned with this strategic goal as although FPGAs are already used to perform both CNN training and inference the usual approach doesn't involve the use of a task-based programming model to do so. The advantages of task-based programming models in general and OmpSs@FPGA in particular (mainly programmability, portability and performance) would pave the way to new developments in the DNN world if the research is successful.

3.3. 2d convolution

Convolution is often the most computationally expensive step in convolutional neural networks. Meanwhile other steps such as relu or pooling have a non-trivial cost, usually the bottleneck of those neural networks is the convolution. The computational cost of a convolution is determined by the product of the sizes of the different inputs and convolutional kernels. For a given dataset, composed of NI input channels of size SI , NO output channels and a kernel of size KS , the cost of a typical convolution would be proportional to $IS \cdot NI \cdot NO \cdot KS$. Meanwhile a typical ReLu step would have a cost of $IS \cdot NI$ and the pooling step will also have a cost proportional to $IS \cdot NI$.

For any non-trivial input data set, convolution will be the most computationally expensive kernel of all three. Therefore, accelerating this kernel will provide greater performance gains than ReLu or Pooling.

3.4. Kernel description

Discrete convolution, 2-dimensional in our case, is implemented by applying a N_o convolutional kernel over all elements of all channels of a 2d matrix. N_o is the number of output channels in this layer of the neural network.

Input matrix size, kernel size as well as number if input and output channels are defined by the model and can be different for each model and for each layer of a given network.

The convolutional kernel in our case is defined by the code in listing 3.1

```
for im in batch:
  for oc in output channels:
    for h in im.height - kernel[oc].height + 1:
      for w in im.width - kernel[oc].width + 1:
        result[im][oc][h][w] = bias[oc]
        for ic in b.input_channels:
          for y in kernel[ic][oc].height:
            for y in kernel[ic][oc].width:
              result[im][oc][h][w] += im[ic][h+y][w+x]*kernel[ic][oc][y][x]
```

Listing 3.1: 2d convolution algorithm

Data is laid out in memory as a contiguous array of 2d row-major matrices. Figures 3.1, 3.2 and 3.3 illustrate the layout of the convolution kernel, input data and output data respectively.

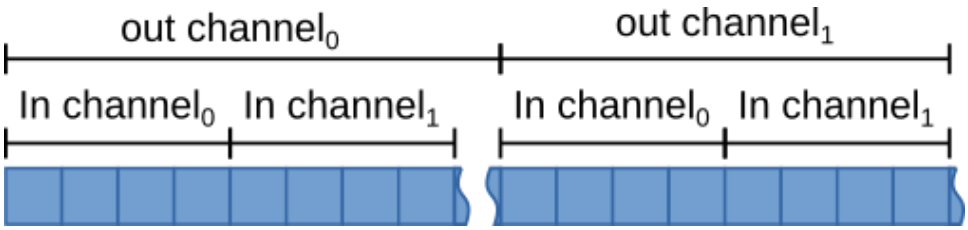


Figure 3.1: Kernel memory layout

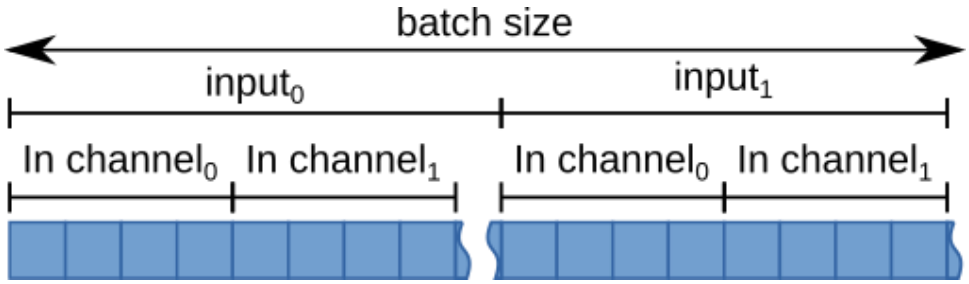


Figure 3.2: Input data memory layout

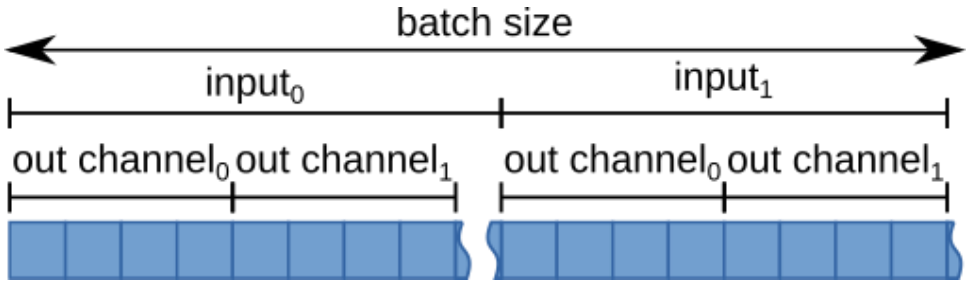


Figure 3.3: Output data memory layout

This data arrangement is assumed throughout the rest of the application and therefore it cannot be easily changed. Any layout transformations for the convolutional step of the neural network layer have to be carefully evaluated as any performance gains resulting from better exploiting data locality can be easily negated by the cost of the transformation itself.

Given this data arrangement in Figures 3.1, 3.2, 3.3 and the algorithm in listing 3.1, some of the data sets have to be visited multiple times. In the case of the algorithm in listing 3.1, input and kernels are visited multiple times for each output. However, output data is visited only once. This scheme is known as *output stationary*. Even though the loops of the algorithm can be rearranged to an input stationary (visit input data only once) or weight stationary (visit a kernel only once) form, whether or not any of the schemes provides any performance advantages will depend on the particular data set of a given model. Therefore, a general solution that provides optimal performance is not possible.

3.5. FPGA implementation

The 2d convolution kernel is implemented in FPGA using the OmpSs@FPGA framework. It provides a productive toolchain that allows quick iteration. Also, by having HLS code tightly integrated in the application, the framework allows quick C-based co-simulations in order to validate correctness of the accelerator kernel before implementing the design which it is a time-consuming operation which takes several hours per iteration.

Kernel taskification

This implementation step requires identifying application computation kernels, splitting them into functions, labelling them with pragma directives and defining task data dependencies.

In this case, the kernel code size is small, and it is already implemented in a function. Besides labelling the task to be implemented in an FPGA, data dependencies need to be labeled. This is shown in listing 3.2.

```
#pragma oss task device(fpga) \
    in( [bs*in_channels*in_H*in_W]input, \
        [out_channels*in_channels*k_H*k_W]kernel, \
        [out_channels]bias) \
    out([bs*out_channels*(in_H - k_H + 1)*(in_W - k_W + 1)]output)
void conv2D_FPGA(int bs,
        float* input,
        const int in_channels, const int in_H, const int in_W,
        float* kernel, const int k_H, const int k_W,
        float* output, const int out_channels,
        float* bias) { ... }
```

Listing 3.2: Convolution kernel OmpSs@FPGA directives and function header

Listing 3.2 shows the directive that defines the conv2D_FPGA function as a task. Then `in()` and `out()` clauses define task input and output data dependencies respectively.

Dependencies are specified as shaping expressions. The syntax for data shaping expression in this example is `[data_size]pointer`. For instance `[out_channels]bias` defines the dependency to be all data from `bias[0]` to `bias[out_channels]`. In this case, data shapes depend on the parameters, therefore, they cannot be determined at compile time.

Kernel specialization

In order to allow an efficient FPGA implementation, the kernel has to be specified for the current use case. This step involves removing infrequent code paths and treating those cases in host code. Usual cases are prologs or epilogs that skew data alignment and cause low usage of FPGA resources or treatment of corner cases. This can be done in the CPU as they usually represent a very low percentage of the total problem size. Also, converting parameters to constants known at compile time when possible, allows better overall implementation as it enables some optimizations that otherwise cannot be performed.

In our case, kernel does not have infrequent code paths. However, most of the parameters can be converted to constants. For a given convolution layer of a given model, input size, kernel size, input channels, output channels, and output size are fixed. Since we target this use case making these values constants known at compile time is a valid solution.

Having constant data size allows the tools to allocate local kernel memory, usually implemented as BRAMs blocks. This allows much faster memory access when compared to accessing off-chip DDR memory. Also,

this enables partitioning of this local storage, which will allow multiple parallel accesses in later optimization stages.

Listing 3.3 shows data storage created by the OmpSs@FPGA compiler in the kernel HLS wrapper, this is only possible when size is known at compile time since hardware resources have to be allocated and properly connected.

```
void conv2D_FPGA_wrapper(hls::stream<ap_uint<64> >& mcxx_inPort,
hls::stream<mcxx_outaxis>& mcxx_outPort, ap_uint<256>* mcxx_memport) {
  static float kernel[2304];
  static float output[61504];
  static float input[65536];
  static float bias[32];
…

}
```

Listing 3.3: Local storage allocated by kernel wrapper.

Also, minor improvements are achieved as loop control is simplified since all loop bounds are known at compile time.

Batch parallelization

Convolution Kernel has been parallelized by splitting the batch loop, which is the outermost loop shown in listing 3.1. Since loop iterations are independent in this loop, it can be easily parallelized by instantiating multiple accelerators implementing the same task without having to duplicate data or needing synchronization between kernels. Distributing iterations of the `output_channels` loop would require replicating input data as each output channel needs data from all input channels. Splitting `input_channels` loop, would require synchronization between accelerators in order to perform accumulations into a given output channel.

Batch level parallelization is achieved by processing only one of the batch elements in each kernel and calling them for each element in the batch. Moreover, all kernel calls in a batch can be performed from an FPGA task. Listing 3.4 shows the code using nested FPGA tasks.

```
#pragma oss task device(fpga) \
    in([batch_size*IN_CHANNELS*IN_H*IN_W]input, \
            [OUT_CHANNELS*IN_CHANNELS*K_H*K_W]kernel, \
            [OUT_CHANNELS]bias) \
    out([batch_size*OUT_CHANNELS*(IN_H - K_H + 1)*(IN_W - K_W + 1)]output)
void conv2D_batch_FPGA(
        float* input,  float* kernel, float* output, float* bias, int
batch_size) {
    const int out_H = IN_H - K_H + 1;
    const int out_W = IN_W - K_W + 1;
    for (int b = 0; b < batch_size; b++) {
        conv2D_FPGA(input + b*IN_CHANNELS*IN_H*IN_W,
                kernel,
                output + b*OUT_CHANNELS*out_H*out_W,
```

```
                bias);
    }
    #pragma oss taskwait
}

#pragma oss task device(fpga) \
   in([IN_CHANNELS*IN_H*IN_W]input, \
    [OUT_CHANNELS*IN_CHANNELS*K_H*K_W]kernel, \
    [OUT_CHANNELS+16]bias) \
    out([OUT_CHANNELS*(IN_H - K_H + 1)*(IN_W - K_W + 1)]output) \
    copy_deps num_instances(FPGA_CONV_INSTANCES)
void conv2D_FPGA(
        float* input,
        float* kernel,
        float* output,
        float* bias) { … }
```

Listing 3.4: FPGA nested tasks implementing batch processing of different inputs

Note that `conv2D_batch` reads the full input data and writes the full output data, but `conv2D_FPGA` needs only a subset of it.

By creating nested tasks in the FPGA, we take advantage of the high throughput of Fast Task Scheduler (FTS) which allows faster task creation. Furthermore, we reduce synchronization points between FPGA and the CPU. When using nesting the host system only needs to synchronize with the top-level task (`conv2D_batch`) instead of synchronizing for every batch element.

By using the `num_instances()` clause, we can control how many accelerators instances are going to be instantiated in the final design. On runtime, tasks will be dynamically distributed among different instances based on accelerator availability.

Kernel pipelining

In order to improve the convolution, the filter has been pipelined in 2 different dimensions as shown in figure 3.4. On one side the filter is memorized and only the front wave is read for each filter application in the w direction. In addition, the filter is unrolled in the h dimension so increasing the front wave by only one element allows the algorithm to compute two positions of the same filter each iteration.

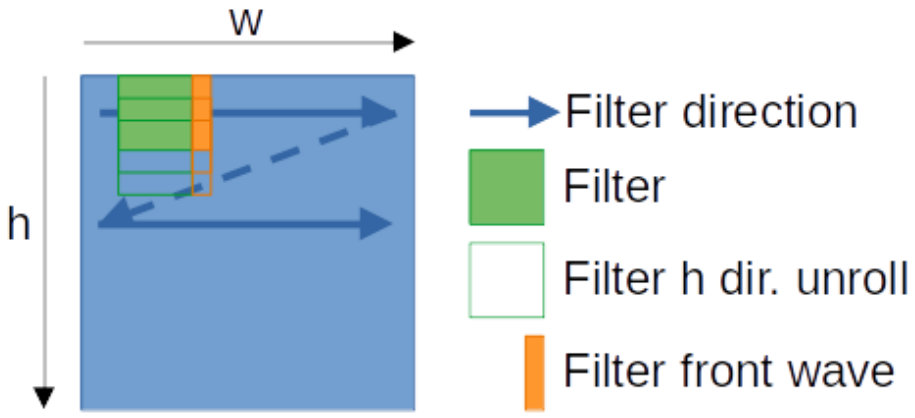


Figure 3.4: Filter pipeline in 2 dimensions

The HLS code that implements the front wave reader shown in orange in figure x.4 can be seen in listing 3.5. As it can be seen only the elements at w+2 are read. The remaining `input` elements are kept in the temporal input buffers (`tinputXY` variables). The X denotes the row (h value) while the Y denotes the column position (added to the w value). Each cycle in the w direction loop a new value is read and the previous values are cycled to apply the filter to the sliding window of `input` elements.

```
tinput00=tinput01;
tinput01=tinput02;
tinput02=input[b*IN_CHANNELS*IN_W*IN_H+ channel*IN_W*IN_H+h*IN_W+(w+2)];
tinput10=tinput11;
tinput11=tinput12;
tinput12=input[b*IN_CHANNELS*IN_W*IN_H+ channel*IN_W*IN_H+(h+1)*IN_W+(w+2)];
tinput20=tinput21;
tinput21=tinput22;
tinput32=input[b*IN_CHANNELS*IN_W*IN_H+ channel*IN_W*IN_H+(h+2)*IN_W+(w+2)];
```

Listing 3.5: Front wave reading and filter sliding code

By applying this solution, the number of data read is reduced as only *kernel height* input elements are read for each output element, instead of the full filter window. Even though this is a big improvement over the naïve implementation, it has a series of issues regarding data access.

Accelerator data, as shown in listing 3.3, is stored in BRAM blocks. Each one has two ports. Therefore, only two elements in each block can be accessed on a given cycle. This is a clear issue if we try to access the full filter front wave in a single cycle. Figure 3.5 shows this effect. Red squares represent conflicting accesses that try to access data in the same BRAM block, represented as a blue square in Figure 3.5.

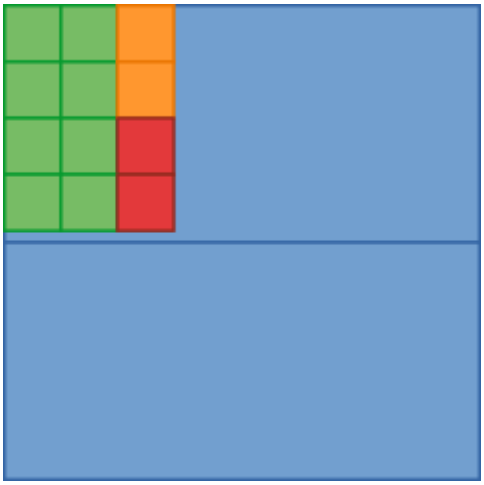


Figure 3.5: Input accesses using default data layout.

By partitioning input storage, we can distribute data among different BRAM blocks to allow 2 accesses to be performed in parallel in each cycle. However, it still would take two cycles to access the new data needed to process one output element. This arrangement is shown in Figure 3.6. Red squares represent conflicting access that will have to wait. Blue stripes represent a BRAM block, after partitioning is applied.

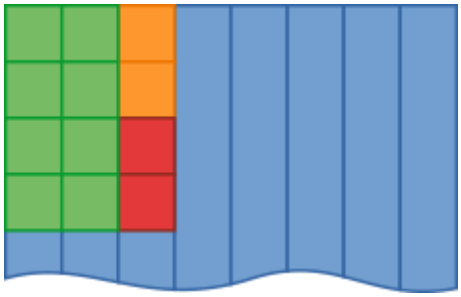


Figure 3.6: Conflicting accesses in the same memory partition (vertical stripes)

To work around these issues, accesses are offset so that they use a BRAM block from a different partition. This is illustrated in figure 3.7.

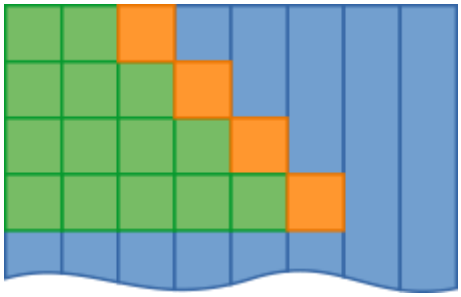


Figure 3.7: Offset input accesses over different partitions.

By doing this, all accesses can be done in parallel. All data needed to apply the filter to two positions can be read in the same cycle.

3.6. Results

Evaluation of the FPGA implementation has been carried out on a Xilinx Alveo U200 accelerator card. The host system is an Intel Xeon Silver 4208 CPU with 64GB of main memory. Figure 3.8 shows performance results.

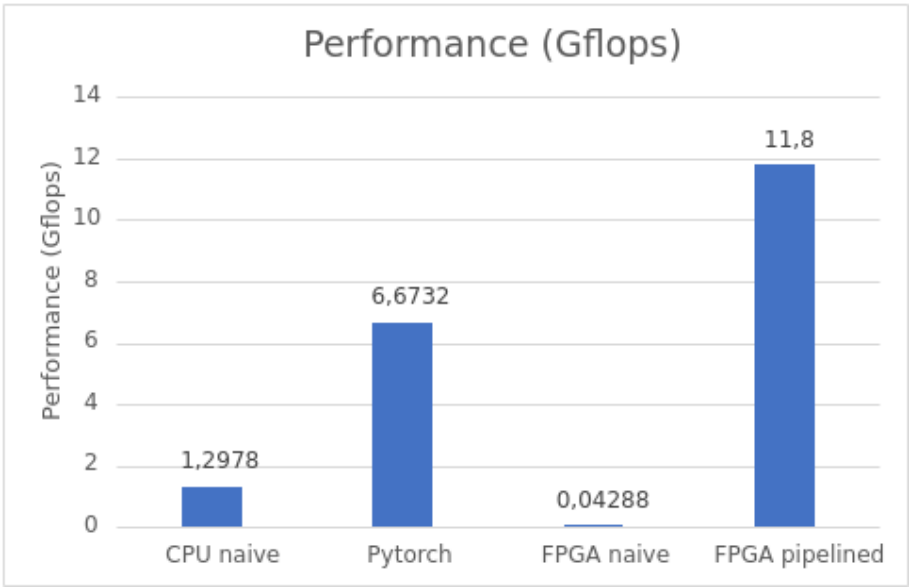


Figure 3.8: 2D convolution performance

Chart shows the performance in Gflops for a naïve CPU implementation, the PyTorch CPU implementation, a naïve FPGA implementation and a pipelined implementation. CPU naïve is a straight implementation of pseudocode shown in listing 3.1. Pytorch implementation is a more CPU optimized implementation that uses all available cores in our test system. FPGA naïve, in the same fashion as the CPU naïve, is a straight implementation of algorithm described in listing 3.1, using OmpSs@FPGA in order to offload task execution to the FPGA, using a single accelerator running at 200 MHz. *FPGA pipelined* corresponds to the improved implementation using pipeline and partitioning techniques described in previous sections. In this case, 10 accelerators instances running at 300 MHz are used.

FPGA naïve implementation is very slow. This is because if no optimization is applied, a single operation is performed each 9 cycles, which running at 200 MHz does not provide a reasonable performance.

	Naïve		Pipelined		Available
	Number	%	Number	%	Alveo U200
BRAM	642	14.86	3378	78.19	4320
DSP	24	0.35	462	6.75	6840
FF	182392	7.71	425818	18.01	2364480
LUT	157785	13.35	425818	28.35	1182240
URAM	7	0.73	17	1.77	960

Table 3.1; 2D convolution FPGA implementation resource usage

Table 3.1: Shows resource usage of the different implementations of the 2D convolution kernel in FPGA. As it can be seen the Pipelined implementation obtains improved performance at the cost of using a high percentage of the board's available BRAMs. Further optimizations that involve increasing the data reuse to raise even further the operations per cycle (and thus the DSPs involved in the computations) are being developed.

3.7. Compiler modifications for CNN mixed precision

As explained in deliverables 4.6 Task-based runtime models and 4.7 HLS flow, as part of the project we have undertaken an update of part of the OmpSs@FPGA framework. One of the biggest challenges has been to discard BSC in-house Mercurium compiler in favor of a llvm fork [17][18] that applies the necessary compiler transformations to the C/C++ source code before feeding it to the vendor tools (Vitis HLS in the case of IDV-E). This change allows the framework to work with more modern codes, such as the ones usually found in CNNs. In this section we describe the necessary llvm modifications that allow us to use Vitis HLS specific precision types, very useful for CNNs. All the code is distributed as open source [17]. It is important to highlight that the fact of using an llvm fork does not interfere with the vendor tools that also use an llvm fork to compile from HLS. Our part of the compiler, (detailed in deliverable 4.7 HLS flow) reads the C code targeting the FPGA and emits, again C source code avoiding any incompatibilities with other environments in the toolchain.

Support for non-built-in data types and C++ constructions has been implemented in the clang frontend action that implements FPGA transformations. This is needed to support arbitrary precision data types such as fixed precision numbers or reduced floating point (FP16). These data types are widely used in CNN

applications. Some parts of such applications do not need the precision provided by standard data types, therefore allowing developers to save resources by using reduced precision data types in certain parts of the application.

Compiler changes are closely related to support for user-defined types (i.e. classes) and constructions added in newer language versions.

3.8. Support for custom data types in tasks

Custom data types were not supported in earlier versions of the compiler toolchain, we added support for using them as copies, dependencies or internal (not related to kernel interfaces) computations.

Dependency resolution of C++ types

In order to emit valid HLS code, kernel closure must be computed. The compiler must identify which definitions are needed by a kernel in order to output them along with the computation kernel source code.

In C, this is straight forward. However, in C++ the compiler ran into declaration conflicts due to template instantiation. This is needed in order to support vitis hls fixed point types as they are implemented as a templated structure.

Listing X.6 shows how the compiler has instantiated the template due to the declaration of x. For each different template instantiation, the compiler creates a specialization. Sample code is correct as multiple definitions of myType are not conflicting. In this case, the compiler should choose the most specialized definition (bottom one) over the more general ones. However, even though listing 3.6 shows two declarations of myType, the input source code contained only the first one.

```
template <typename T, int L, int M> struct myType {
    T                                          a[L];
    T                                          b[M];
};
template    <>    struct    myType<int,    16,    5>    {
    int                                        a[16];
    int                                        b[5];
};

…

myType<int, 16, 5> x;
```

Listing 3.6: HLS code containing source and instantiated templates.

Internally, this is handled by the compiler by creating an AST tree representing each instantiation. However, comparing both trees for structural equivalence to check for violations of the One definition Rule (ODR) are not trivial and false positives were found when dealing with templates in the FPGA frontend action.

To overcome this issue, a more complex type of AST import needs to be used when duplicating fragments of the program AST in the FPGA phase. This import method takes template parameters into account when computing structural equivalence to identify type declarations that should not conflict with each other.

Implicit Array initialization of non-standard types

Support for declaring arrays of user-defined types without explicit initialization had to be implemented. Support for this use case is needed since fixed point and half precision floating point are implemented as user-defined types. Otherwise, the pretty printer module in clang assumes that all arrays containing C++ user-defined data types will have an explicit initializer. The pretty printer is the compiler module that outputs C++ code from an AST. Explicit initialization is not always the case. A user can define an array of objects and initialize them later. This is illustrated in listing 3.7, which is now supported.

```
ap_fixed<16,5> layer2_out[LAYER2_SIZE];
```

Listing 3.7: Declaration of an uninitialized array of ap_fixed<16, 5> elements

Support for custom type serialization

Serialization and deserialization of non-trivial data types was added to the FPGA clang frontend action. These operations are implemented using a union to build any type from raw bits that are read from an interface as shown in listing 3.8. These operations are implemented in the task wrapper.

```
template<class T>
union __mcxx_cast {
  unsigned long long int raw;
  T typed;
};
static float bias[32];
…
__mcxx_cast<float> cast_tmp;
cast_tmp.raw =  mxcc_data[I](32,0) //port[address](bit_range)
bias[I] = tmp.typed;
```

Listing 3.8: Deserialization of a floating-point number

However, this is not valid when the destination type is not a trivial type, which is the case of fixed point and half-precision. In such cases a union constructor has to be emitted into the HLS code. This is shown in listing 3.9.

```
template<class T>
union __mcxx_cast {
  unsigned long long int raw;
  T typed;
  __mcxx_cast() {}
};

static half c[4096];
…
__mcxx_cast<half> cast_tmp;
cast_tmp.raw = mcxx_data[addr](16,0);
c[I] = cast_tmp.typed;
```

Listing 3.9: Deserialization of a half precision floating point number,

Note that the constructor is in fact empty as type will not be initialized on union constructor but built using raw bits. Furthermore, this is a C++11 construct. Therefore, changes to the toolchain, specifically AIT, are needed in order to enable the proper C++ standard version in HLS.

Support for operator calls in FPGA tasks

Arithmetic operations such as addition or subtraction of fixed-point or half precision floating point types are implemented as C++ operators, therefore, support for calling overloaded operators inside FPGA tasks has been added. Treating them as regular function calls inside the compiler Is not enough. They are special for the compiler and have to be processed accordingly. We have introduced the proper processing code into the OmpSs@FPGA llvm fork.

3.9. Support for C++ constructions

Support for certain C++ constructions have been implemented to support output code generated using ML4HLS [19], [20]. This tool generates an HLS implementation of a convolutional neural network from a PyTorch model. Since the code is automatically generated, it may contain code that is not common for programmers to write, using some features that lacked compiler support. This tool is used in the RAIDER application. It uses this tool in order to implement an FPGA accelerated CNN.

Support for templates and namespaces in function calls

Support for calling functions that are defined inside a namespace have been implemented into the FPGA frontend action. An example of this use case is shown in listing 3.10

```
nnet::conv_2d_cl<input_t, layer2_t, config2>(input1, layer2_out, w2, b2);
```

Listing 3.10: Template call inside a namespace

As opposed to C, where the namespace is global, information provided by the caller has to be preserved. This information includes the fully qualified name, which is the function name including the namespace where it is defined, and the list of template parameters.

Support for recursive constexpr

Recursion in FPGA kernels is not supported in general. However, it is allowed inside a constexpr qualified function. By qualifying a function as constexpr, it guarantees that it can be evaluated at compile time and will be a constant at runtime. Therefore, this special case needs to be treated since it is valid code for FPGA kernels. An example from raider is shown in listing 3.11. When processing FPGA kernel AST, the compiler has to keep track of which functions it has already visited when processing function calls in order to detect such cases.

```
constexpr int ceillog2(int x) {return (x<=2) ? 1 : 1 + ceillog2((x + 1)/2);}
```

Listing 3.11: Recursion inside a constexpr from the RAIDER application

4. Workflow for the Deployment of CNNs on FPGA in the RAIDER Application

RAIDER is a high throughput online streaming processing application implemented on FPGA with the APEIRON framework. Its task is to perform particle identification (PID) on the stream of events generated by the RICH (Ring Imaging CHerenkov) detector in the CERN NA62 experiment at a rate of about 10 MHz, using neural networks. The inference task consists in providing an estimate for the number of charged particles (0, 1, 2, >=3) for any event detected on the RICH detector, that corresponds to the number of ring tracks that can be reconstructed from the pattern of photomultipliers that have been illuminated (hit) by the Cherenkov light cone emitted by a charged particle traversing the detector, as shown in Figure 4.1.

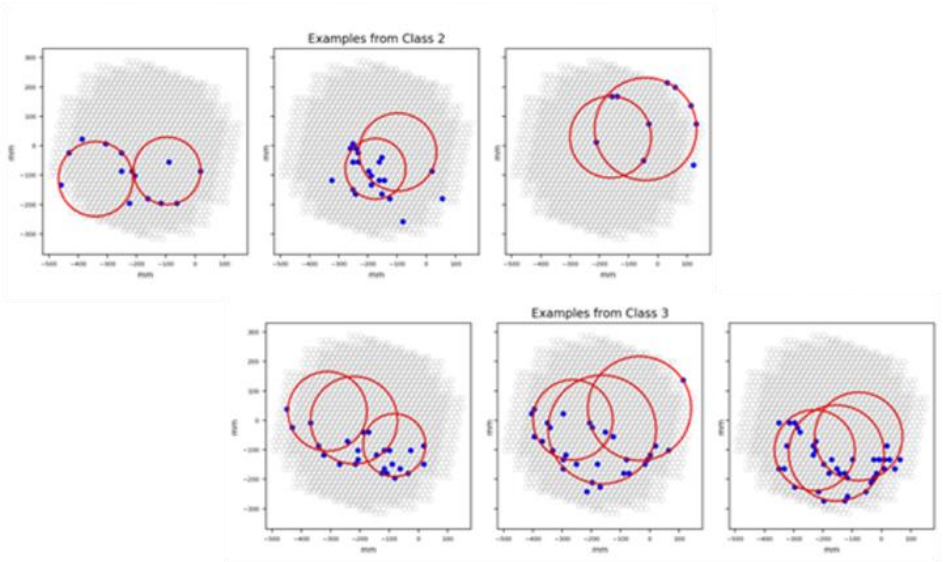


Figure 4.1: Examples of events belonging to class 2 and 3 (2 or >=3 charged particles) as detected by the array of RICH photomultipliers (blue dots are the hit photomultipliers, red circles are the tracks reconstructed offline by the NA62 experiment offline analysis software framework)

Figure 4.2 depicts the workflow for the generation of processing Kernels implementing convolutional neural networks designed for the inference tasks in RAIDER; these kernels are then integrated in the FPGA design as HLS kernels in the APEIRON framework, as described in deliverable D4.1.

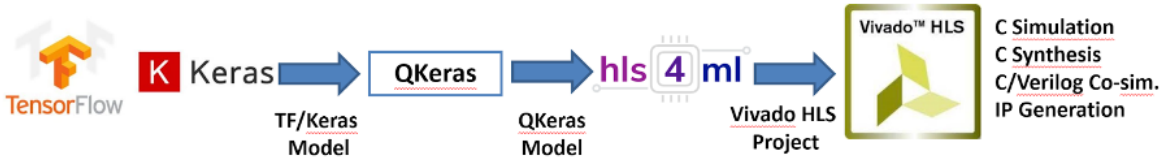


Figure 4.2: The workflow for the generation of CNN kernels in RAIDER

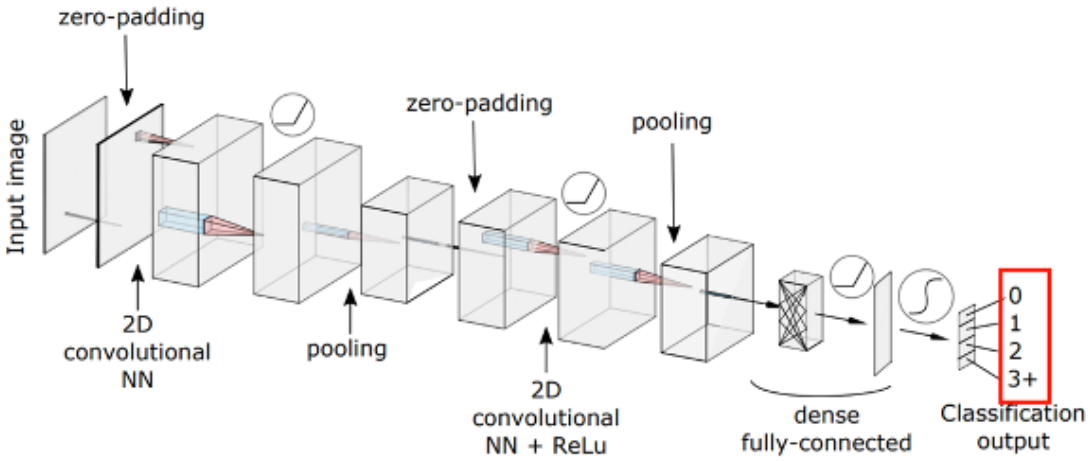
Using this workflow, design targets (efficiency, purity, throughput, latency) and constraints (mainly FPGA resource usage) must be taken into account and verified at any stage:

1. TensorFlow/Keras [2]: on this first stage the NN architecture (number and kind of layers) and representation of the input is designed, then using an appropriate training strategy (class balancing, batch sizes, optimizer choice, learning rate, etc.), the network is trained and KPIs can be measured. If they don't meet the targets the process is repeated, modifying input representation and the CNN architecture.

2. QKeras [3]: in this second stage, the original TF/Keras NN model is modified by searching iteratively the minimal representation size in bits of weights, biases and activations, possibly by layer that preserves the expected KPIs. For the RAIDER application, the neural network generated through this quantization step, yielded a neural network that uses an 8-bit fixed point <8, 1> representation for weights and biases and 16-bit fixed point <16, 6> for activations.

3. HLS4M L[4]: the QKeras model is translated into the corresponding Vivado HLS implementation (annotated C++ code) by means of the HLS4ML Python package. Several handles are available at this stage to guide the translation, e.g. tuning of REUSE FACTOR configuration parameter (low values yield low latency, high throughput, high resource usage design), also clock frequency can be set.

4. Vivado HLS [5]: C/Verilog co-simulation for rapid verification of performance and synthesis of kernel IP to be integrated in the APEIRON framework.



Layer (type)	Output Shape	Param #
input1 (InputLayer)	[(None, 16, 16, 1)]	0
conv1 (Conv2D)	(None, 16, 16, 8)	80
act1 (Activation)	(None, 16, 16, 8)	0
maxp1 (MaxPooling2D)	(None, 8, 8, 8)	0
conv2 (Conv2D)	(None, 8, 8, 8)	584
act2 (Activation)	(None, 8, 8, 8)	0
maxp2 (MaxPooling2D)	(None, 4, 4, 8)	0
flatten (Flatten)	(None, 128)	0
fc3 (Dense)	(None, 16)	2064
act3 (Activation)	(None, 16)	0
fc4 (Dense)	(None, 4)	68
softmax (Activation)	(None, 4)	0

Total params: 2,796

Figure 4.3: Details of the designed Convolutional Neural Network model

Following this workflow, we designed a lightweight Convolutional Neural Network having just 2796 parameters and suitable to be implemented on a FPGA, along with the corresponding representation of the input data. The designed CNN model, represented in Figure 4.3, has been deployed on a Xilinx Alveo U200 FPGA with a very limited resource usage. This CNN receives as input a compressed representation of the original event in form of a B&W 16x16 image, as depicted in Figure 4.4.

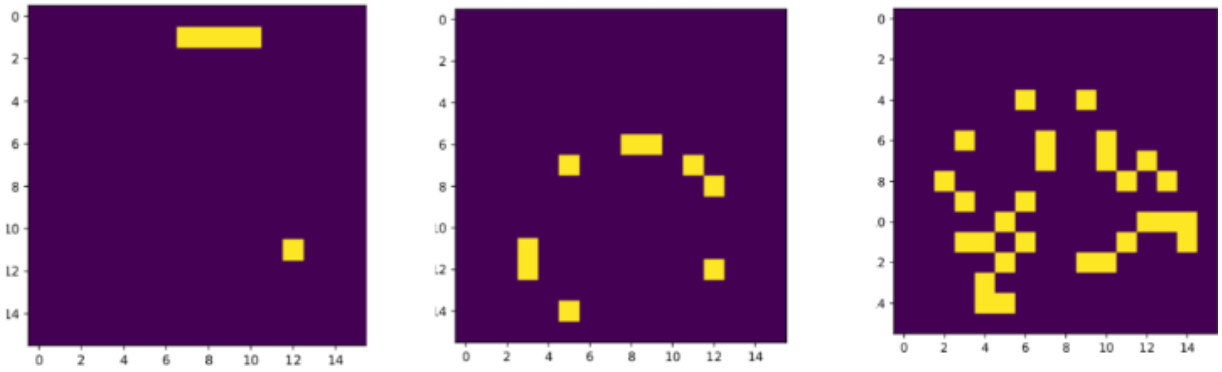


Figure 4.4: Example of input images for the CNN (left class 0, center class 1, right class 2).

The relevant KPIs for the CNN implementation are its efficiency (or recall), purity (or precision), and the throughput (in terms of processed events per second) and energy efficiency (in terms of processed events per Joule) it delivers when integrated in the FPGA processing pipeline.

These KPIs are reported in deliverable D6.2 for two different FPGA design configurations, integrating one or two replicas of the above-described CNN.

5. Conclusions

This deliverable presents several contributions for improving the efficiency of CNN applications at the IDV level. We have presented re-materialization strategies to reduce the memory requirements of training large neural networks, and a combination of re-materialization and model parallelism to efficiently use multiple GPUs when training. We have also discussed improvements for the inference phase, by providing algorithmic and technical developments for better resource usage on heterogeneous servers, using both the CPU cores and the available accelerators. These developments are still on-going work and will be carefully evaluated in the upcoming months.

We have designed and improved a 2D convolution kernel for the FPGA using the OmpSs@FPGA framework, competitive with CPU state-of-the-art implementations. In addition, the OmpSs@FPGA framework has been extended with new features added to the llvm compiler fork that compiles the FPGA code in order to feed it to the vendor HLS tool. The new developments have added mixed precision types support to the compiler to improve the processing of newly developed applications, CNNs in particular.

References

[1] E. Commission, “Grant Agreement 671668 - TEXTAROSSA: exploring Manycore Architectures for Next-GeneratiOn HPC systems.” 2015.

[2] François Chollet et al. 2015. Keras. https://github.com/fchollet/keras

[3] Claudionor Coelho. 2019. QKeras. https://github.com/google/qkeras

[4] Duarte J, Han S, Harris P, Jindariani S, Kreinar E, Kreis B, Ngadiuba J, Pierini M, Rivera R, Tran N and Wu Z 2018 Journal of Instrumentation 13 P07027–P07027

[5] Vivado Design Suite User Guide High-Level Synthesis UG902 (v2020.1), Xilinx, San Jose, CA, 2021

[6] Beaumont, O., Eyraud-Dubois, L., Herrmann, J., Joly, A., and Shilova, A. (2019). Optimal Checkpointing for Heterogeneous Chains: How to Train Deep Neural Networks with Limited Memory. Research Report RR-9302, INRIA.

[7] Zhao, X., Le Hellard, T., Eyraud-Dubois, L., Gusak, J., & Beaumont, O. (2023, July). Rockmate: an Efficient, Fast, Automatic and Generic Tool for Re-materialization in PyTorch. In ICML (International Conference on Machine Learning) 2023.

[8] Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Gonzalez, J., Keutzer, K., and Stoica, I. Checkmate: Breaking the memory wall with optimal tensor rematerialization. Proceedings of Machine Learning and Systems, 2:497–511, 2020.

[9] Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H. J., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. (2019). GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In NeurIPS.

[10] PyTorch developers, PyTorch. https://pytorch.org/

[11] PyTorch developers, Pipe implementation: https://pytorch.org/docs/stable/pipeline.html#torch.distributed.pipeline.sync.Pipe

[12] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009, 23:187-198, February 2011

[13] ONNX Runtime developers, ONNX Runtime, 2021. https://onnxruntime.ai/

[14] NVIDIA Triton Inference Server. https://developer.nvidia.com/triton-inference-server

[15] Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A., and Keckler, S. W. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In The 49th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 18. IEEE Press, 2016.

[16] Zinkevich, M., Weimer, M., Li, L., and Smola, A. J. Parallelized stochastic gradient descent. In Advances in neural information processing systems, pp. 2595–2603, 2010.

[17] OmpSs@FPGA framework source code. https://github.com/bsc-pm-ompss-at-fpga/

[18] OmpSs@FPGA User Guide. https://pm.bsc.es/ftp/ompss/doc/user-guide

[19] Aarrestad, Thea and others. Fast convolutional neural networks on FPGAs with hls4ml. In JINST, pp 7-27, 2018

[20] HLS4ML https://github.com/fastmachinelearning/hls4ml