DEEPHEALTH

D2.3 EDDLL Hardware algorithms and adaptation to HPC

Project ref. no.	**H2020-ICT-11-2018-2019 GA No. 825111**
Project title	Deep-Learning and HPC to Boost Biomedical Applications for Health
Duration of the project	1-01-2019 – 31-12-2021 (36 months)
WP/Task:	WP2/T2.3
Dissemination level:	PUBLIC
Document due Date:	01/06/2020 (M17)
Actual date of delivery	22/06/2020 (M17)
Leader of this deliverable	BSC
Author (s)	Santiago Marco (BSC), Lluc Alvarez (BSC), Miquel Moreto (BSC), Roberto Paredes (UPV), Jon Ander Gomez (UPV), Carles Hernandez (UPC), Jose Flich (UPV), Eduardo Quiñones (BSC), Philippe Dore (CEA), Tomas Teijeiro (EPFL), Marina Zapater (EPFL)
Version	V3

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 82511

Document history

Version	Date	Document history/approvals
1	04/05/2020	First draft contents to request for contributions
1.1	01/06/2020	Partner's contribution received
2	04/06/202	Version for review with all partners contributions
3	22/06/202	Version reviewed by internal peer-review, Technical Manager and Project Coordinator

Table of contents

1 Introduction

The European Distributed Deep Learning Library (EDDLL) is an optimized library for distributed deep learning. It allows the definition of different neural network architectures and its executions on different hardware platforms and accelerators. Moreover, the library aims to offer tailored implementations that can be efficiently executed on general CPUs, Graphical Processing Units (GPUs), Field Programmable Gate Arrays (FPGA). The different implementations are managed by COMPSs, which transparently distributes the work across the available accelerators.

The goal of WP2 is to develop and deploy EDDLL using the different use cases of the DeepHealth project. The design and implementation of EDDLL aims to replicate the state-of-the-art in terms of Deep Learning functionality in a way that they will also be ready to run on hybrid and heterogeneous High Performance Computing (HPC) + Big Data clusters.

All the activities described in this delivery are mainly related to task T2.3 “EDDLL adaptation to heterogeneous HPC hardware”. For this task effort, the goal is to optimize and adapt key algorithms and methods deployed in EDDLL to HPC heterogeneous components; mainly high-end CPUs, GPUs, and FPGAs. This task also tackles the deployment of those algorithms and the adaptation to the foreseen runtime environment of the HPC system.

The work efforts of this task started at different project months and, because of that, some developments have been active longer and have made further progress. It is important to highlight that this deliverable has not been substantially impacted by the world-wide crisis related to the COVID19 pandemia. However, many development and testing activities have suffered considerable disruptions due to the strict restriction to access the workplace and reach the required facilities to perform the tasks associated with this deliverable.

In this deliverable we present the results of the characterization, optimization, and adaptation of EDDLL for HPC infrastructures. The deliverable is divided into six main sections. First, Section 2 presents the results of the performance profiling and characterization of the EDDLL algorithms. Then, Section 3, 4 and 5 present the algorithm adaptations and optimizations for CPUs, GPUs and FPGAs, respectively. Section 6 describes the design of the FPGA dataflow accelerator for efficient inference, and Section 7 contributes with the results of the parallelization of the pyEDDLL training operation on HPC infrastructures. Finally, Section 8 concludes this deliverable.

2 Profiling and performance characterization

The EDDLL is an optimized tensor library for distributed deep learning. It allows the definition of different neural network architectures and its executions on different hardware platforms and accelerators (a.k.a. computing services). The library is built around the concept of *Tensor*, which offers a wide range of functionalities and connects to other components of the library. The Tensor interface is device independent, thus enabling a strong decoupling between the network training logic and the hardware implementation. This way, the EDDLL offers tailored implementations that can be efficiently executed on general CPUs, Graphical Processing Units (GPUs), Field Programmable Gate Arrays (FPGA), or distributed using COMPSs.

It is broadly known that executing deep neural networks (DNN) requires a significant amount of memory and computation. In recent years, deep learning has become ubiquitous and many tools employ DNNs to solve a wide variety of problems. Hence, modern processors offer specific features to accelerate these types of workloads. Moreover, high performance hardware platforms allow high degrees of parallel execution in order to further accelerate the execution of DNNs. Nevertheless, full exploitation of the available hardware resources requires careful evaluation and adaptation fo the DNNs. Depending on the specific target, different bottlenecks can arise and limit the potential of the targeted accelerator.

In this section, we present the performance evaluation of the EDDLL on the high performance computing (HPC) infrastructure. The goal of this characterization is to identify the most time-consuming parts of the algorithms and to analyze their potential for being accelerated with parallel CPUs, GPUs and FPGAs. First, we present the experimental setup based on DeepHealth's use cases and a comprehensive performance characterization of the library. After that, we indicate the main performance bottlenecks and the underlying causes. Then, we propose solutions to overcome these inefficiencies and adapt EDDLL to be executed on HPC hardware in a way that fully exploit its capabilities. Furthermore, we present experimental results of the optimized version together with some recommendations for further performance improvement. We conclude by compiling the most relevant results found during the experimentation, analysis, and optimization.

2.1 Experimental setup

As described in Deliverable D1.2, the Barcelona Supercomputing Center (BSC) provides a set of HPC resources to study how the most relevant performance limitations of biomedical applications can be effectively overcome on modern HPC infrastructures. BSC hosts several HPC machines, being Marenostrum 4 the most relevant one. Marenostrum 4 consists of 48 racks with 3456 nodes of two Intel Xeon Platinum chips (Skylake), each with 24 cores running at 2.1 GHz. The whole cluster sums up a total of 165,888 processors and 390 Terabytes of main memory and is capable of reaching peak performance of 11.15 PetaFLOP/s. The nodes are interconnected by a low-latency Omnipath network with a fully connected fat-tree topology.

Additionally, Marenostrum 4 is equipped with a cluster featuring emerging technologies that combines IBM POWER9 CPUs and NVIDIA Volta GPUs (V100). This cluster is composed of 54 nodes, where each node is equipped with 2 POWER9 processors, 4 Volta GPUs and 6.4TB of NVMe. The nodes are like the ones in the Sierra supercomputer at Lawrence Livermore National Laboratories, which is the 3rd fastest supercomputer in the top500 list. This cluster is very suitable both for HPC and for machine learning workloads, as it reaches a peak performance of 1.57 PetaFLOP/s in double precision computations.

Several profiling tools have been used for the experimental evaluation and characterization of the EDDLL. Foremost, we have utilized the performance counters for Linux (PCL or perf) to access the hardware Performance Monitoring Counters (PMC), monitor specific kernel-based subsystem events, and collect high-level performance metrics like cycles and instructions executed per function. Also, the profiling tool Intel® VTune™ Amplifier has been used on the Xeon platform to support further characterization experiments. In this evaluation, the VTune profiler has facilitated

tracking multithreading synchronization and scalability problems, capturing specific PMCs of the Intel Xeon processor that is being used, and also understanding interactions between the critical computing kernels within the library. Additionally, in order to model the memory usage and detect any issues and limitations related to the allocation/deallocation patterns, we have performed several analyses using the Valgrind memory profiling tool.

Throughout the experimentation we have benchmarked the existing use cases provided by the DeepHealth developers. Notwithstanding that the EDDLL and the use cases are still under development, we have focused our analysis on the available prototypes to characterize the main core functions of the library. In particular, we have evaluated the UC12 (Skin cancer melanoma detection) for both applications, skin lesion classification and skin lesion segmentation. In this case, we have selected a representative subset of the ISIC 2019 challenge database composed by randomly selecting 200 images for training (2 epochs) and 200 for testing inference. All the results have been obtained executing EDDLL v0.5.4-alpha for the DeepHealth use case pipeline v8491a67. The support libraries used for the experimentation are ECVL v0.2.1, Eigen v3.3.90, GTest v1.10.0, OpenCV v4.1.1, and Protobuf v3.5.1.

2.2 Experimental results and performance characterization

The primary goal of this task effort is to characterize the performance of the EDDLL. Therefore, this task focuses on modeling the performance, hardware exploitation, and scalability of the EDDLL on the selected HPC platforms. Following a top-down approach, we first evaluate the overall execution performance and memory consumption running on the MN4 HPC hardware platform (i.e., the Intel Xeon processor) for different batch sizes and working threads. This way, we narrow down the analysis to detect critical functions in the library and identify specific performance bottlenecks. We analyze the interrelation between the compute kernels (i.e., computationally intensive functions) identifying the critical call-path between them. Then, gathering the information retrieved from the profiled executions, we identify the most important performance bottlenecks and the cause of these inefficiencies. This analysis establishes the foundations for a CPU-optimized version, adapted to the HPC hardware infrastructure, which can overcome these performance penalties and improve the running time of the EDDLL.

2.2.1 Execution time, memory footprint, and scalability

In Table 2.1, we present the running time (measured in seconds), peak memory (measured in GBytes), and CPU average usage obtained for different executions of the UC12 on an Intel Xeon Platinum computing node. It is important to highlight that these results vary from the ones presented on D5.1 (Table 4) as the EDDLL is under development and performance results may change on newer versions of the library.

Batch Size	Time(s)	Memory(MB)	CPU Usage
1	5865.80	2559.94	31.63%
2	5815.43	3028.91	32.29%
4	5953.64	3883.02	33.17%
8	5874.45	5684.04	33.02%
12	5753.99	2253.99	32.88%
24	5744.85	2244.85	32.79%
48	5710.89	2438.25	32.69%

***Table 2.1**. Time performance (measured in seconds) and memory consumption (measured in GBytes) of the UC12 Classification (training 200 samples for 2 epochs) executed on an Intel Xeon Platinum varying number of threads and the input batch size (using EDDLL v0.5.4-alpha).*

As the results in Table 2.1 show, the execution time is insensitive to the input batch size and the average memory required is 3.1GB. Also, note that the EDDLL baseline (*v0.5.4-alpha*) is insensitive to the number threads set by the user. At the moment, the library automatically detects the number of virtual CPUs available and spawns as many threads as it deems necessary to maximize the system utilization. For the executions presented in Table 2.1, the EDDLL assumes that all CPUs on the system are available (i.e., 48 in the case of the Intel Xeon Platinum). However, the average CPU utilization is low for all the executions, barely reaching a 33% of the peak core capacity.

A more detailed analysis (Figure 2.1) shows that the CPUs are being underutilized, almost idle for more than 77% of the execution time. Figure 2.1 shows that the executions of the UC12 Classification alternate between parallel regions that attempt to exploit the 48 cores and sequential regions where single-threaded functions are executing meanwhile the remaining 47 cores are idle. As a result, the overall core exploitation is very low with scarce peaks of parallel activity.

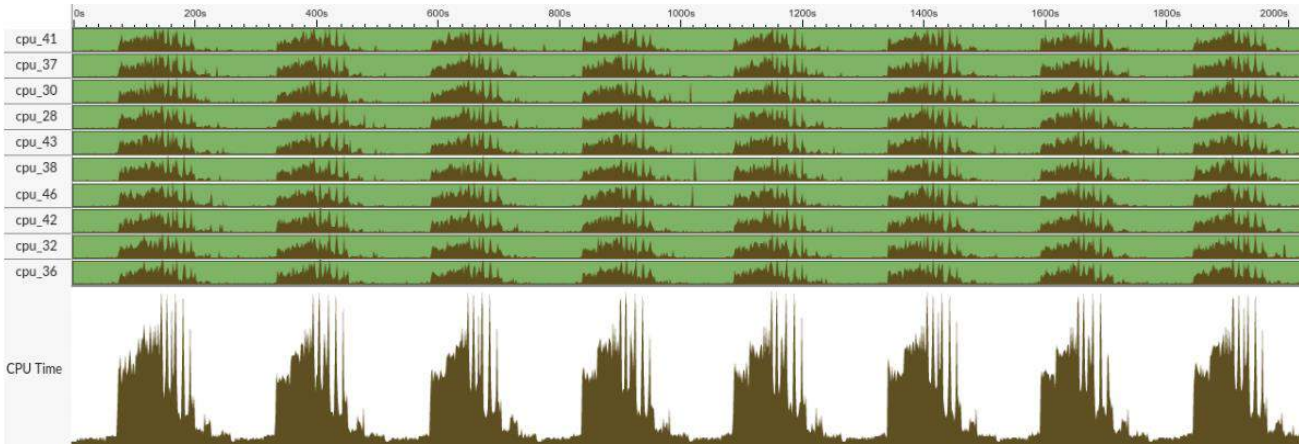

***Figure 2.1**. Sample CPU utilization histogram for 10 CPUs and the overall system core exploitation (i.e., CPU Time) executing the baseline version of the EDDLL (UC12, training 200 samples for 2 epochs) on an Intel Xeon Platinum.*

2.2.2 Critical functions and hardware exploitation

In order to narrow down the source of inefficiencies and determine the main performance bottleneck, we perform several function-level profile analysis of the EDDLL. In Table 2.2, we present the most time consuming functions together with some basic hardware performance metrics.

These results show that almost 96% of the user time is spent doing basic linear algebra operations (e.g., matrix multiplications, additions, or transpositions) within the Eigen library. Moreover, 96% of the retired instructions correspond to Eigen's functions. Then, for all the main matrix multiplication functions, the overall CPI achieved is reasonably good, near the theoretical optimum 0.25. In contrast, the general block packed product kernel (gebp_kernel) depicts a very high CPI rate of 3. This result is specially concerning as the gebp_kernel is among the most critical functions of the EDDLL. Furthermore, the matrix data mapping functions (e.g., copy of matrices) also achieve a poor CPI indicating performance issues when handling matrix memory.

Besides, the results point out a non-negligible amount of cycles (~2 TCyles) spent on OpenMP functions. These functions correspond to thread handling and synchronization operations. Hence the poor CPI rate which suggests synchronization issues or penalties associated with inadequate thread handling.

In addition, results on Table 2.2 indicate that proper EDDLL and ECVL functions have a minor impact on the overall execution time (as for the baseline implementation). More specifically, the most time-consuming functions of the EDDLL are im2col, add_pixel, and get_pixel, which amount to less than the 1% of the overall user time.

Function	% Time	Cycles	Inst.	CPI	Module
Eigen::internal::general_matrix_matrix_product	63.50%	42.51 T	126.61 T	0.34	Eigen
Eigen::internal::general_matrix_matrix_product	17.17%	11.47 T	34.16 T	0.34	Eigen
Eigen::internal::gebp_kernel	11.82%	7.92 T	2.64 T	3.00	Eigen
Eigen::internal::general_matrix_matrix_product	3.45%	2.32 T	6.90 T	0.34	Eigen
func@0x189a0	1.78%	1.21 T	57.63 G	20.99	OMP
func@0x18810	1.32%	887.60 G	46.50 G	19.09	OMP
im2col	0.46%	318.22 G	430.09 G	0.74	EDDLL
get_pixel	0.15%	105.56 G	79.43 G	1.33	EDDLL
Eigen::internal::blas_data_mapper	0.14%	87.50 G	14.69 G	5.96	Eigen
add_pixel	0.08%	48.86 G	62.75 G	0.78	EDDLL
Eigen::internal::blas_data_mapper	0.07%	46.90 G	9.12 G	5.14	Eigen
Eigen::internal::blas_data_mapper	0.02%	11.34 G	9.79 G	1.16	Eigen
cpu_mpool2D._omp_fn.0	0.01%	9.10 G	21.58 G	0.42	EDDLL
ecvl::RearrangeChannels	0.01%	6.86 G	14.69 G	0.47	ECVL
cpu_conv2D_back._omp_fn.4	0.01%	7.00 G	6.68 G	1.05	EDDLL

***Table 2.2**. Profile summary of the most time-consuming functions for the baseline version of the EDDLL (UC12 Classification, training 200 samples for 2 epochs) executed on an Intel Xeon node. For each function, the table displays user time percentage (with respect to the overall user time taken by the execution), cycles, instructions, cycles per instruction (CPI), and the module each function belongs to (i.e., EDDLL, ECVL, OpenMP, or Eigen).*

Focusing on those functions performing actual DNN computations, we proceed to analyze a more detailed profile report that will help understand the source of the main performance bottlenecks. In the following, we present a brief description of the performance counters captured using Intel Vtune and presented in Table 2.3.

- Retiring: This metric represents the pipeline slots fraction utilized by useful work, meaning the issued uOps that eventually get retired. Retiring of 100% would indicate the maximum possible number of uOps retired per cycle has been achieved. Maximizing Retiring typically increases the Instruction-Per-Cycle metric.
- Front-end Latency: This metric represents the pipeline slots fraction during which the CPU was stalled due to front-end latency issues (e.g., instruction-cache misses or fetch stalls after a branch misprediction).

- Front-end Bandwidth: This metric represents the pipeline slots fraction during which the CPU was stalled due to front-end bandwidth issues (e.g., inefficiencies in the instruction decoders).
- Bad Speculation: Bad Speculation represents the pipeline slots fraction wasted due to incorrect speculations (e.g., mispredicted branches or incorrect data speculation followed by memory ordering nukes).
- Memory Bound: This metric measures the fraction of slots during which the pipeline could be stalled due to demand load or store instructions.
- Divider: This metric represents the fraction of cycles the divider unit was active (i.e., divisions and square roots).
- Port Utilization: This metric represents the fraction of cycles during which an application was stalled due to core non-divider-related issues (e.g., data-dependency, overloading of specific ports).
- FPU Utilization: This metric represents the utilization of the vector capacity by the vectorial floating point computations.

Function	Eigen Matrix Multiplication	im2col	get_pixel	add_pixel	Eigen Data Mapper	ECVL Rearrange Channels
Retiring	11.4%	40.1%	20.2%	30.9%	3.7%	41.9%
Front-end Latency	54.0%	11.2%	2.9%	16.8%	0.1%	0.0%
Front-end Bandwidth	49.4%	11.6%	12.8%	19.0%	0.3%	10.4%
Bad Speculation	19.0%	0.0%	11.5%	10.4%	1.2%	0.0%
Memory Bound	0.6%	16.0%	24.8%	2.0%	90.5%	3.6%
Divider	0.0%	42.1%	56.0%	70.6%	0.0%	57.3%
Port Utilization	97.7%	16.2%	21.9%	18.0%	3.7%	61.4%
FPU Utilization	25.0%	6.3%	6.3%	6.3%	0.0%	6.3%

***Table 2.3**. Selected performance counters of the most time-consuming functions for the baseline version of the EDDLL (UC12, Training 200 samples for 2 epochs) executed on an Intel Xeon node.*

First and foremost, results in Table 2.3 show that the execution of Eigen's functions related to matrix multiplication suffer from continuous stalls in the front-end. That is, for more than the 50% of the pipeline slots, the front-end was stalled. As a result, only 11,4% of the pipeline slots were utilized by useful work. In spite of this, note that the hardware operation ports appear to be saturated meanwhile the floating-point unit is only used at 25% of its capacity. After rejecting other possible explanations, these numbers come to suggest that the microcode sequencer is issuing uOps to the back-end to address some modes of operation (Microcode assist), like in Floating-Point assists. This performance issue is concerning due to the relevance of these operations for the EDDLL performance as it can lead to severe execution slowdowns.

Concerning the remaining Eigen's operations (i.e., data mapping functions), we can see that these functions are severely memory bound. That is, the pipeline is stalled for more than 90% of the

available slots. In turn, this causes that only 3.7% of the pipeline slots are utilized by useful work. These stalls in memory operation suggest pressure in the memory hierarchy, unaligned memory accesses, pressure on high demand load or store instructions, or other memory related issues.

For the remaining functions, that is, the most time consuming functions from the EDDLL and ECVL libraries, the results show that their retirement rate is also slightly low. Although the port utilization is low, the divider utilization is significantly high (70.6% for the add_pixel function). This means that the code for these functions is stressing this operational unit. This suggests it is advisable to review these functions towards accelerating or optimizing these operations.

2.2.3 Call graph and detailed analysis of critical kernels

In the previous section we have identified the most critical functions and their exploitation of the available hardware resources. This way, we aim to narrow our analysis on those functions with critical performance. For the sake of optimizing and adapting these functions to the HPC infrastructure, it is important to understand their interaction with the library. In this section, we aim to give a general overview of the usage of these computational kernels, their function within the library, and their interaction with other functions.

Considering the results presented in Table 2.2, we can assess that the most computational intensive functions involve performing matrix operations with the layers of the DNN. In particular, those operations lie at the core of the forward and backward propagation functions during the training of the DNN. Unsurprisingly, these functions represent the most computing-critical components of every modern DL framework. In the case of the EDDLL, these functions represent more than 99% of the execution time, devoted to training the model.

Figure 2.2 depicts the timeline of the EDDLL functions for training batches of images. Batches are trained in sequential order using the function train_batch_t. For each batch trained, the majority of the time is spent on do_forward and do_backward (forward and backward propagation, respectively).

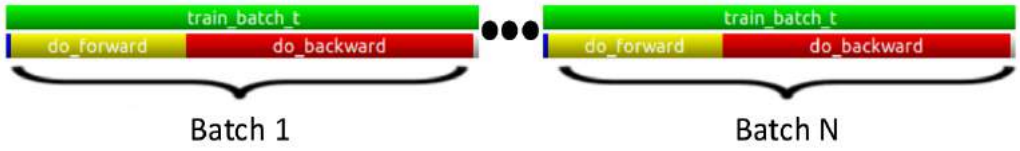

***Figure 2.2**. Timeline representation of the EDDLL functions involved in training a DNN using N batches of input images.*

More specifically, Figure 2.3 summarizes the call graph from the high-level function train_batch_t down to the individual kernel functions that perform the matrix operations using Eigen; that is, EDDLL's most time-consuming kernels. The functions cpu_conv2D, cpu_conv2D_grad, and cpu_conv2D_back are in charge of performing the 2D convolution for the forward and backward propagation. Roughly speaking, these are the functions responsible for calling Eigen's routines to perform a series of matrix operations on each sample of the batch, taking more than 99% of the execution time.

Note that each of these functions operate over a single input batch in mutual exclusion from other batches (or epochs) during training. This allows offloading the computations of the whole batch to a hardware accelerator. However, computing each batch in mutual exclusion represents a potential bottleneck as the training process is forced to serialize computing work between batches and epochs.

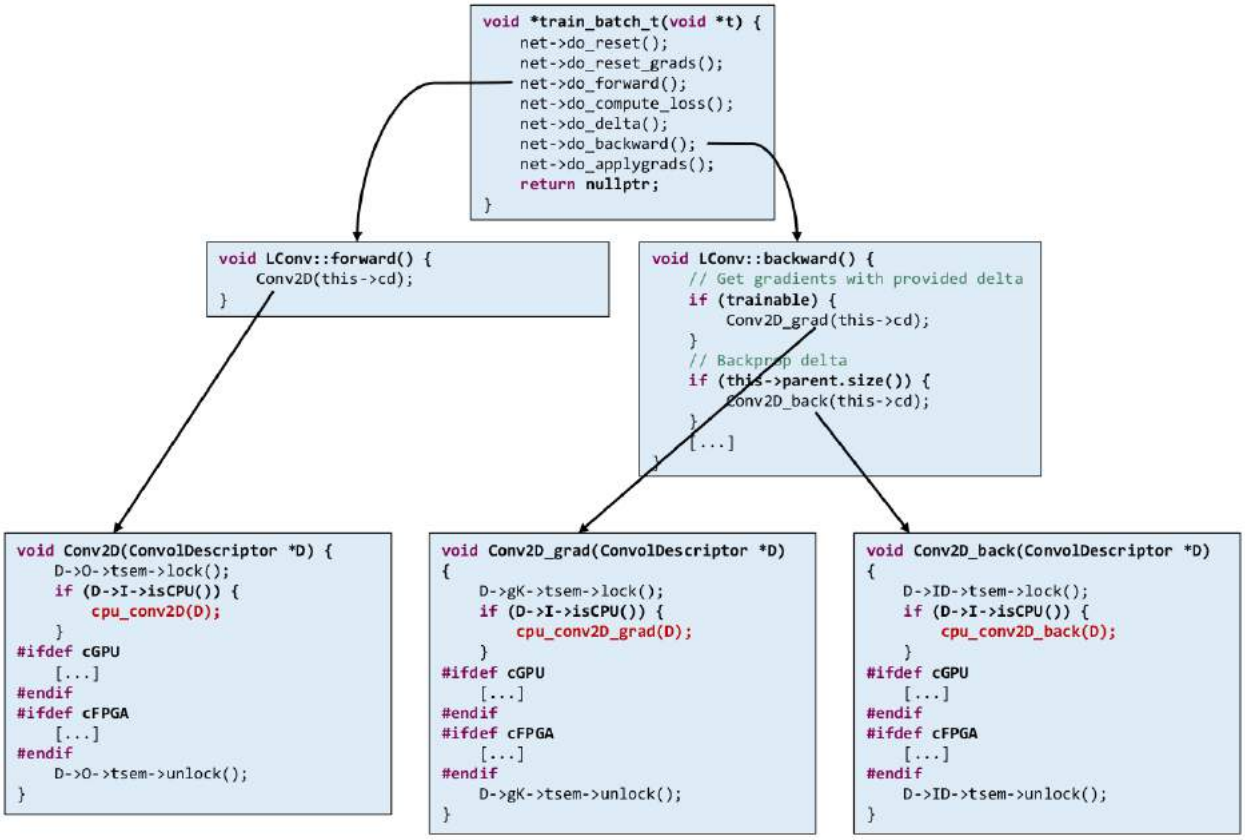

***Figure 2.3**. Simplified pseudocode of the DNN train method (traing_batch_t) and the chain of calls down to cpu_conv2D, cpu_conv2D_grad, and cpu_conv2D_back.*

More in depth, Figure 2.4. shows a simplified version of the cpu_conv2D, cpu_conv2D_grad, and cpu_conv2D_back functions. In essence, these functions perform matrix additions, multiplications, and transpositions through calls to Eigen's routines.

In particular, the functions cpu_conv2D and cpu_conv2D_back depict a map computation pattern where all the batches can be processed independently. In contrast, the function cpu_conv2D_grad presents a reduction pattern that forces to combine (i.e., matrix addition) the product of processing each sample of the batch into a single matrix (i.e., D->matgK). Nonetheless, cpu_conv2D_grad code can be reorganized so that the reduction is performed at the end of the function, allowing independent computations at the cost of extra memory to store the intermediate matrices.

Anyhow, these computation patterns allow processing each batch sample independently. Thus, the three for-loops on Figure 2.4 can be parallelized using as many threads as the size of the batch (never exceeding the total number of logical CPUs to avoid oversubscription). The implementation of this coarse-grain parallelization was suggested during the early stages of T2.4 and has been successfully integrated into the current EDDLL baseline.

```
void cpu_conv2D(ConvolDescriptor *D)
{
    float* ptrO=D->O->ptr;
    float* ptrI=D->ptrI;

    // Map memory to Eigen
    new(&D->matK) Eigen::Map<Eigen::MatrixXf>(...);
    new(&D->matI) Eigen::Map<Eigen::MatrixXf>(...);

    for (int b=0;b<D->I->shape[0];b++) {
        float* ptrO=D->O->ptr+(b*osize);
        float* ptrI=D->ptrI+(b*isize);
        Eigen::Map(matI,ptrI,b); // Map mem to Eigen
        Eigen::Map(matO,ptrO,b); // Map mem to Eigen
        im2col(b,D,ptrI,0);
        matO=matI*D->matK;
    }
}
```

```
void cpu_conv2D_grad(ConvolDescriptor *D)
{
  // Map memory to Eigen
  new(&D->matgK) Eigen::Map<Eigen::MatrixXf>(...);

  for (int b=0;b<D->I->shape[0];b++){
    float* ptrD=D->D->ptr+(b*osize);
    float* ptrI=D->ptrI+(b*isize);
    Eigen::Map(matI,ptrI,b); // Map mem to Eigen
    Eigen::Map(matD,ptrD,b); // Map mem to Eigen
    D->matgK+=matI.transpose()*matD;
  }
}
```

```
void cpu_conv2D_back(ConvolDescriptor *D)
{
    float* ptrD=D->D->ptr;
    float* ptrI=D->ptrI;

    // Map memory to Eigen
    new(&D->matK) Eigen::Map<Eigen::MatrixXf>(...);
    new(&D->matI) Eigen::Map<Eigen::MatrixXf>(...);

    for (int b=0;b<D->I->shape[0];b++){
        float* ptrD=D->D->ptr+(b*osize);
        float* ptrI=D->ptrI+(b*isize);
        Eigen::Map(matI,ptrI,b); // Map mem to Eigen
        Eigen::Map(matD,ptrD,b); // Map mem to Eigen
        matI=matD*D->matK.transpose();
        im2col(b,D,ptrI,1);
    }
}
```

***Figure 2.4**. Simplified pseudocode of the CPU implementations of cpu_conv2D, cpu_conv2D_grad, and cpu_conv2D_back.*

All in all, each kernel function performs a simple matrix operation (i.e., multiplication, auditions, and transposition) for each batch sample. At this level, Eigen allows fine-grain parallelization of these operations. The current EDDLL baseline allows Eigen to use as many threads as cores are available in the hardware platform. However, we must stress the fact that parallel computations entail certain overheads (e.g., thread creation, resource allocation, or thread dispatch) and synchronization penalties. For those reasons, it is key that the computation payload provided to these critical functions is large enough to maximize the utilization of the hardware resources available. Otherwise, the EDDLL executions can incur in severe penalties that, ultimately, would render the available computing cores underutilized.

2.2.4 Performance bottlenecks

Thus far, we have presented a comprehensive characterization of the performance of the EDDLL. We have identified the most critical functions and their interactions with the library. Additionally, we have analyzed the overall exploitation of the available hardware resources and identified the main performance problems. In this section, we summarize the execution bottlenecks found on the EDDLL baseline and their underlying cause, so we can point out the most effective optimizations to tackle these inefficiencies.

Overall, executions using the EDDLL are computationally bound and require large amounts of memory, depending on the batch size configured. In particular, the most computationally intensive kernels of the library are devoted to the forward and backward propagation during the training of the DNN. These critical kernels perform relatively simple matrix operations using Eigen's linear algebra functions. These functions involve intensive computations on floating-point data. As the results on Table 2.3 indicate, these functions miss to fully exploit the Single-Instruction Multiple-Data (SIMD) capabilities of the architecture. More importantly, their execution is limited by severe front-end stalls, caused by the microcode sequencer (MS) when processing denormals numbers. For that, it is paramount to overcome the penalties caused by the denormals' executions (e.g., activating the FTZ/DAZ modes) and enable the use of wider SIMD instructions that can fully take advantage of the vector processing units available in the processor.

Moreover, we have found that many of the memory operations (i.e., load and stores) operate on unaligned memory. This causes a significant pressure on the memory resources of the processor (Table 2.3), which degrades the CPI of memory intensive functions (e.g., Eigen's memory mapping operations) and ultimately slowdowns the overall execution of the EDDLL. Furthermore, working with unaligned memory prevents the use of memory-aligned SIMD instructions that can vastly improve the execution of the critical kernels.

Beyond these hardware exploitation inefficiencies, the overall CPU utilization is very low (Figure 2.1). On the higher CPU-utilization parts of the execution, these results reveal that the utilizations is below the core peak capacity. This is the result of severe thread handling overheads and large synchronization waiting times (Table 2.2). Whereas, on the lowest CPU-utilization parts, we can clearly identify parts of the code that run sequentially or have such strong synchronization restrictions that cannot exploit more than a single core. In consequence, the performance of the execution is heavily limited, especially in highly parallel platforms such as the HPC infrastructure of Marenostrum4. For that, we suggest investigating and evaluating the parallelization strategy implemented on the EDDLL, identify performance penalties derived from mismanagement of parallel threads, and evaluate the parallelization of those functions that are currently executed sequentially on a single thread.

Altogether, these performance issues cause significant slowdowns and the underutilization of the available hardware resources. Nonetheless, the majority of these performance issues can be easily mitigated or resolved. For this reason, the current EDDLL baseline shows large potential for performance improvement.

2.2.5 Performance of the PyEDDLL in CPU and GPU environments

In this section we analyze the profiling results obtained for the PyEDDLL in different CPU/GPU environments, and we compare them with another reference Deep Learning library (Keras/Tensorflow). Since most use cases are not yet fully ported to the EDDLL/ECVL ecosystem, we will rely on the network architecture designed for UC13: Epileptic seizures detection. This network is described in detail in deliverable "D2.6: EDDLL adaptation to cloud environments", and basically consists of 5 convolutional layers followed by 2 dense layers. The input is just the raw multi-lead EEG signal, so no preprocessing is required.

The system used for the benchmarks was a computing server with 2 Dual Intel Xeon Gold 6154 CPUs (36 cores, 72 threads), with a maximum processor speed of 3.70 GHz and 384 GB of RAM memory, while the GPU is a NVIDIA Tesla T4 with 16 GB of memory. The installed operating system is CentOS v7.7.

Library	CPU/GPU	Time per epoch (s)	Speedup	Nº Measures
Pyeddll	CPU 1 core	452 ± 7,09	Reference	150
Pyeddll	CPU 4 cores	246 ± 4,66	1,84x	150
Pyeddll	GPU	5,569 ± 0,02	81,16x	750
Keras	CPU 1 core	17,09 ± 0,37	Reference	750
Keras	CPU 4 cores	11,05 ± 0,35	1,54x	750
Keras	GPU	0,724 ± 0,31	23,60x	750

***Table 2.4**. Speedup comparison between single-core, multi-core and GPU execution for UC13 training.*

Table 2.4 shows the profiling results of training the network with the full CHB-MIT public database (https://physionet.org/content/chbmit/1.0.0/) for a different number of epochs, both for the PyEDDLL and Keras implementations. If we take as a reference the single-core execution, the speedup achieved by PyEDDLL is significantly higher than that achieved by Keras/Tensorflow, both for the multi-core environment and specially for GPU, in which the speedup is more than 80x. However, if we focus on the absolute execution time numbers, there is still a great margin for improvement on the PyEDDLL, which shows running times per epoch that are between 7,7x and 26x slower than Keras/Tensorflow for the same architecture and training task. No relevant differences were observed in the classification accuracy of the different models.

It is important to notice that these results correspond to a single network architecture and base dataset, so as more use cases are ported to the EDDLL it will be possible to complement these results and provide a more accurate overview of the performance offered by the library.

3 Algorithm adaptation and optimizations for CPUs

In this section, we present a series of optimizations based on the performance insights described on Section 2. As a result of this task effort, we have implemented these optimizations and algorithm adaptations into an improved version of the EDDLL we denote as CPU-Optimized EDDLL. In the following subsections, we described the optimizations implemented into the CPU-Optimized EDDLL, the main inefficiencies and bottlenecks these modifications target, and some technical suggestions to further improve the performance of the library. Additionally, we present a comprehensive characterization of the CPU-Optimized EDDLL. To sum up, we briefly recapitulate the most notable results of the CPU adaptation and optimization.

3.1 Bottlenecks and performance optimizations

3.1.1 Floating-point denormal execution

Broadly speaking, a denormalized number (a.k.a. denormals) is any non-zero number with magnitude smaller than the smallest normal number encoded using the IEEE 754 floating point standard.

In most applications, processing denormal numbers is uncommon. For that reason, many processors don't handle them directly and generate a trap which resolves their operation using microcode (i.e., small programs injected into the execution stream). This process is referred to as x87 floating-point assists (FP Assists) and tends to stress the microcode sequencer (MS), causing multiple machine resets, and resulting ultimately in major front-end stalls. In the case of executions

using the EDDLL (UC12, Classification training), this issue causes the stall of the front-end during more than 50% of the pipeline slots (Table 2.3).

In general, denormal numbers are an issue for the performance of most processors. In some situations, handling denormals can produce drops in performance up to 10x or more. For that reason, many processors provide instructions to process these numbers in hardware, without calling a software exception, at the cost of losing strict IEEE 754 standard compliance. In the case of Intel, the SSE and SSE2 instructions were added towards solving this issue. Thus, note that SSE/SSE2 and x87 operations are handled differently.

To avoid serialization and performance issues due to denormals, it is recommended to use the SSE and SSE2 instructions to set Flush-to-Zero (FTZ) and Denormals-Are-Zero (DAZ) modes within the hardware to improve performance for floating-point applications.

In order to assess the impact of processing denormals using the EDDLL, in Table 3.1 we present a comparison of the baseline version (Denormals) against a compilation of the EDDLL activating the FTZ/DAZ modes (No-Denormals). The results reveal that the baseline version triggers FP Assists 59,1% of the times it has to execute a FP operation. Meanwhile the No-Denormals version does not require any at all. This is reflected in the dramatic reduction of the amount of switches to the microcode sequencer (MS Switches) and the machine clears. Ultimately, this is reflected in the total number of retired instructions (i.e., 5.7x less instructions) and reduces the execution cycles by 5,67x (core cycles, not wall-clock time). Consequently, this affects the overall execution time achieving a speedup of 2.57x (Table 3.2).

	Denomals	**No-Denormals**
Cycles	355.00 T	62.55 T
Instructions	919.73 T	160.57 T
CPI	0.39	0.39
FP float	41.97 G	36.75 G
FP double	7.71 G	6.86 G
FP 128bits	8.74 T	7.58 T
FP 256bits	0	0
FP 512bits	0	0
FP Assist	29.41 G	0
MS Switches	425.05 G	20.90 G
Machine Clears	55.36 G	336.87 M
MS uops	2.87 T	189.06 G

***Table 3.1**. Performance counters related to floating-point execution and microcode sequencer (MS) for the baseline version and the No-Denormals version of the UC12 Classification (training 200 samples for 2 epochs) executed on an Intel Xeon Platinum.*

	Time (s)	Cycles	Inst.	CPI	Loads	Stores	L1Miss	L2Miss	L3Miss
Denomals	5710.89	355.00 T	919.73 T	0.39	235.44 T	263.26 G	49.12 G	9.09 G	1.89 G
No-Denormals	2259.63	62.55 T	160.57 T	0.39	38.00 T	223.96 G	42.93 G	8.60 G	1.72 G

***Table 3.2**. Performance results for the baseline version and the No-Denormals version of the UC12 (training 200 samples for 2 epochs) executed on an Intel Xeon Platinum.*

3.1.2 Unaligned memory accesses and SIMD instructions

It is broadly documented that operations on unaligned memory have a significant impact on performance. Some hardware instructions suffer from performance penalties if executed on unaligned memory addresses (e.g., being forced to execute extra instructions to load/store the data), some others can't actually perform reads on non-aligned addresses (i.e., generate an exception).

For instance, some of Intel's SSE instructions mandate special alignment. Most notably, 128bits, 256bits, or 512bits SIMD instructions require alignment to 16Bytes, 32Bytes, or 32Bytes memory boundaries, respectively. In case the alignment restrictions are not met, the performance of these instructions can severely degrade or even raise an exception if the alignment is mandatory. This way, operating with aligned memory allows using wider SIMD instructions and can reduce the latency and bandwidth requirements of memory instructions. For that reason, alignment and arrangement of data memory can make a big difference in performance. Using memory-aligned SIMD instructions not only reduces the overall number of instructions, allowing a more efficient vectorization, but also increases the performance of load/store instructions. In turn, this alleviates the pressure on the front-end resources and improves the overall performance.

The EDDLL implements a custom memory allocator for large memory allocations, mainly Tensors' memory, called get_fmem. This function is implemented on top of the operator 'new' that ultimately requests virtual memory from the OS. However, the operator 'new' does not guarantee that the memory returned is aligned beyond the 8 Bytes boundaries (nor does malloc or mmap for that matter).

Notwithstanding the advanced memory-alignment features that the Eigen library has in place, doing the allocation using this custom allocator prevents the usage of wider SIMD instructions as the memory alignment cannot be guaranteed. This results in the execution of suboptimal Eigen code that is robust enough to handle any memory alignment at the cost of a major performance loss.

Figure 3.1 shows the compiled x64-86 machine code for the EIGEN_GEBP_ONESTEP core function using the AVX512 vector instruction set. This function lies at the core of the matrix to matrix multiplication within the Eigen library. The SEE code blocks handle unaligned memory access, working with 4x less data per instructions (i.e., 128bits).

```
0xb98e3  1671   prefetcht0z  (%rdx)
0xb98e6  1671   vmovapsz  -0x400(%rdx), %zmm4
0xb98ed  1671   vmovapsz  -0x3c0(%rdx), %zmm2
0xb98f4  1671   vmovapsz  -0x380(%rdx), %zmm9
0xb98fb  1671   vbroadcastss1  (%rax), %zmm21
0xb9901  1671   vbroadcastss1  0x4(%rax), %zmm1
0xb9908  1671   vbroadcastss1  0x8(%rax), %zmm0
0xb990f  1671   vbroadcastss1  0xc(%rax), %zmm20
0xb9916  1671   vfmadd231ps %zmm21, %zmm4, %zmm15
0xb991c  1671   vfmadd231ps %zmm21, %zmm2, %zmm14
0xb9922  1671   vfmadd231ps %zmm21, %zmm9, %zmm11
0xb9928  1671   vfmadd231ps %zmm1, %zmm4, %zmm17
0xb992e  1671   vfmadd231ps %zmm1, %zmm2, %zmm7
0xb9934  1671   vfmadd231ps %zmm1, %zmm9, %zmm8
0xb993a  1671   vfmadd231ps %zmm0, %zmm4, %zmm16
0xb9940  1671   vfmadd231ps %zmm0, %zmm2, %zmm5
0xb9946  1671   vfmadd231ps %zmm0, %zmm9, %zmm12
0xb994c  1671   vfmadd132ps %zmm20, %zmm19, %zmm4
0xb9952  1671   vmovaps %zmm4, %zmm1
0xb9958  1671   vmovaps %zmm2, %zmm0
0xb995e  1671   vfmadd132ps %zmm20, %zmm18, %zmm0
0xb9964  1671   vfmadd132ps %zmm20, %zmm13, %zmm9
```

***Figure 3.1**. Intel x86-64 AVX512 assembly code for the EIGEN_GEBP_ONESTEP core processing block from Eigen's matrix to matrix multiplication function (22 instructions).*

For the reasons presented before, we recommend modifying the custom allocator to make it return memory blocks aligned to 64Bytes (i.e., 512bits words). This can be easily achieved using C11 aligned_alloc function, POSIX compliance posix_memalign() function, or enhancing the custom allocator get_fmem to guarantee memory alignment. In turn, this will allow compiling the EDDLL natively (i.e., in gcc using "-march=native"). For instance, in the case of the Intel Xeon Platinum, this will allow using AVX512 instructions and maximize the FPU utilization. Moreover, this will reduce the overall number of retired instructions and improve considerably the execution time of the EDDLL.

3.1.3 Multithreading performance and nested parallelism

Results from Section 2.2 (Figure 2.1) suggest a low CPU utilization, possibly due to overheads on thread handling and synchronization penalties. Table 3.3 presents a summary of the performance of the system calls for the execution of UC12 (training) using the baseline EDDLL and the CPU-Optimized EDDLL. For the baseline version, the results show that there is an unusual number of calls to futex and clone. Indeed, the remarkably high number of calls to futex, and the large time spent on them is a clear indication of frequent and lasting synchronization waiting times. In the same manner, the elevated number calls to clone indicates the recurrent creation and destruction of threads during the execution. Moreover, the creation of new threads involves setting up internal thread structures (e.g., attributes) and allocating working memory, resulting in an elevated number of calls to mmap, munmap, madvise, mprotect, and set_robust_list.

A more detailed inspection of the EDDLL baseline reveals that the library employs two different multithreading libraries (i.e., OpenMP and Pthreads) to implement parallel processing sections. This is generally discouraged as the two multithreading implementations can be incompatible, causing many performance issues, or even runtime errors. In the case of the EDDLL, the Pthread library is used to create and dispatch a thread that executes the training for a single batch, and the OpenMP library is used to parallelize the processing of each individual sample within the batch (Figure 2.4).

OpenMP implements a pool of working threads to avoid the overheads of creation/destruction of threads. Whenever a parallel section is encountered, or a task is ready for being processed, the OpenMP dispatcher selects a thread from the pool and assigns work to it. In practice, this mechanism is proven to be very efficient. Nevertheless, the OpenMP pool of threads is stored in

thread-local data which, in the case of the EDDLL, refers to the pthread created to handle each batch. Whenever the whole batch has been processed, the pthread is destroyed (i.e., pthread_join) and the OpenMP thread pool too. As a result, the OpenMP is repeatedly creating and destroying pools of thread each time a new batch is processed.

Furthermore, the EDDLL allows Eigen to further parallelize the execution of its functions. This triggers the OpenMP nested behavior (OMP_NESTED), that allows each OpenMP thread to spawn threads on its own. Nonetheless, the newly created threads by the Eigen library (nested to each batch-sample thread) are not returned to the pool of threads. OpenMP's internal implementation of the pool opts to destroy these surplus of threads, leaving the pool with no more than the maximum number of configured working threads (i.e., OMP_NUM_THREADS).

	Baseline		**CPU-Optimized**	
System Call	**Calls**	**CPU Time (s)**	**Calls**	**CPU Time (s)**
futex	1303802	2633.97	18334	92.96
clone	417379	0.56	49	0.00
set_robust_list	417378	0.47	48	0.00
madvise	417377	11.76	0	0.00
mmap	389320	0.81	2130	0.23
munmap	389151	3.62	1896	2.09
mprotect	387401	0.41	786	0.27
read	5373	0.02	5398	0.02
open	1975	0.01	2000	0.00
brk	942	0.00	1148	0.00
fstat	815	0.00	815	0.00
write	25	0.00	25	0.00

Table 3.3. *Comparison of the number of system calls performed by the baseline and the CPU-Optimized version of the EDDLL executing the UC12 Classification (training 200 samples for 2 epochs) on an Intel Xeon Platinum.*

On top of that, a closer analysis reveals that the amount of useful work assigned to each of the Eigen's threads (computational payload) is insufficient to compensate for the cost of thread setup, dispatch, and synchronization. Profiling the executions of the UC12 Classification (training), we observe that the average matrix size processed by Eigen ranges from 2 to 5 million elements (i.e.,

FP). Even the largest observed matrix processed by Eigen only contains 28 million elements. The resolution of this fine-grain parallelism demands a more efficient thread management.

Regrettably, these unfortunate conditions make the EDDLL spend a non-negligible amount of time creating and destroying threads. Due to this disarrangement, many of the running threads became idle, meanwhile the working threads lack enough computational payload that compensates.

```
#include <omp.h>
void Net::run_snets(void *(*F)(void *t))
{
  struct tdata td[100];

  int comp=snets.size();
  if (batch_size<comp) comp=batch_size;

  #pragma omp taskloop num_tasks(comp)
  for (int i = 0; i < comp; i++) {
    td[i].net = snets[i];
    F(&td[i]);
  }
}
```

***Figure 3.2**. Simplified pseudocode of the run_snets function in charge of dispatching/offloading the processing of batch to each working thread/device.*

All in all, the situation requires selecting only one multithreading library for parallel processing. In our opinion, OpenMP is the most suitable choice due to its versatility, high-level abstraction, and parallel expressiveness. This way, we suggest replacing the Pthread creation on “net_api.cpp” at the method Net::run_snets with an OpenMP taskloop construction (Figure 3.2). Similarly, we strongly advise to replace all mutexes, locks or semaphores with OpenMP locks (i.e., omp_lock_t). We advise for reducing the interactions between different multithreading frameworks and attain a robust code based solely on OpenMP.

Besides, we deem the parallelization of Eigen’s functions ineffective provided that the selected batch size is larger than the number of available CPU cores. In those cases where each core can be assigned the workload of processing a sample, a coarse-grain parallelization approach is preferred in terms of performance. Therefore, we recommend configuring Eigen to execute single-threaded (i.e., calling setNbThreads(0)) and to disable the nesting and dynamic behaviors of openMP (i.e., OMP_NESTED and OMP_DYNAMIC, respectively).

3.1.4 Sequential execution of support functions

So far, we have analyzed those performance issues that limit the performance of the inherently parallel parts of the library. Nevertheless, the executions of the UC12, using the EDDLL, depict sequential code regions that deteriorate the overall performance. Consider the results on Table 3.4 showing the scalability speedup of the CPU-Optimized EDDLL increasing the number of threads and the batch size. Roughly speaking, these numbers suggest that the optimized version scales on-par to the theoretical optimum up to 8 threads. However, executions using 24 threads or more are 2x-3.4x below the optimal performance. Provided that there is enough workload to distribute across threads (i.e., the bath size is sufficiently large) and the thread management is adequate, the parallel sections of the code should reach peak CPU. Hence, we suspect that the sequential code sections of the EDDLL have become the main performance bottleneck after incorporating all the previous optimizations.

		Threads					
		1	2	4	8	24	48
Batch size	1	1.00x	1.04x	1.04x	1.05x	1.05x	0.99x
	2	1.00x	1.81x	1.83x	1.86x	1.80x	1.72x
	4	1.00x	1.85x	3.24x	3.31x	3.24x	3.07x
	8	1.00x	1.90x	3.40x	5.59x	5.48x	5.30x
	12	1.00x	1.90x	3.43x	4.74x	7.43x	7.09x
	24	1.00x	1.92x	3.49x	5.95x	10.98x	10.81x
	48	1.00x	1.92x	3.51x	6.02x	11.54x	14.56x

***Table 3.4**. Scalability of the CPU-Optimized version of the EDDLL, executing the UC12 (training 200 samples for 2 epochs) on an Intel Xeon Platinum, varying the number of threads used and the image batch size.*

Figure 3.3 helps to identify the culprit showing the CPU utilization histogram obtained from executing the UC12 (training) using the CPU-Optimized version of the EDDLL. On the histogram we observe the regions labeled as 1, 2, and 3 where the only thread active is the OpenCV dispatcher. More specifically, these regions correspond to the loading of the batch images from disk. Not only each image is loaded sequentially but also the whole batch has to be read before the next training process can begin. On top of that, OpenCV functions to read and prepare the samples take a non-negligible time to complete (i.e., 166 ms per image on average).

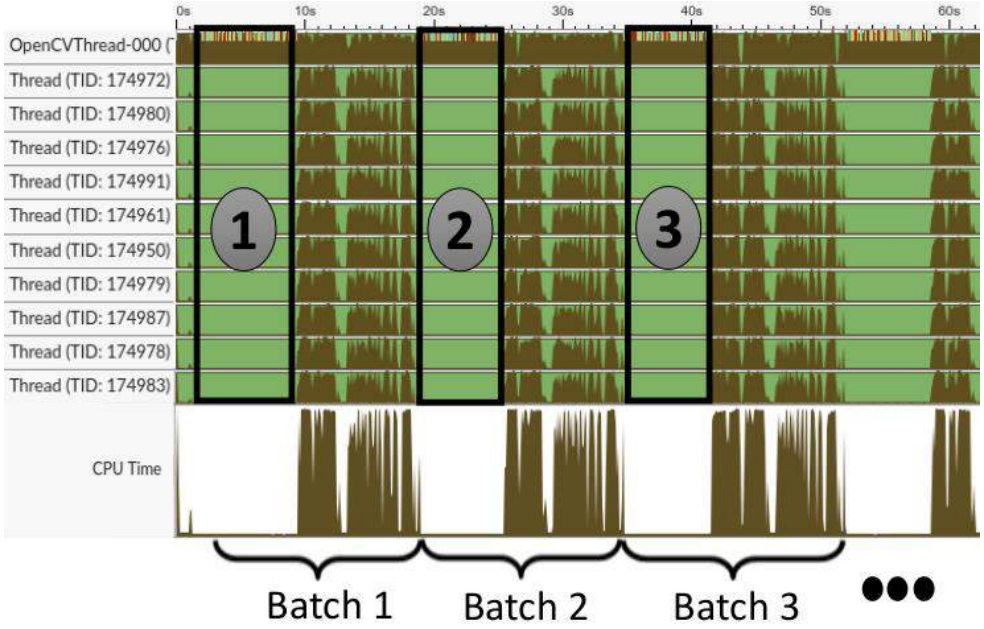

***Figure 3.3**. Sample CPU utilization histogram for 11 OS threads (i.e., OpenCV dispatcher and 10 EDDLL worker threads) and the overall system CPU exploitation (i.e., CPU Time) executing the CPU-Optimized version of the EDDLL (UC12 Classification, training 200 samples for 2 epochs) on an Intel Xeon Platinum. Sequential code sections have been squared in black and labeled 1, 2, and 3 corresponding with the batch of samples they read from disk.*

3.2 Performance characterization of the CPU-optimized EDDLL

As we outlined before, all the suggested optimizations have been incorporated on a CPU-Optimized version of the EDDLL. To further characterize the performance of the library and better understand its current limitations, we present the results of executing and profiling the optimized version. Likewise it was presented in Section 2, we first analyze the scalability of the library with an increasing number of threads and for different batch sizes. Then, we identify the most-time consuming functions when executing the CPU-Optimized library for both training and inference. Finally, we present experimental results on the Power9 HPC platform for CPU and GPU executions.

3.2.1 Execution time, memory footprint, and scalability

For the CPU-Optimized EDDLL, Table 3.5 shows the scalability results of the library, executing the UC12 classification (training), with an increasing number of threads and for different batch sizes. Compared to the single-thread execution, scalability is on-par to the theoretical optimum up to 8 threads. Then, increasing even the number of threads, cause that the penalization of executing the sequential sections of the library dominate the execution time. Although the overall execution time decreases significantly, the scalability speedup stalls reaching a peak improvement of 14.56x using 48 threads and sufficiently large batches (see Table 3.4).

Skin Lesion Classification (Training)		Threads											
		1		2		4		8		24		48	
		T	Mem	T	Mem	T	Mem	T	Mem	T	Mem	T	Mem
Batch size	1	2065	2.5	1990	2.5	1978	2.5	1970	2.5	1963	2.7	2079	2.9
	2	1973	2.9	1093	2.9	1080	2.9	1060	3.0	1097	3.1	1148	3.3
	4	1895	3.7	1025	3.8	585	3.9	573	4.0	585	4.1	618	4.3
	8	1861	5.4	981	5.5	547	5.7	333	6.3	340	6.4	351	6.7
	12	1761	7.2	928	7.3	514	7.6	372	8.1	237	8.8	249	9.0
	24	1761	12.7	915	12.8	504	13.0	296	13.5	160	15.8	163	16.0
	48	1757	23.8	917	23.8	501	23.9	292	24.3	152	26.5	121	29.9

***Table 3.5**. Time performance (measured in seconds) and memory consumption (measured in GBytes) of the CPU-optimized EDDLL, executing the UC12 Classification (Training 200 samples for 2 epochs) on an Intel Xeon Platinum, varying the number of threads and the input batch size.*

As for the executions of the UC12 classification (inference), the effect of the sequential code regions is even stronger. As the results in Table 3.6 show, increasing from 8 to 24 threads yields a small speedup of 2x; and increasing to 48 threads gives a marginal improvement of approximately 1s.

Skin Lesion Classification (Inference)		Threads											
		1		2		4		8		24		48	
		T	Mem	T	Mem	T	Mem	T	Mem	T	Mem	T	Mem
Batch size	1	461	1.6	442	1.6	440	1.6	441	1.6	440	1.6	443	1.6
	2	442	2.0	238	2.0	241	2.0	238	2.0	241	2.0	251	2.0
	4	438	2.8	230	2.9	130	2.9	130	2.9	131	2.9	130	2.9
	8	432	4.5	228	4.6	127	4.6	74	4.8	76	4.8	79	4.8
	12	418	6.2	220	6.3	120	6.3	87	6.5	55	6.6	55	6.6
	24	426	11.3	218	11.3	119	11.4	71	11.5	37	12.1	38	12.1
	48	421	21.5	220	21.5	119	21.6	69	21.7	36	22.3	29	23.1

***Table 3.6**. Time performance (measured in seconds) and memory consumption (measured in GBytes) of the CPU-optimized EDDLL, executing the UC12 Classification (Inference of 200 samples) on an Intel Xeon Platinum, varying the number of threads and the input batch size.*

3.2.2 Critical functions and hardware exploitation

More in detail, Tables 3.7 and 3.8 show a relation of the most-time consuming functions for the execution of the UC12 Classification performing training and inference, respectively. As expected, we find the same functions that appear profiling the baseline version. Nevertheless, Eigen's functions have remarkably improved their running time. This improvement on the Eigen execution causes that the most time-consuming functions of the optimized version are the EDDLL's im2col, get_pixel, and add_pixel.

UC12 Classification Training	% Time	Cycles	Inst.	CPI	Module
im2col	30.96%	1.64 T	2.26 T	0.73	EDDLL
Eigen::internal::gebp_kernel	15.63%	786.05 G	1.99 T	0.40	EDDLL
func@0x18810	10.39%	549.36 G	27.83 G	19.74	OMP
func@0x189a0	8.77%	461.57 G	23.01 G	20.06	OMP
get_pixel	8.08%	426.15 G	331.46 G	1.29	EDDLL
add_pixel	7.42%	397.70 G	474.80 G	0.84	EDDLL
cpu_conv2D_back._omp_fn.4	5.15%	274.56 G	17.54 G	15.66	EDDLL
func@0x18190	3.34%	176.70 G	8.07 G	21.90	OMP
Eigen::internal::blas_data_mapper	4.97%	263.31 G	45.25 G	5.81	EDDLL
cpu_conv2D_grad._omp_fn.2	1.48%	77.66 G	6.92 G	11.23	EDDLL
cpu_d_relu._omp_fn.1	0.75%	39.53 G	41.09 G	0.96	EDDLL
cpu_mpool2D._omp_fn.0	0.69%	36.50 G	88.04 G	0.42	EDDLL

cpu_fill_._omp_fn.2	0.62%	34.05 G	5.76 G	5.91	EDDLL
ecvl::RearrangeChannels	0.60%	32.21 G	66.48 G	0.48	ECVL
cpu_relu._omp_fn.0	0.59%	30.84 G	4.16 G	7.42	EDDLL
cpu_conv2D._omp_fn.0	0.56%	29.33 G	3.11 G	9.44	EDDLL

***Table 3.7**. Profile summary of the most time-consuming functions for the UC12 Classification (training 200 samples for 2 epochs), using the CPU-Optimized EDDLL, executed on an Intel Xeon Platinum. For each function, the table displays user time percentage (with respect to the overall user time taken by the execution), cycles, instructions, cycles per instruction (CPI), and the module or framework each function belongs to (i.e., EDDLL, ECVL, OpenMP, or Eigen).*

As for the inference, we conclude that it shares the same performance characteristics as the training. Note that the same functions appear in both Tables 3.7 and 3.8 with the same weight on the execution performance. For this reason, improving the performance of these functions has a direct impact on the overall execution time on both applications: training and inference.

UC12 Classification Inference	**% Time**	**Cycles**	**Inst.**	**CPI**	**Module**
im2col	22.37%	447.57 G	594.89 G	0.75	EDDLL
get_pixel	10.78%	211.22 G	166.96 G	1.27	EDDLL
func@0x18810	6.98%	138.51 G	6.92 G	20.03	OMP
Eigen::internal::gebp_kernel	6.48%	120.77 G	329.07 G	0.37	EDDLL
Eigen::internal::blas_data_mapper	46.62%	918.06 G	1.10 T	0.84	EDDLL
func@0x189a0	2.66%	53.08 G	2.43 G	21.89	OMP
cpu_mpool2D._omp_fn.0	0.85%	16.98 G	42.65 G	0.40	EDDLL
ecvl::RearrangeChannels	0.76%	14.76 G	31.23 G	0.47	ECVL
cpu_relu._omp_fn.0	0.72%	14.47 G	2.24 G	6.46	EDDLL
cpu_conv2D._omp_fn.0	0.67%	13.25 G	1.56 G	8.52	EDDLL
func@0x74b90	0.42%	8.42 G	10.74 G	0.78	EDDLL
decode_mcu	0.15%	2.95 G	7.60 G	0.39	ECVL
cpu_conv2D	0.12%	2.39 G	2.74 G	0.87	EDDLL
jpeg_idct_islow	0.12%	2.30 G	6.88 G	0.33	ECVL
cpu_copy._omp_fn.1	0.08%	1.63 G	65.00 M	25.08	EDDLL
fast_randn	0.07%	1.31 G	3.93 G	0.33	EDDLL

***Table 3.8**. Profile summary of the most time-consuming functions for the UC12 Classification (inference of 200 samples) execution on an Intel Xeon node. For each function, the table displays user time percentage (with respect to the overall user time taken by the execution), cycles, instructions, cycles per instruction (CPI), and the module or framework each function belongs to (i.e., EDDLL, ECVL, OpenMP, or Eigen).*

3.2.3 Performance results on IBM Power9 nodes

For the sake of completeness, we have reproduced the same experiments on a different HPC infrastructure (Power9 architecture) both for CPU and GPU executions of the CPU-Optimized EDDLL (Table 3.9). Generally speaking, the peak performance is similar to that obtained on the Intel Xeon Platinum. However, executions of the Intel platform have slightly better execution times; that is, 1,3x faster on average using the same number of threads.

UC12 Classification Training		Batch size						
		1	2	4	8	12	24	48
Threads	1	2547.1	2451.4	2361.1	2325.2	2229.7	2211.5	2210.7
	2	2484.3	1348.9	1258.2	1226.2	1169.5	1157.9	1148.9
	4	2492.1	1329.7	695.6	663.7	629.4	618.8	610.4
	24	2527.5	1349.2	714.9	428.7	286.8	193.7	179.0
	48	2586.0	1377.0	728.6	405.8	287.0	190.5	175.4
	90	2763.4	1460.7	762.0	419.2	295.1	194.3	168.1
	160	3145.1	1657.8	825.9	459.4	317.8	210.6	171.4
UC12 Classification Inference		**Batch size**						
		1	**2**	**4**	**8**	**12**	**24**	**48**
Threads	1	453.7	445.2	441.2	439.9	422.5	420.4	421.1
	2	450.5	238.0	231.2	230.2	221.7	220.2	220.3
	4	449.6	238.2	126.5	124.9	119.5	119.0	118.5
	24	457.2	239.7	130.8	76.0	55.3	39.9	38.1
	48	454.7	240.5	129.7	75.7	54.5	40.2	39.6
	90	463.4	245.1	130.6	74.4	54.4	38.0	35.4
	160	476.4	248.7	134.1	77.2	54.8	38.1	35.0

***Table 3.9**. Time performance (in seconds) of the CPU-optimized EDDLL, executing the UC12 Classification (training 200 samples for 2 epochs, inference of 200 samples, respectively) on a Power9 node, varying the number of threads and the input batch size.*

Moreover, Table 3.10 presents a succinct summary of the performance results obtained for the execution of the UC12 (training) running on GPU. The results show that GPUs are indeed quite efficient performing the matrix operations performed at the core of the EDDLL training routines. Nonetheless, their performance reaches a plateau for the same reason the CPU-Optimized does not scale beyond 8 threads (i.e., sequential loading of the input samples). In fact, a deeper analysis using the NVIDIA profiler reveals a 4.47% average utilization of the GPU with no overlapping kernels on execution at all.

UC12 Classification Training		Batch size					
		1	2	4	8	12	24
GPUS	1	86.91	142.33	96.57	72.62	61.89	61.86
	4	81.17	233.48	156.39	105.92	83.86	67.96

***Table 3.10**. Time performance (measured in seconds) of the GPU baseline EDDLL, executing the UC12 Classification (training 200 samples for 2 epochs) on a Power9 node (equipped with 4 NVIDIA V100 GPUs), varying the number of GPUs used and the input batch size. All executions were made using the default "low-memory" profile and delay=1. See Section 4, Table 4.3, for a complete discussion on the GPU EDDLL results processing the full ISIC dataset for different memory profiles and delay values.*

3.3 Summary of the CPU adaptation and optimization

In this section, we have presented a series of optimizations that effectively adapt the EDDLL to the HPC infrastructure when executed using regular CPUs. Compared to the baseline EDDLL, the optimized version achieves speedups ranging from 2.7x, using a single thread, to 47.3x using all the 48 cores available on the Intel Xeon Platinum (Figure 3.4). Moreover, compared to the GPU executions on the Power9 using an NVIDIA Voltas 100, the optimized EDDLL is just 1.7x slower.

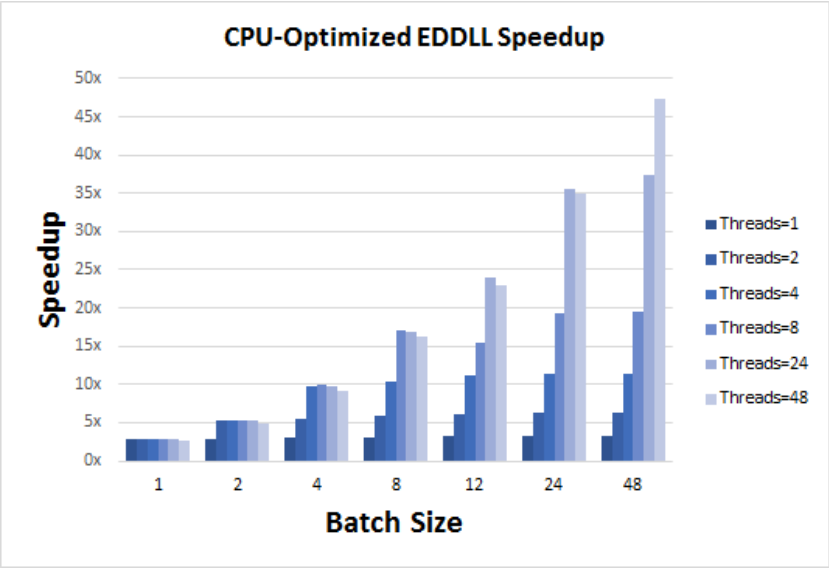

***Figure 3.4**. Speedup obtained by the CPU-optimized version of the CPU-Optimized EDDLL (adapted for HPC hardware CPU), executing UC12 (training 200 samples for 2 epochs) on an Intel Xeon Platinum, compared to the baseline implementation.*

4 Algorithm adaptation and optimizations for GPUs

In this section we present the adaptations and optimizations for GPUs. EDDLL provides a hardware abstraction through the Tensor objects. Tensors are n-dimensional arrays that can be placed in different devices: CPU, GPU and FPGA. Therefore the different operations must be implemented considering the different properties of the hardware in charge. All these considerations are implemented in a Tensor level and are provided to the rest of the EDDLL objects.

From this abstraction the EDDLL builds different objects that allow final users to define multiple neural network topologies. These different objects are mainly Layers and Nets. Layers define mathematical operations between Tensors and also the differentiation of those operations in order to allow error back propagation mechanism. Nets represent how the different Layers are interconnected in order to provide the desired functionality.

From Layers to higher level objects the user does not mind where the tensors are placed and how the mathematical operations are carried out. Therefore, Layers define the operations using common programming functions abstracted from the final hardware implementation.

The hardware specification is defined when the neural network is compiled by means of using a Computing Service object (CS_CPU, CS_GPU or CS_FPGA). The final user defines the network connections and optimization criteria independently of the hardware available and this computing service is the only difference. The user programming is completely agnostic to the hardware where the optimization is going to be executed.

One common limitation when dealing with big neural network topologies is the amount of memory that require. This amount of memory used to be a major drawback when using hardware accelerators like GPUs and FPGAs. In such situations the most common solution is to use several of these hardware accelerators to split the memory requirements into different units implementing a parallelization mechanism. The most common way of parallelization is “data parallelism” where the neural network is replicated in all the devices but the data that goes through the network is split. Usually this data parallelism is accomplished by means of splitting the batch size by the number of available accelerators.

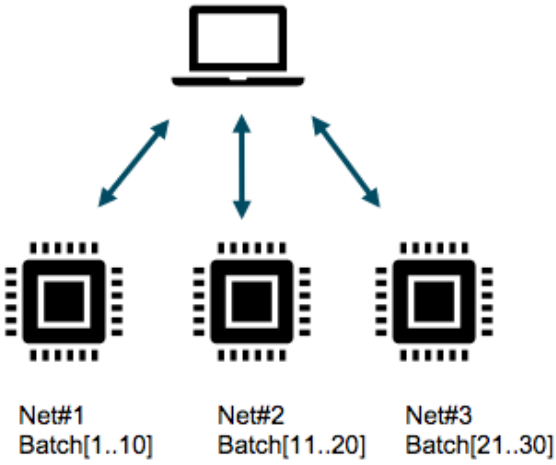

***Figure 4.1**. Diagram depicting the parallel distribution of the input batches to the available accelerators.*

As mentioned before, the neural network is replicated in all the devices and this entails that the parameters of the neural network are replicated in the different devices. However, the network parameters should be the same in all the hardware accelerators in order to ensure that the mathematical operations involved are the same independently on the device that is finally executing them. This is straightforwardly accomplished by means of the synchronization of the neural network

parameters. This synchronization is time consuming since very big tensors (network parameters) have to be collected from the hardware accelerators to main memory (other communication alternatives can be also explored) and then aggregated and distributed back to the hardware accelerators.

In order to speed up this mechanism EDDLL also proposes to perform a delayed synchronization with a parameter that controls the amount of delay. If this parameter is zero then we obtain a full synchronization, while larger than zero provides some kind of partial synchronization that speeds up significantly the whole process. Moreover, our experiments show that this delayed synchronization improves the generalization capabilities of the neural network since it adds some stochastic noise to the process.

4.1 Description of the EDDLL adaptation to GPUs

GPU accelerators can be selected in the process of building the neural network using the **CS_GPU** object. This is the unique difference in the program that users must consider in order to select among the different accelerators. The network definition and training procedure is agnostic about the hardware accelerator selected. Therefore, in the building command the user can select the Computing Service associated to the GPUs accelerators:

build(net, optimizer, {losses}, {metrics}, ***CS_GPU(params)*** *);*

Where params are the following:

- Binary vector selecting the available accelerators (mandatory)
- Memory level requirements (optional, default=”full_mem”)
- Delay synchronization parameter (optional, default=1)

EDDLL allows the user to select among the available GPUS. For instance, in a system with 4 GPUs installed the user can specify to use only the first and third GPU by means of using a binary vector in the CS_GPU object: *CS_GPU({1,0,1,0})*

EDDLL checks whether the system actually has these 4 GPUs installed and selects only those activated by the user. Regarding the memory levels, GPUS used to have low memory with respect to the main computer memory. This GPU memory is a worth resource and EDDLL allows three different memory levels:

- **full_mem**: EDDLL tries to get as much memory as possible to ensure fast processing.
- **mid_mem**: EDDLL performs some memory saving with a low speed degradation.
- **low_mem**: EDDLL performs a strong memory saving with a significant speed degradation.

An example of a 2 GPUs system where both GPUs are selected with a “low_level” setup: *CS_GPU({1,1}, ”low_level”)*

Finally, when using multiple GPUs we perform “data parallelism”. The neural network is replicated in all the GPUs but the data is split and distributed among them. In this scenario we must ensure that GPUs have the same parameters of the network. To this end the GPUs synchronize the parameters every batch of data processed. This synchronization entails a significant overhead in time. Therefore, we propose to postpone this synchronization after some batches, being this value set by the parameter of delay synchronization.

For instance, an example of a 2 GPUs system where both GPUs are selected with a “mid_level” setup and synchronized every 10 batches will be *CS_GPU({1,1}, 10, ”mid_level”).*

4.2 Performance results

Two different experiments have been carried out in order to assess the performance of the GPUs accelerators regarding the different parameters involved.

In the first experiment we deal with a classical problem (CIFAR10) that does not entail any special memory requirements (always full_mem). We build a VGG16 neural network, train with batch size 100 and run a total of 50 epochs. The following Table 4.1 shows the total training time in minutes varying the number of GPUs and the delay parameter.

GPUs	Delay	Time
1	**-**	50
2	**1**	49
2	**10**	33
2	**100**	31
2	**1000**	31

***Table 4.1**. Execution time (measured in seconds) for the CIFAR10 problem varying the delay and the number of GPUs used.*

In the second experiment we use a larger neural network topology and deal with the UC12 about skin classification. This experiment requires to take care of the amount of memory since the size of the images and the neural network topology are larger. The following Table 4.2 shows the time per epoch on the UC12 about skin classification. Each epoch comprises a total number of 19328 samples. The table shows the results using 1 GPU varying the batch size and memory requirements. The time is measured in seconds for one epoch.

	full	mid	low
8	668.5	755.1	647.2
16	-	-	614.7
32	-	-	745.5

***Table 4.2**. Execution time (measured in seconds) for the UC12 classification (training) using a single GPU RTX2080 and varying the batch size and the memory allocation strategy.*

The following Table 4.3 shows the performance results using a system with 2 GPUs, varying the batch size, memory requirements and delay synchronization parameters. The time is in seconds for one epoch. The void cells represent those setups that do not fit into memory.

	full			mid			low		
Delay	**100**	**10**	**1**	**100**	**10**	**1**	**100**	**10**	**1**
batch=8	571.3	859.4	3119.1	616.8	1001.5	3291.9	616.0	966.4	3230.7
batch=16	441.0	726.9	2286.7	477.4	767.8	2407.6	470.4	746.7	2404.8
batch=32	-	-	-	-	-	-	381.9	586.9	2634.1

***Table 4.3**. Execution time (measured in seconds) for the UC12 classification (training) using two GPUs RTX2080 and varying the batch size, the delay, and the memory allocation strategy.*

5 Algorithm adaptation and optimizations for FPGAs

The design complexity of efficient kernels for FPGAs relies on the fact that the architecture is adapted to the algorithm. Instead, in GPUs and CPUs the algorithms adapt to the fixed architecture and there are programming models and libraries (e.g. CUDA, MKL,…) that are already optimized for those fixed architectures. Instead, in FPGAs the design is imposed by the algorithm and thus, the efficiency of the use of the resources in the FPGA may be compromised. In addition, design times are much larger when compared to CPUs and GPUs counterparts. The design flow for FPGAs necessarily passes through different steps (software emulation, hardware emulation, final hardware design) which requires higher compilation (synthesis and floorplan processes) times. Indeed, typically we can expect several hours of compilation time for a set of kernels implemented on a target FPGA device. Moreover, depending on the algorithms, the final outcome may be disappointing since the final design may not fit on the device. The previous higher complexity of FPGA designs, combined with the expected availability of stable versions of both the EDDLL and ECVL, and the necessary refactors of such libraries as they are designed from scratch, leads to a criticality in the support of FPGAs for both libraries.

Another important aspect related to FPGA designs is the expected difference in performance for some specific kernel functions. It is well known that FPGAs are more energy efficient than CPUs and GPUs but their capabilities and capacities are much lower in some specific types of resources, e.g. FPU support. This leads to the necessity of tailoring and profiling the process of training neural networks in order to identify which kernel functions are frequent, which ones take more complexity and which ones should not be targeted to FPGAs. This profiling can be achieved with the EDDLL and ECVL when targeting other devices such as CPUs and GPUs. We target the CPU device for an analysis of frequencies of basic functions being used, thus steering the development of FPGA kernels for equivalent functions.

We target OpenCL as the programming model for developing the interface between the CPU and the FPGAs in EDDLL (and ECVL; see Deliverable D3.3). Kernels in FPGAs are developed using OpenCL or C and synthesized with High-level Synthesis (HLS) tools. In the EDDLL, however, GPU kernels are coded with CUDA and target NVIDIA GPUs as they exhibit higher performance. As a consequence, we have had to develop a complementary tool, a testbed platform which also targets OpenCL kernels on GPUs. Using OpenCL for both devices allows us to reduce our development times with faster processes of profiling, test, and development of kernel functions. Once tested, kernels are easily ported to the EDDL/ECVL libraries with minor modifications, mainly in the interface.

5.1 Native support for FPGAs in EDDLL

The EDDLL offers an API to define neural networks and execute on several target devices. The core part of the library is the Tensor class. The Tensor interface is device independent, allowing for a strong decoupling between logic behavior of the algorithm and its hardware implementations.

To abstract hardware devices EDDLL uses the concept of compute service designed and implemented at the core of the library, which can also be exploited with COMPSS to efficiently execute on different hardware resources. For the FPGA we have created a specific compute service. The text below illustrates how to specify in the EDDLL main file that the network model has to be executed in an FPGA.

build(net,sgd(0.01, 0.9), {"soft_cross_entropy"},{"categorical_accuracy"}, CS_FPGA({1}));

As we rely on the OpenCL API to implement host-to-FPGA communication, we extended the Tensor class to add a handle to the OpenCL buffer representing the tensor memory area in the device memory. As explained before, the FPGA implementation of the mentioned kernels is specified in OpenCL or C. The kernels take one or multiple buffers as input and output, along with any additional parameter. For each input or output buffer, a standard AXI port is generated to ease the

exploitation of parallelization opportunities. Scalar arguments are passed through a single AXI Lite interface, which is the same one used by the Xilinx runtime to launch kernels and query their status.

Then, we extended the EDDLL compilation flow to support the integration with the Vitis toolflow, used to generate a bitstream containing all the kernels that could be simulated or flashed on the physical FPGA. Figure 5.1 shows how the support for FPGAs is implemented in the EDDLL.

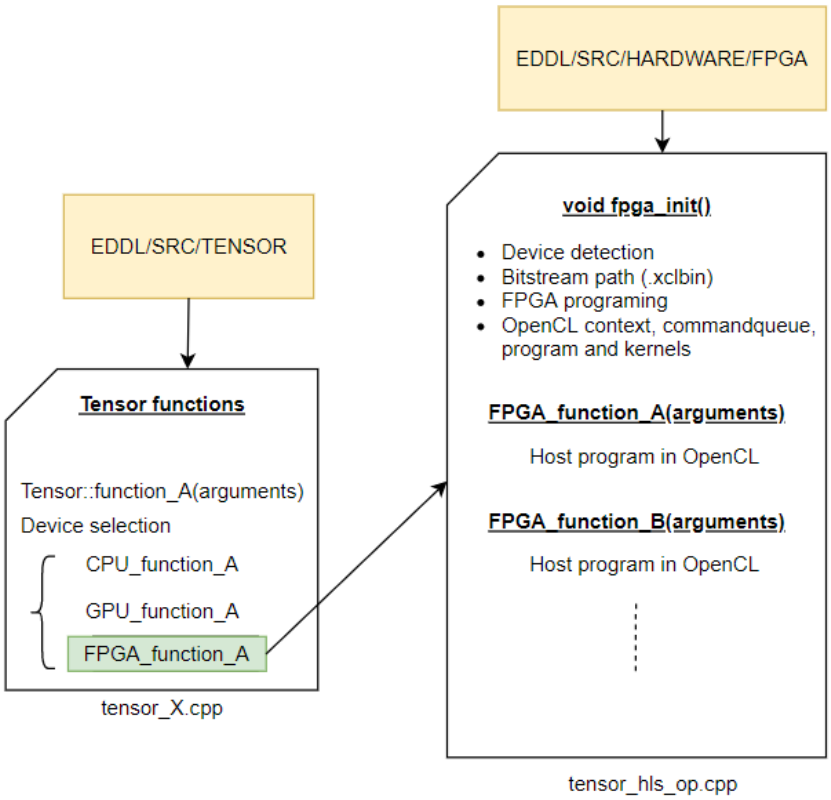

***Figure 5.1**. FPGA Integration Scheme.*

We illustrate the flow to support FPGAs with an example. In particular, we show how to support tensor addition (Tensor::add()) functionality included in tensor_math.cpp. First of all, in the main function which is intended to support, is added the device selection by the cFPGA flag. The function “fpga_tensor_add(scA, A, scB, B, C, incC)”, marked in the green box, is the host program that determines the kernel launch on the FPGA.

```
void Tensor::add(float scA, Tensor *A, float scB, Tensor *B, Tensor *C, int incC) {
   ////////////////////////////////////
   //// sum C=(sca*A)+(scb*B)
   //// or C+=(sca*A)+(scb*B) if incC is 1
   //// Dimensions and types must be compatible
   ////////////////////////////////////

   if ((A->device != B->device) || (A->device != C->device)) msg("Tensors in different devices", "Tensor::add_");
   if ((!eqsize(A, B)) || (!eqsize(A, C))) {
      A->info();
      B->info();
      C->info();
      msg("Incompatible dims", "Tensor::add");
   }

   C->tsem->lock();
   if (A->isCPU()) {
     cpu_add(scA, A, scB, B, C, incC);
     printf("CPU ADD \n" );
   }
#ifdef cGPU
   else if (A->isGPU())
    {
      gpu_addc(scA,A,scB,B,C,incC);
    }
#endif
#ifdef cFPGA
 else if (A->isFPGA())
    {
      fpga_tensor_add(scA, A, scB, B, C, incC);
    }
#endif
   C->tsem->unlock();
}
```

Figure 5.2. *eddl/src/tensor/tensor_math.cpp.*

File tensor_hls_op.cpp describes all the functions which are needed to initialize the device, the OpenCL context and CommandQueue, determine the bitstream path and select the active kernel on it. In addition, you can also find all host programs related to the file tensor functions of the previous figure. The bitstream generation method was described in the deliverable 5.1. The programming flow is based on creating the error and event variables, sending the parameters in the correct order to the kernel and launching the kernel task.

```
void fpga_init(){ // initialize only once

   cl_int err;
   std::string binaryFile =
"/home/user/Deephealth/eddl/src/hardware/fpga/kernels/xclbin/eddl_train_batch.hw.xilinx_u200_xdma_201830_2.xclbin";
   unsigned fileBufSize;
   std::vector<cl::Device> devices = xcl::get_xil_devices();
   cl::Device device = devices[0];
   OCL_CHECK(err, context = cl::Context(device, NULL, NULL, NULL, &err));
            OCL_CHECK(err, q = cl::CommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE |
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &err));
   char *fileBuf = xcl::read_binary_file(binaryFile, fileBufSize);
   cl::Program::Binaries bins{{fileBuf, fileBufSize}};

   devices.resize(1);
   OCL_CHECK(err, program = cl::Program(context, devices, bins, NULL, &err));
   OCL_CHECK(err, multitensor_op = cl::Kernel(program,"multitensor_op", &err));
   OCL_CHECK(err, kernel_add = cl::Kernel(program,"kernel_add", &err));
   OCL_CHECK(err, sum2D_rowwise = cl::Kernel(program,"kernel_sum2D_rowwise", &err));
   OCL_CHECK(err, kernel_cent = cl::Kernel(program,"kernel_cent", &err));
   OCL_CHECK(err, relu_soft_d = cl::Kernel(program,"relu_soft_d", &err));
   OCL_CHECK(err, reduce_sum2D = cl::Kernel(program,"reduce_sum2D", &err));
   OCL_CHECK(err, kernel_core = cl::Kernel(program,"kernel_core", &err));
   OCL_CHECK(err, kernel_accuracy = cl::Kernel(program,"kernel_accuracy", &err));
   OCL_CHECK(err, kernel_total_sum = cl::Kernel(program,"kernel_total_sum", &err));
   if (err != CL_SUCCESS) printf("Error creating kernel\n");

}

void fpga_tensor_add(float scA,Tensor *A, float scB,Tensor *B, Tensor *C,int incC){
   cl_int err;
   cl::Event event;
   #ifdef DBG_FPGA
      printf("FPGA::add\n");
   #endif

   OCL_CHECK(err, err = kernel_add.setArg(0, scA));
   OCL_CHECK(err, err = kernel_add.setArg(1, (A->fpga_ptr)));
   OCL_CHECK(err, err = kernel_add.setArg(2, scB));
   OCL_CHECK(err, err = kernel_add.setArg(3, (B->fpga_ptr)));
   OCL_CHECK(err, err = kernel_add.setArg(4, (C->fpga_ptr)));
   OCL_CHECK(err, err = kernel_add.setArg(5, incC));
   OCL_CHECK(err, err = kernel_add.setArg(6, A->size));

   OCL_CHECK(err, err = q.enqueueTask(kernel_add, NULL, &event));
   event.wait();
}
```

HOST PROGRAM

***Figure 5.3**: eddl/src/hardware/fpga/tensor_hls_op.cpp*

The SDAccel environment supports kernels expressed in OpenCL C, C/C++, and RTL (SystemVerilog, Verilog, or VHDL). You can use different kernel types in the same application. However, each kernel has specific requirements and coding styles that should be used.

The communication interface with the kernel is based on the Xilinx AXI protocol. There are three types of AXI4 interfaces:

- AXI4—for high-performance memory-mapped requirements.
- AXI4-Lite—for simple, low-throughput memory-mapped communication AXI4-Stream—for high-speed streaming data.

All kernels require the following:

- A single slave AXI4-Lite interface used to access control registers (to pass scalar arguments and to start/stop the kernel)
- At least one of the following interfaces (can have both interfaces):
    - AXI4 master interface to communicate with global memory.
    - AXI4-Stream interface for transferring data between kernels or directly with the host.

Figure 5.4 shows an example of the kernel used for the tensor add function. It should be noted that the kernel shown is not optimized. In this case, there are five scalar arguments, three start/stop port control for three data streams and one port “gmem” to receive all values of A, B and C tensors.

```
extern "C" {

void kernel_add(
        float scA,
        const float *A,
        float scB,
        const float *B,
        float *C,
        int incC,
        int tam
       )
{

#pragma HLS INTERFACE m_axi port=A offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=B offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=C offset=slave bundle=gmem
#pragma HLS INTERFACE s_axilite port=A  bundle=control
#pragma HLS INTERFACE s_axilite port=B  bundle=control
#pragma HLS INTERFACE s_axilite port=C  bundle=control
#pragma HLS INTERFACE s_axilite port=scA bundle=control
#pragma HLS INTERFACE s_axilite port=scB bundle=control
#pragma HLS INTERFACE s_axilite port=incC bundle=control
#pragma HLS INTERFACE s_axilite port=tam bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control

for(int i=0;i<tam;i++){
    if (incC) C[i]+=scA*A[i]+scB*B[i];
    else C[i]=scA*A[i]+scB*B[i];
}

}
}
```

***Figure 5.4**: eddl/src/hardware/fpga/kernels/kernel_add.cpp*

5.1.1 FPGA kernel design flow

With the availability of our testbed platform and the specifications of the EDDL/ECVL libraries, we defined the following design flow for each target kernel for FPGAs:

1. Identification of the kernel, which depends on the neural network and the use case
2. Memory requirements profiling of the kernel as a function of the batch size and the dataset size
3. Execution profiling (and instantiation frequency) of the kernel on different target devices, checking its criticality in the complete pipeline design of the training process
4. Functional implementation of the kernel on FPGA in software emulation mode, checking its arguments and dimensioning of the algorithm. The kernel is embedded into the testbed platform to check whether the network using the kernel still converges (kernel functional validation)
5. Hardware implementation of the kernel with hardware emulation. The kernel is implemented in hardware but tested in emulated form (for rapid exploration of validity of the implementation). Resources needed on the FPGA are evaluated.
6. Actual hardware implementation and testing on the FPGA. The platform is hooked to the FPGA and the complete training process uses the kernel running on the FPGA. Convergence of the training process is checked.
7. Area, performance profiling within the platform and potential optimization, looping with previous two steps (hardware emulation and hardware implementation).
8. Final adaptation to EDDLL and ECVL libraries. This process implies argument's adaptation using the same programming models and tools (OpenCL and OpenCV).
9. Final validation on EDDLL and ECVL libraries.

5.1.2 FPGA emulation mode

One important aspect when designing multiple kernels is that they cannot be optimized or tested at once. Indeed, it is quite complex to get a kernel validated in isolation as the implications on the training process (with multiple instances of many other kernels being executed) may be unexpected. For this reason, we have developed an emulated mode for the kernels which allows us to embed a kernel supposedly running on the FPGAs to actually being run on the CPU in an emulated mode. All the tensors are located on the FPGA but the kernel is executed on the CPU. Therefore, the tensors are firstly copied back to CPU memory, then the kernel implemented on CPU is run and after that the output tensors are copied back to the FPGA memory.

With this mode we can easily code most of the kernels in emulated mode and focus on a specific kernel optimization and test it using the complete training process, thus not waiting to have all the kernels implemented on FPGA. Figure 5.5 shows the case for one such kernel in emulated mode in the testbed platform.

Indeed, the emulated mode allows us also to determine which kernels do not deserve being implemented on FPGAs and therefore, being launched on CPU. This is the case, for instance, of the initialization function kernel which only takes place once during the complete training process and therefore, make no sense to waste FPGA resources for its implementation as the impact on execution time is negligible. With this emulated mode we allow the EDDLL and ECVL libraries to support those optimizations without needing any modification.

```
int fn_vect_max_opencl_fpga(cl_mem a, cl_mem b, cl_mem c, int n) {
    PROFILING_HEADER_EXTERN(vect_max);
    type *a_cpu = malloc(sizeof(type) * n);
    type *b_cpu = malloc(sizeof(type) * n);
    type *c_cpu = malloc(sizeof(type) * n);
    fn_read_buffer_opencl_fpga(a, a_cpu, n * sizeof(type));
    fn_read_buffer_opencl_fpga(b, b_cpu, n * sizeof(type));
    fn_vect_max_cpu(a_cpu, b_cpu, c_cpu, n);
    fn_write_buffer_opencl_fpga(c_cpu, c, n * sizeof(type));
    free(a_cpu);
    free(b_cpu);
    free(c_cpu);
    PROFILING_DEVICE(vect_max, DEV_OPENCL_FPGA);
    return 1;
}
```

***Figure 5.5**: Example of FPGA kernel call in the Testbed platform*

5.1.3 Data quantization, reduced precision, and kernel fusion

One critical aspect with FPGA designs is that they do not natively support FPU units or their support is limited. Indeed, FPGAs are well suited for fixed point integer operations or even lower integer formats. The EDDLL and ECVL libraries are not, in principle, designed for such support. Indeed, it is not the goal of the libraries (as they target the training process). However, within the project, we will explore the support at least on inference processes and with specific well-defined scenarios. The testbed platform allows us to decouple our needs from the expected development of the EDDL/ECVL libraries within the project. Indeed, the platform already analyses the impact of precision reduction on the generated models on the accuracy obtained.

Additionally, the platform allows us to explore multi-function kernels or combined kernels which can be proven to be efficiently implemented on FPGAs. One case is the combination of the matrix multiplication operation and the matrix addition operation, both are used in dense layers and run in sequential mode to multiply activations with weights and then to add bias to the output. We can explore designs with a unified kernel which performs both actions. With the testbed we can explore these venues without impacting unnecessarily on the design process of EDDLL and ECVL. Notice that kernel fusion allows also to reduce memory pressure since temporal tensors are avoided.

5.2 Functional hardware support for a complete training in FPGA

We have implemented all the kernels required to execute the training batch example provided in the EDDLL. This example uses an input layer with 784 size, three Dense layers with “ReLu” activation and 1024 size. Finally, the network has a dense layer with 10 classes and “Softmax” activation. The model is built for “sdg” optimizer, “soft_cross_entropy” losses and “categorical_accuracy” for metrics. The dataset has one epoch and 100 images for batch size. Table 5.1 lists the kernels we have implemented to run the eddl_train_batch example in FPGA. Functions not supported for FPGA are executed on CPU. Each function is associated with a specific kernel.

FUNCTION	ASSOCIATED KERNEL
sum2D_rowwise	kernel_sum2D_rowwise.cpp
reduce_sum2D	reduce_sum2D.cpp
add	kernel_add.cpp

sum	kernel_total_sum.cpp
ReLu	multitensor_op.cpp
D_Relu	relu_soft_d.cpp
Softmax	multitensor_op.cpp
fill_	kernel_core.cpp
accuracy	kernel_accuracy.cpp
cent	kernel_cent.cpp

Table 5.1 *Functions and associated kernels.*

Table 5.2 shows the resources required by each kernel. For the FPGA implementation we use the ALVEO U200 board and the design flow described in Deliverable D5.1.

KERNEL	FF	LUT	DSP	BRAM	URAM
kernel_sum2D_rowwise	2705	2167	6	4	0
reduce_sum2D	332	552	0	0	0
kernel_add	2755	2105	5	2	0
kernel_total_sum	1867	1965	2	4	0
multitensor_op	5010	5438	28	2	0
relu_soft_d	4005	2996	11	10	0
kernel_core	904	1116	0	2	0
kernel_accuracy	2304	2349	0	2	0
kernel_cent	9465	4651	71	4	0

Table 5.2 *FPGA reources for each kernel.*

The bitstream includes all kernels and consumes only 2% DPS blocks, 16% LUTs, 12% FF and 22% RAMs. This means that there is a significant room for improvement since the available resources can be used to increase the data-level parallelism. Figure 5.6 shows how each kernel is interconnected within the FPGA using the available logic and memories. Figure 5.7 depicts how the resources are allocated to specific FPGA regions. Note that in the ALVEO U200 we have different logic regions (e.g slr0 and slr1 depicted in the plot). Each region has access to a different bank of the DDR memory. Thus, using several logical regions is useful to improve the memory bandwidth.

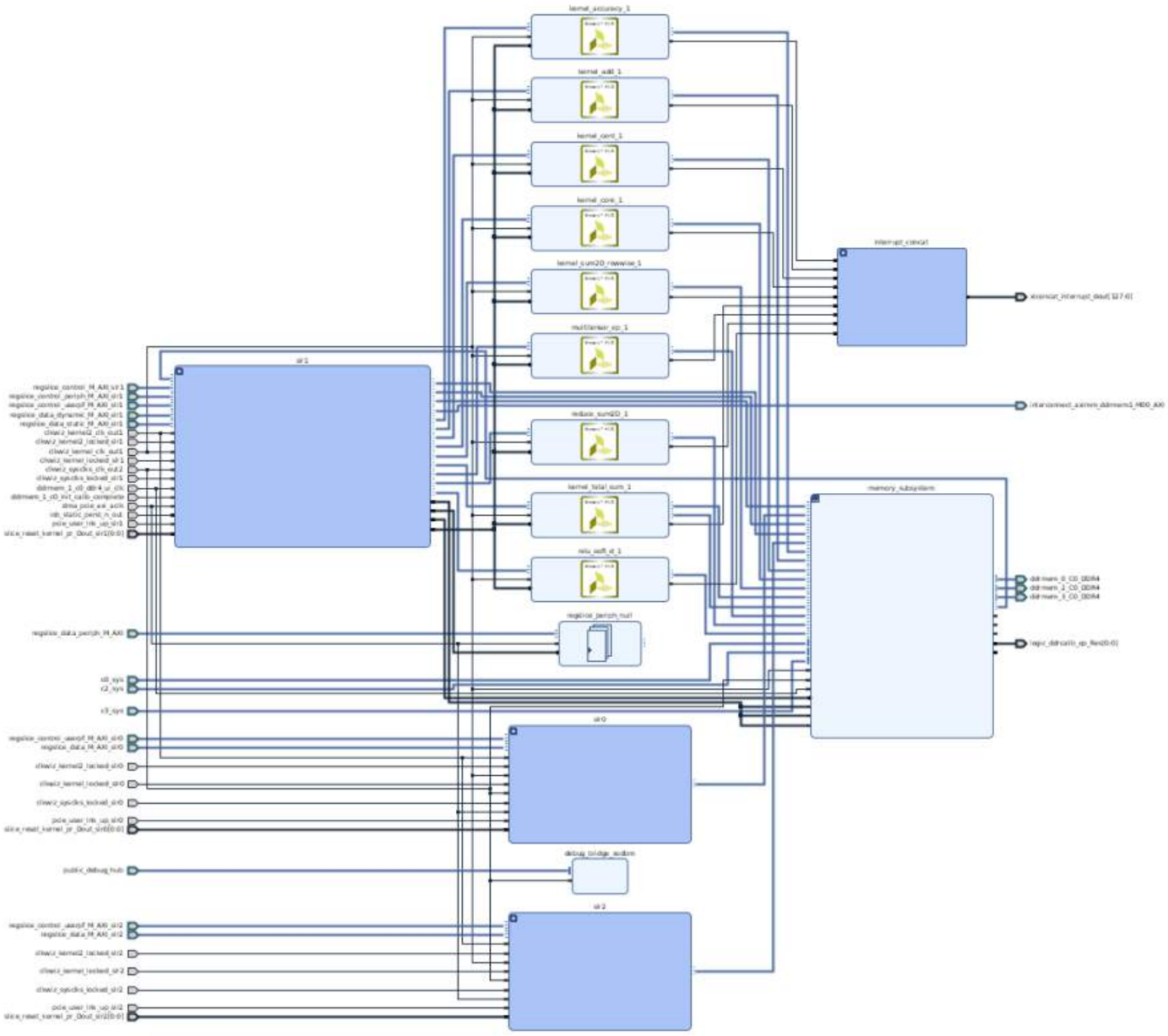

Figure 5.6. *Block diagram eddl_train_batch bitstream.*

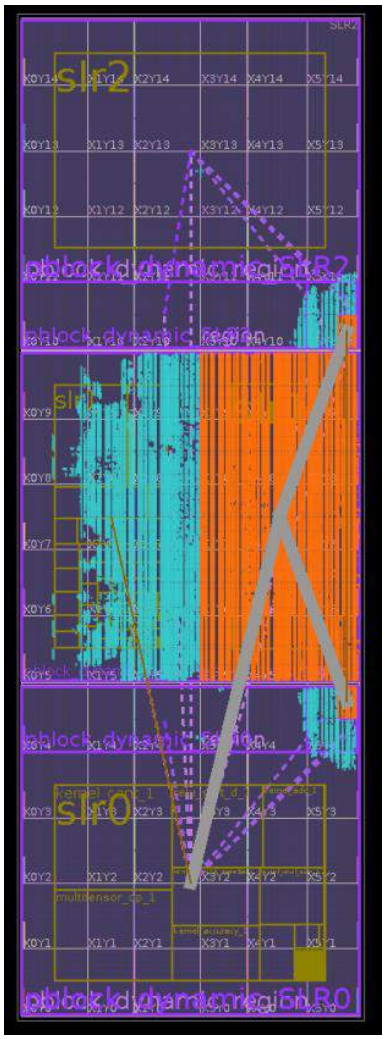

***Figure 5.7**. Layout for eddl_train_batch bitstream in Alveo U200.*

After generating the bitstream and compiling the example for the FPGA device, the network training is launched, achieving the results shown in Table 5.3. As shown in this table the FPGA implementation achieves results that are very similar to the ones provided by the CPU. Differences can be explained by the inherent randomness of the training process and by the different treatment of floating point operations in CPU and FPGA.

DEVICE	SOFT_CROSS_ENTROPY.	ACCURACY
CPU	0.379	0.934
FPGA	0.367	0.930

***Table 5.3**: Accuracy results CPU/FPGA.*

Finally, we show the speed comparison between function calls for CPU and for FPGA. The default FPGA implementation provides significantly worse results that the CPU. However, this is expected since this FPGA implementation is the result of blindly applying HLS to the CPU kernels code. As we show later when FPGA kernels are optimized we can improve the performance by several orders of magnitude.

FUNCTION	CPU [s/call]	FPGA [s/call]
sum2D_rowwise	4,62E-05	0,0127531
reduce_sum2D	2,62E-05	1,64E-04
add	0,000231415	0,0358812
sum	1,12E-06	3,59E-04
ReLu	4,85E-05	5,35E-04
D_ReLu	0,000106174	0,0164061
Softmax	1,10E-05	2,38E-04
fill_	5,30E-04	9,70E-04
accuracy	4,19E-06	0,000411316
cent	9,04E-05	0,0033001

***Table 5.4**: Results comparison in EDDLL train batch CPU/FPGA.*

Figure 5.8 shows the comparison of the time required by CPU and FPGA kernels exercised in the basic train batch example. Note that we have not included the matrix multiplication kernel since the non-optimized version of this kernel in the FPGA lasts too much. In this experiment all matmult operations are performed in the CPU which requires for the case of the FPGA to move tensors between devices. The goal of this exercise was twofold. First, we want to show that the EDDLL functionality in the FPGA is correct. Second, we want to quantify what other kernels, in addition to the matmults, will need to be optimized not to penalize training process execution times in the FPGA.

As we can observe in Figure 5.8 sum2D_rowise, add, and D_ReLu kernels do not perform well when they are not optimized while for the rest of the kernels the differences between CPU and non-optimized FPGAs are negligible.

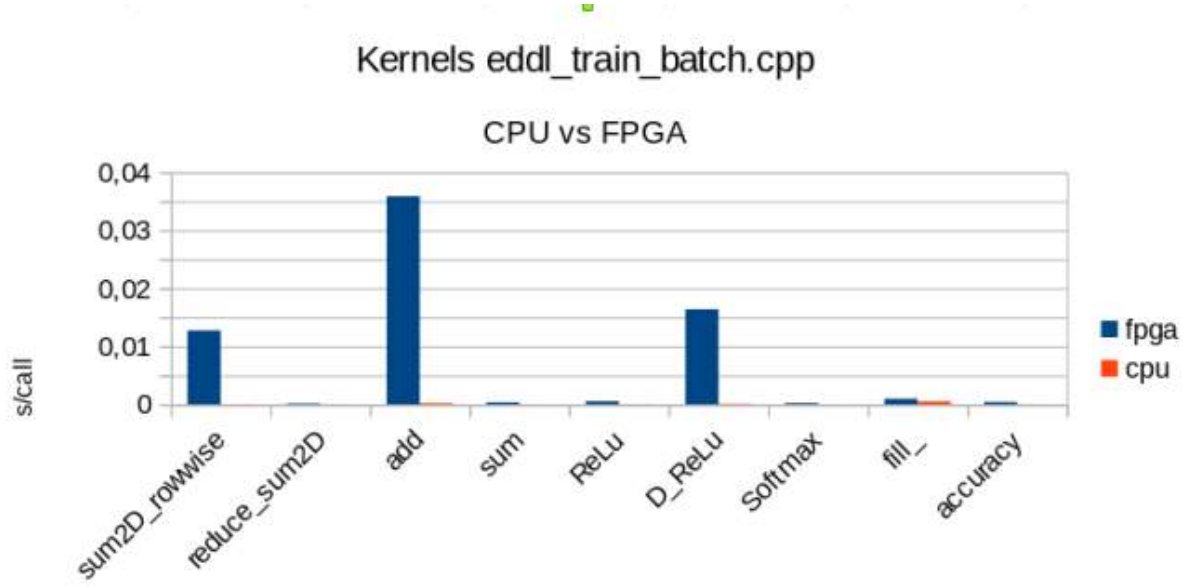

***Figure 5.8**: Results comparison in EDDLL train batch CPU/FPGA.*

5.3 Profiling of training process

To get a full picture of the time-consuming kernels in other several neural networks models, we have profiled the complete function set implemented for CPUs in the EDDLL. Using this profiling extension we ran the following available examples of the EDDLL library:

- CIFAR dataset with convolutional network
- MNIST dataset with autoencoder
- CIFAR dataset with ResNet network
- CIFAR dataset with VGG16 network
- CIFAR VGG16 network with group normalization
- Segmentation process (drive database)
- MNIST dataset with Recurrent neural network

Notice that complexities vary between the chosen examples.

Figure 5.9 shows the breakdown of functions for each example, showing the involved time for each function. Notice we show accumulated running time for each function.

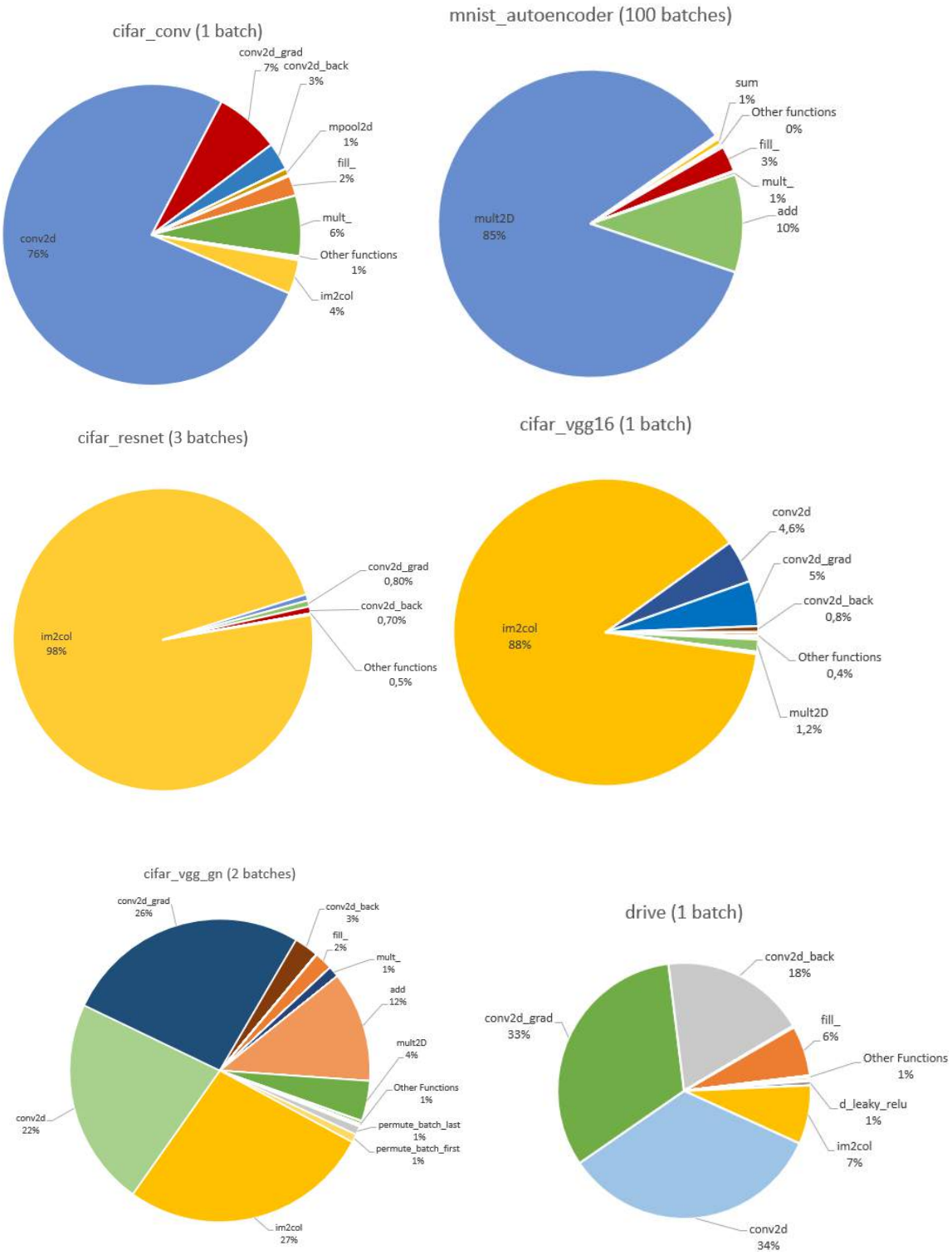

***Figure 5.9**: Execution time per kernel breakdown for several network models*

We can see for all the cases, mainly, two important conclusions. First, some functions are seldom used and therefore have a negligible impact on the execution time in the training process. This is the case of functions rand_signed_uniform and rand_normal. Indeed, all the initialization functions are called only once per training process and therefore will always have no impact. We can consider that when targeting FPGAs those initialization processes will be run on the CPU and the initialized tensors will then be transferred to the FPGA.

The second observation is the significant impact of the convolutional process and the matrix multiplication functions. Indeed, within the convolutional functions profiled there are embedded further matrix multiplications. Functions such as im2col, conv2d, conv2d_grad, and conv2d_back, need an efficient implementation on FPGAs. The same applies for mult2D.

Other types of neural networks use batch normalization. This is a heavy process as shown in the Figure 5.9. Indeed, the batch normalization process is distributed over many functions. Some functions perform permutations to the input activations (permute_channels_last, permute_channels_first, permute_batch_last, permute_batch_first) and others perform the arithmetic operations needed (add_, mult_, sqr_, sqrt_, el_div, el_mult).

Other aspects we can see from the profiling is that maxpooling (mpool2d, mpoool2d_back) and average pooling functions do not have a significant impact on the execution time. Also, some functions in EDDLL are used for data augmentation (e.g. single_crop, single_crop_scale, crop, crop_scale_random). Those functions are expected to be replaced by the ECVL library support for which we are already providing FPGA support.

Finally, activation functions (relu, d_relu, sigmoid, d_sigmoid) have a lower impact on performance but not negligible. Those functions will need to be implemented on FPGAs. Notice that the most frequently used activation function is RELU.

Knowing the profiling of functions for a particular training and/or inference process is important when targeting FPGAs. In particular, knowing which functions are called (and which ones not) enables efficient implementation on the FPGA device. Indeed, not all the functions implemented in EDDLL for the CPU can be fit on a single FPGA at the same time. Therefore, it is needed to know in advance the set of functions that are needed for a specific training or inference process. With this information the FPGA can be programmed and configured with the specific set of functions needed.

Moreover, in the case of a complex training and/or inference process where too many functions are needed at the same time, we need to carefully select which functions will be supported in FPGA and which ones will be offloaded to the CPU (or GPU). This will be a needed tradeoff in case of complex designs.

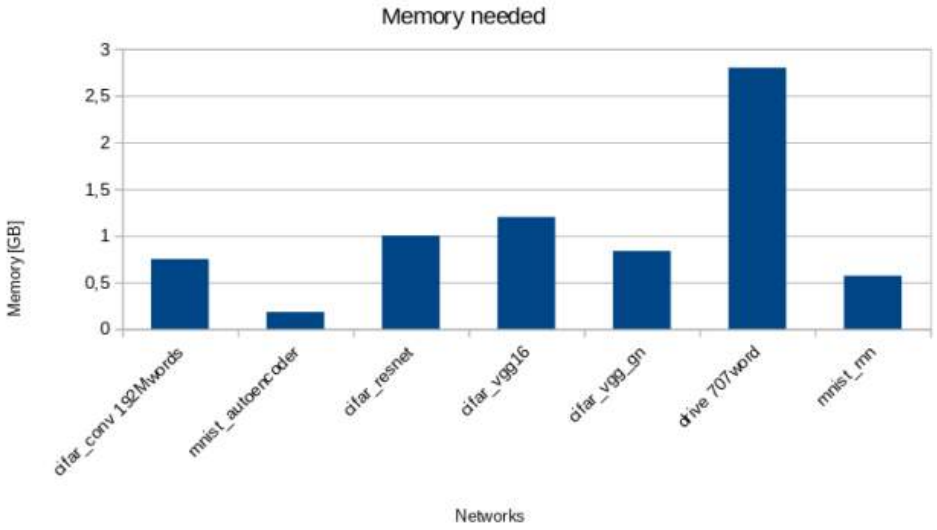

***Figure 5.10**: Memory requirements of different neural network examples*

Another important aspect when targeting FPGAs is the required amount of memory needed. FPGAs, unfortunately, have a limited memory space and may compromise their performance and effectiveness. Therefore, we need to argument memory requirements and memory bounds for our designs. We have profiled the previous examples with the required memory instantiated during the training process. Figure 5.10 shows the buffer requirements for each example. As can be seen, the most complex example in terms of memory requirements is drive, which requires 2.8GB. This may

limit the applicability of FPGA kernels as tensors may not all fit on the FPGA. Indeed, the FPGA will need to store most of the tensors off-chip (in its associated DDR), thus potentially impacting performance.

Notice also that the memory requirements depend on the neural network design but mostly on the batch size. For the drive example, which took the highest memory needs, the batch size was set to 2 items only. If we double the batch size, the memory requirements increase to 4,5GB, which may challenge FPGA design. If due to memory limitations the model cannot be completely trained and/or inferred on the FPGA we would need both FPGA and CPU to cooperate.

5.4 FPGA kernels optimizations

After the identification of the most relevant kernels it is time to optimize their execution in the FPGA. In this section we show and compare the performance of two highly optimized implementations of matrix multiplications in FPGA. In particular, we have ported GEMX to the EDDLL, an efficient implementation of matrix multiplications and related functionalities provided by Xilinx, and GEMM_HLS an open-source implementation of matrix operations developed by ETH Zurich (https://github.com/spcl/gemm_hls) to test its functionality.

The first thing we have observed is that both implementations impose certain limitations to the size of the matrix taken as input. Thus, to use such implementations tensors will have to be padded with zeros to support these FPGA overlays without imposing limitations to the sizes of the input data.

We have executed both overlays in the ALVEO U200 FPGA board for several matrix sizes. Figure 5.11 shows a comparison of the execution time. As shown in the plot both overlays produce very similar results. However, GEMX is able to achieve higher peak performance numbers (lower execution times). In particular, we achieve a peak performance of 127,22 and 210,3 GFLOPS for GEMM_HLS and GEMX, respectively. Note that this is up to 200X faster than the non-optimized version of the matrix multiplication. For the EDDLL running on our server (i7-7800X CPU @ 3.50GHz) we get 138.85 GFLOPS when using multiple threads. Note that in the case of the FPGA we can also host more than one GEMX and GEMM_HLS overlay allowing to achieve higher performance.

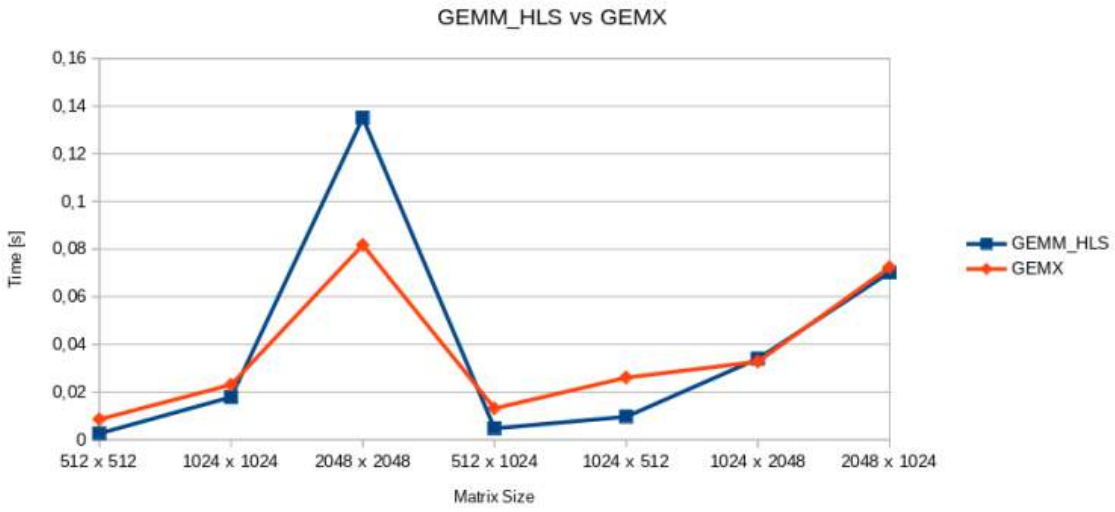

***Figure 5.11**: GEMX and GEMM_HLS performance comparison for different matrix sizes*

6 FPGA dataflow accelerator for efficient inference

In order to improve the inference performance of deep learning applications, HPC infrastructures will rely on FPGA based boards as hardware accelerators. The challenge is to take advantage of the acceleration brought by FPGA boards without increasing in a prohibitive way the time-to-model-in-production (TTMIP).

The objective of this flow is to easily generate a hardware implementation of the inference code of a Deep Neural Network (DNN). This code will run efficiently on a FPGA device. The proposed flow intends to be as automatized as possible in order to reduce the need for Neural Network experts.

The flow will rely on the following artefacts:

- The EDDLL library, mainly used for DNN design and training.
- The DNeuro CEA accelerator.
- The N2D2 CEA framework for inference code generation using the DNeuro accelerator.

Interworking between the EDDLL and the N2D2 framework will rely on the ONNX exchange format.

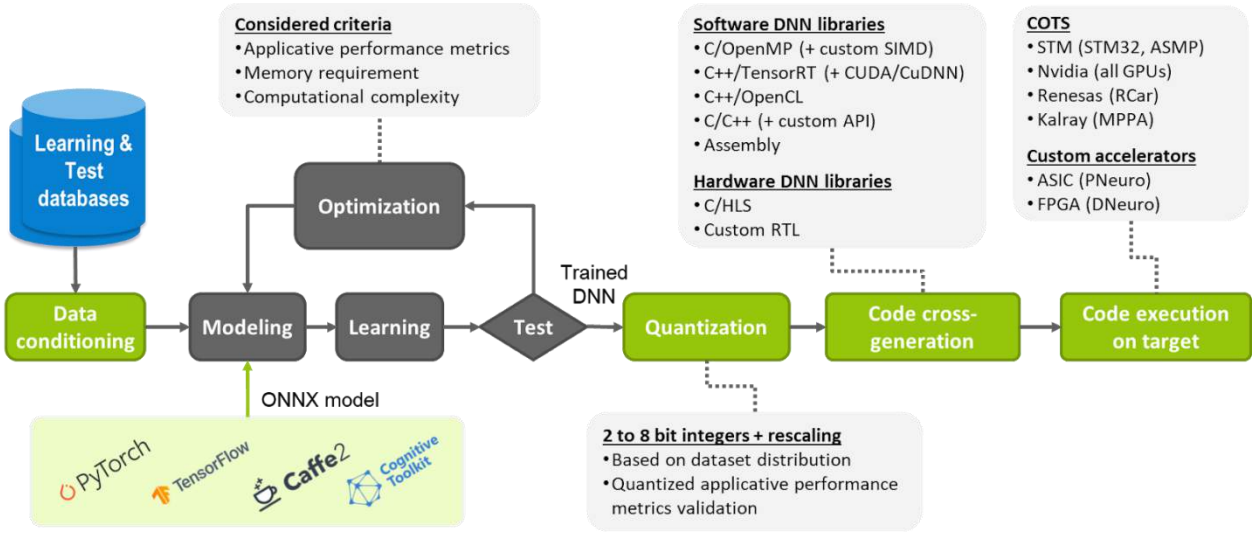

***Figure 6.1**. CEA N2D2 framework overview for inference code generation.*

6.1 Accelerator DNeuro

The aim of the N2D2/DNeuro flow is, for a given Neural Network, to automatically generate an inference IP that will be directly loaded and executed on a targeted FPGA. The IP is built using the N2D2 framework by assembling optimized pre-designed IP blocks corresponding to various NN computation layers.

6.1.1 Inference IP generation flow

The N2D2 CEA framework will be responsible to generate the DNeuro-based inference IP of a given Neural Network. The IP generation flow is as follows:

1. Loading of a pre-trained ONNX model;
2. Accuracy validation of the ONNX model in inference (Inference);
3. Calibration and quantization of the DNN (Calibration);
4. Accuracy validation of the quantized model in inference (Quantized inference);

5. Generation of the top-level RTL instantiating the DNeuro HW blocs and generation of test vectors from the DNN dataset (DNeuro export);
6. Accuracy validation of the DNN implemented with DNeuro HW using a bit-accurate emulator (DNeuro emulator);
7. Neural network performance tuning.

The full generation flow process with the above steps is summarized in Figure 6.2.

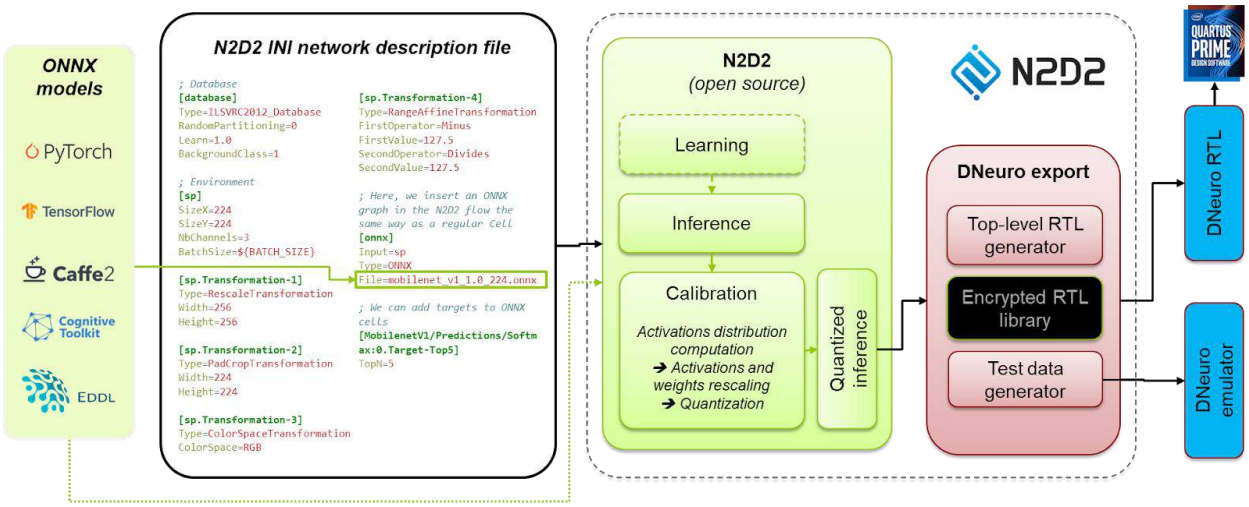

***Figure 6.2**. N2D2 / DNeuro generation flow.*

6.1.2 Calibration and quantization of the DNN

The N2D2 framework will perform a post-training 8-bits quantization of the input Neural Network. The quantization uses a calibration process to determine the optimal range of activations in the different layers of the network, based on samples from the dataset.

6.1.3 DNeuro export: HW blocs library building

The N2D2 framework will build a library of all the blocs (layers) required by the input Neural Network. The library is a subset of the collection of optimized pre-designed HW blocs that can be linked one to another. The DNeuro accelerator mainly targets Convolutional Neural Networks (CNNs). Figure 6.3 shows a list of the HW blocs on which the accelerator can rely on.

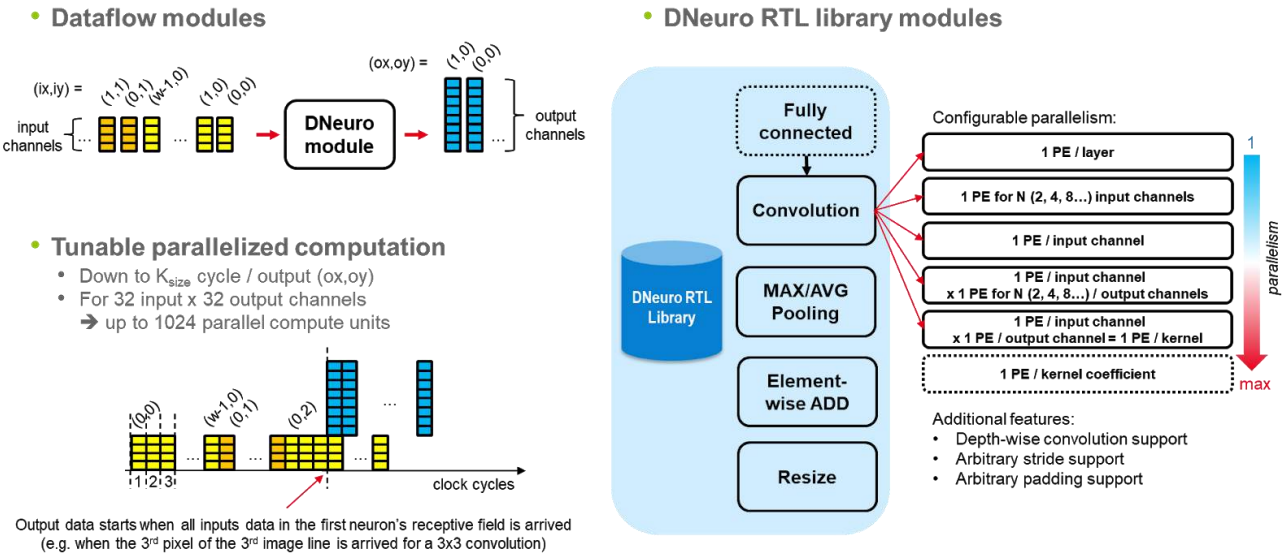

***Figure 6.3**. DNeuro working principle and RTL library modules summarized.*

In Figure 6.4 we present the complete list of layers supported by DNeuro.

Layer type	Support	Module(s)	Comments
			Common
Dropout	n.a.		removed during export
Fc	✓		implemented with Conv during export
InnerProduct → see Fc			
Rbf	✗		
Transformation	✗		
			Convolutional Neural Network
BatchNorm	✓	n.a.	merged with Conv during export with -fuse option
Conv	✓	$Conv_g$	generic
		$Conv_{pc}$	optimized for partial connections
		$Conv_{ps}$	optimized for stride $\neq 1$
Concat → implicit for Conv/Deconv/Pool/Fc			
Deconv	planned		
EltWise	✓		*Sum* operation only
EltWise → see EltWise			
FMP	✗		
Flatten → implicit to Fc/Rbf			
LRN	✗		
Maxout → see Pool			
Padding	✓	n.a.	merged with Conv/Pool during export
Pool	✓		*Max* operation only
Resize	✓		*NearestNeighbor* mode only
Softmax	planned		integer approximation
SortLabel → see .Target*			
Unpool	✗		
Upscale → see Resize			
.Target*	✓		top-1 sorting

***Figure 6.4**. Complete list of layers supported by DNeuro.*

Likewise, Figure 6.5 shows all the activation functions supported.

Activation type	Support	Specificities
Linear	✓	saturated arithmetic
Logistic	✓	saturation approximation, configurable zero, up to two configurable thresholds
ReLU → see Rectifier		
bReLU → see Rectifier		
Rectifier	✓	saturated arithmetic (positive values)
Saturation	✓	
Softplus	✗	
Tanh	✗	

***Figure 6.5**. Activation functions supported*

6.1.4 DNeuro export: neural network dataflow building

Using the library of needed HW blocks, N2D2 flow will create the Neural Network dataflow that implements its inference. The output of this step is a “top RTL” that can be directly synthesized by FPGA tooling. The IP generated here corresponds to an implementation that requires the minimum amount of FPGA resources. Most of the NN topologies accepted here are those used in CNN. Below is a list of those topologies, on an Arria 10 target:

Arria 10	GX/SX 160	GX/SX 220	GX/SX 270	GX/SX 320	GX/SX 480	GX/SX 570	GX/SX 660	GX 900	GX 1150
M20K (MB)	1.12	1.37	1.87	2.12	3.5	4.37	5.25	5.87	6.62
DSP	156	191	830	985	1,368	1,523	1,688	1,518	1,518
Mult. (MAC/c.)	312	382	1,660	1,970	2,736	3,046	3,376	3,036	3,036
MobileNet_v1_0.25	●	●	●	●	●	●	●	●	●
MobileNet_v1_0.5			●	●	●	●	●	●	●
MobileNet_v1_0.75			●		●	●	●	●	●
MobileNet_v1_1.0			●				●	●	●
SqueezeNet_v1.0			●	●	●	●	●	●	●
SqueezeNet_v1.1			●	●	●	●	●	●	●
MobileNet_v2_0.35				●	●	●	●	●	●
MobileNet_v2_0.5					●	●	●	●	●
MobileNet_v2_0.75					●	●	●	●	●
MobileNet_v2_1.0						●	●	●	●
MobileNet_v2_1.3								●	●
MobileNet_v2_1.4									●
AlexNet			●	●	●	●	●	●	●
VGG-16						●	● ○	● ○	● ○
GoogLeNet							●	●	●
ResNet-18							●	●	●
ResNet-34								●	●
ResNet-50									○

Legend:
- ● should be OK for the standard 224x224 input, but depends on the resolution;
- ● M20K memory may be insuffisent depending on the resolution;
- ● there is a better equivalent neural network (see the corresponding arrow);
- ○ using an alternative neural network is possible with a small accuracy loss.

***Table 6.1**. Arria 10 neural networks compatibility table with DNeuro v2, in terms of memory requirements*

.

6.2 Neural network performance tuning

This is the last step of the flow. Based on the Neural Network size and the FPGA capabilities, this step will try to increase the throughput and decrease the latency of the generated Neural Network by adding computation parallelism at the layer level. Knowing that a Neural Network processes received data through a pipeline represented by its layers, the overall system performance is constrained in two ways:

- Latency represented by the sum of:
  - The time to acquire the required data needed to start computation,
  - The time for data to go through all the Neural Network layers in sequence.
- Throughput: inverse of the time spent in the slowest layer.

Performance increase will be achieved by adding parallelism at the layer level. The N2D2 framework will perform the following actions:

- Scan the graph of the Neural Network dataflow in order to identify the slowest layer;
- Determine the amount of FPGA resources needed to increase the parallelism of the layer by one factor;
- If there are enough FPGA resources, it performs the transformation of the HW block to increase its parallelism by one factor.

The process is repeated until either there are no enough remaining FPGA resources to increase the parallelism by one factor or, the maximum parallelism has been reached. Each layer receives several input channels, applies on them a computation kernel and generates data on a certain

amount of output channels. The HW block representing the layer that has been identified as a candidate for parallelization will be transformed in order to:

- Simultaneously acquire data on several input channels,
- Perform in parallel computations on independent input channels,
- Simultaneously produce data on several output channels.

Doing so will reduce the overhead time to get data available for computation and reduces the delay needed to make data available to the subsequent layers. Moreover, the level of parallelism and thus the FPGA resource usage can be configured in order to find the best trade-off between latency and resource usage, as shown in Figure 6.6.

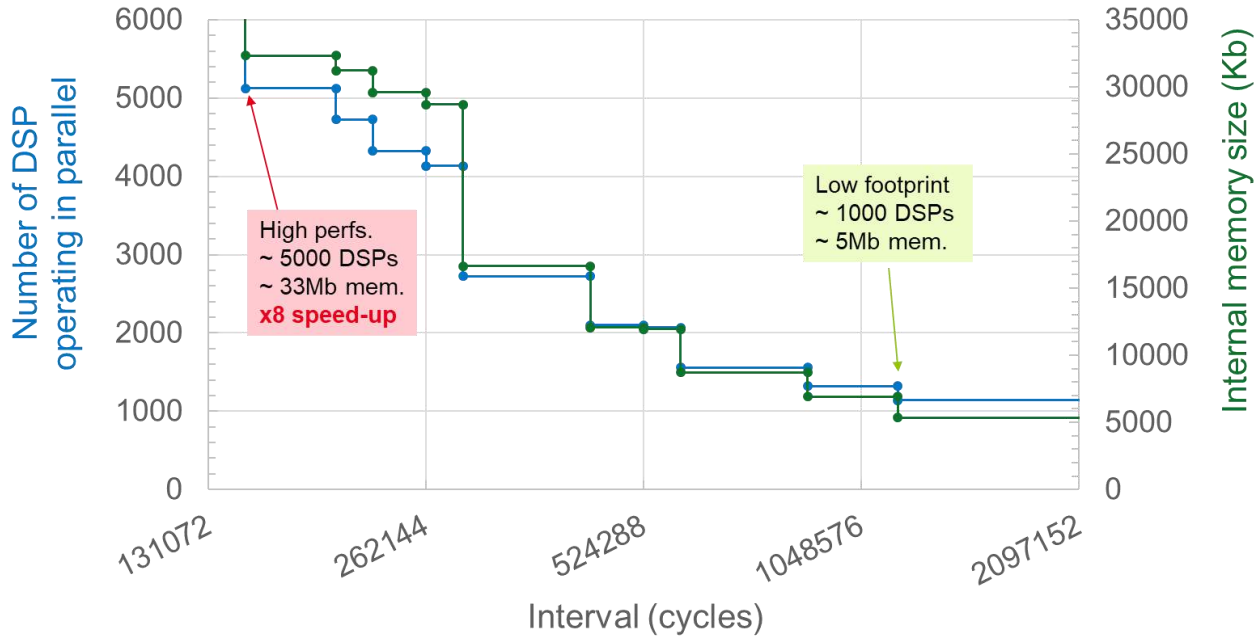

***Figure 6.6**. DSP and memory usage can be automatically adjusted to achieve the desired latency.*

6.3 Neural network topology adaptation and validation

The proposed generic dataflow accelerator relies on a collection of optimized hardware blocks (computational layers). Therefore, a Neural Network that has been designed without this generic dataflow accelerator in mind can make use of kernels that are not directly available. Adding the support of the missing layer by the generic accelerator may result in a significant design effort, incompatible with the TTMIP. A less costly approach consists in reworking the Neural Network architecture in order to use an alternative to the missing layer. Such a rework will require retraining the network. Standard topologies like MobileNet, ResNet or VGG will be supported, albeit without the Softmax layer, which is not required for classification during inference and can be replaced by a simple MAX operation.

The proposed flow will be first validated on known Neural Networks, available as examples with the EDDLL like the EDDLL MNIST example and MobileNet and/or ResNet topologies. Then the flow will be checked against one of the Neural Network designed within the project. Even if the definitive use-case has not been selected yet it appears that the “Skin cancer melanoma detection” (UC12) could be a good candidate.

6.4 Perspective: HW acceleration of huge neural network

The proposed generic dataflow accelerator allows the generation of the inference code of a Neural Network for a FPGA target. The execution efficiency of the code is directly linked to the level of parallelism achieved during the IP process generation. However, the level of achievable parallelism is constrained by the size of the Neural Network and the FPGA capabilities. A way to allow a huge Neural Networks to get benefit of the generic dataflow acceleration is to split the generated IP over several FPGA devices connected together through high-speed links. The general process CEA intends to set up is as follows:

- Completely unfold the Neural Network IP, exposing the theoretical maximum demand of FPGA resources on a per layer basis or not.
- Reduce the parallelism to an acceptable level, based on a maximum amount of FPGA resources available, on a per-layer basis, or if possible, in a more flexible way.
- Perform a logic partitioning of the network among the different available FPGAs on the target platform.
- Implement the result by adding the necessary communication resources between FPGAs.

The last two steps are currently under active research and development by the CEA team involved in WP5. This work involves mathematical modeling for the problem of neural network partitioning. As DNeuro provides full control on resource utilization and communication between layers, the partitioning model will allow partitioning by optimizing two possible criteria: the minimization of latency and the maximization of resource usage.

7 Distributed pyEDDLL on HPC infrastructures

This section describes the parallelization of the pyEDDLL training operation in the Marenostrum HPC infrastructure owned by the DeepHealth partner BSC. We refer the interested reader to check deliverables D2.1 “EDDLL library” (May 2020) for a complete description of the parallelization strategy for the pyEDDLL training operation, and D5.4 “The runtime system for DeepHealth libraries” (March 2020) for the modifications included in the COMPSs runtime to support the execution.

In the next section we briefly describe the parallel structure of the pyEDDLL training operation and its execution on the Marenostrum supercomputer, and provides a first evaluation from a performance and accuracy point of view.

7.1 Parallelization approach

The parallelism exposed by the pyEDDLL training operation is shown in Figure 7.1, in the form of a Task Dependency Graph (TDG). Each node of the TDG represents a COMPSs task that can be distributed among the different computing nodes, in this case, Marenostrum nodes. The edges of the TDG represent the data transfers and synchronization between tasks, defining an execution order.

The deep neural network model is built in parallel on each of the available distributed computing resources (build COMPSs tasks). As soon as the building process is completed, the training process starts. The parallelization strategy follows a synchronous training approach: each train_bach COMPSs task operates in parallel over a subset of the dataset (divided into a given number of batches). When all tasks complete, the computed (partial) weights are collected in the master node (update_gradients task). This process is repeated for a given number of epochs (see deliverable D2.1 “EDDLL library” for further details).

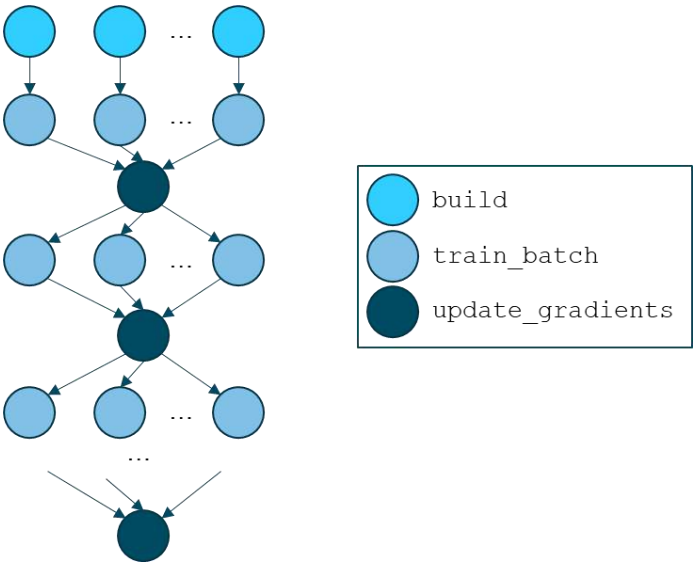

***Figure 7.1**. Task-level parallelism of the pyEDDLL training operations.*

7.2 Supporting Marenostrum with COMPSs

One of the main features of the COMPSs framework is that it abstracts the parallel execution model from the underlying distributed infrastructure. Hence, COMPSs programs do not include any detail that would tie them to a particular platform, boosting portability among diverse infrastructures and so enabling its execution in both a classical HPC environment and a cloud-based environment.

To do so, COMPSs abstracts the underlying infrastructure by creating a set of execution environments, named *COMPSs workers*, in which COMPSs tasks execute. Internally, the COMPSs runtime implements different adapters to support the execution of COMPSs tasks in a given resource. Through a set of configuration files, the user specifies the available computing resources, which may reside in a cluster or in the cloud.

The COMPSs runtime is already supported in the Marenostrum supercomputer as a loadable module, in which the COMPSs workers are executed in the different Marenostrum computing nodes, each equipped with 2 Intel Xeon Platinum 8160 CPU with 24 cores each @ 2.10GHz, 96 GB of main memory and 200 GB local SSD available as temporary storage during jobs. The COMPSs runtime is then responsible for distributing the parallel version of the pyEDDLL training operation as described above.

7.3 Preliminary evaluation

We have conducted a preliminary set of experiments in the Marenostrum supercomputer to evaluate the performance speedup of the distributed training operation when the trained accuracy is above 90%. Concretely, the experimental setup is the following:

- The MNIST training dataset (http://yann.lecun.com/exdb/mnist/) composed of 60000 images.
- A DNN topology with the following configuration: 784 x 1024 x 1024 x 1024 x 10, linear rectified activation function for hidden layers and softmax for output layer.
- The number of COMPSs Workers ranging from 1 (corresponding to sequential execution) to 32.
- The number of batches has been set up equal to the number of COMPSs workers.

Figure 7.2 shows the execution time (in seconds) and the performance speedup (Y-axis) of the distributed pyEDDLL training operation when ranging the number of COMPSs workers from 1 to 32

(X-axis). There is a nearly linear speed-up when the number of workers ranges from 2 to 8. However, this trend does not continue for 16 and 32 workers. This is mainly due to the tasks granularity, that decreases and does not compensate the high communications costs, nor the parameter aggregation that requires synchronization. The accuracy achieved in all experiments is 94.01%, independently of the distribution strategy.

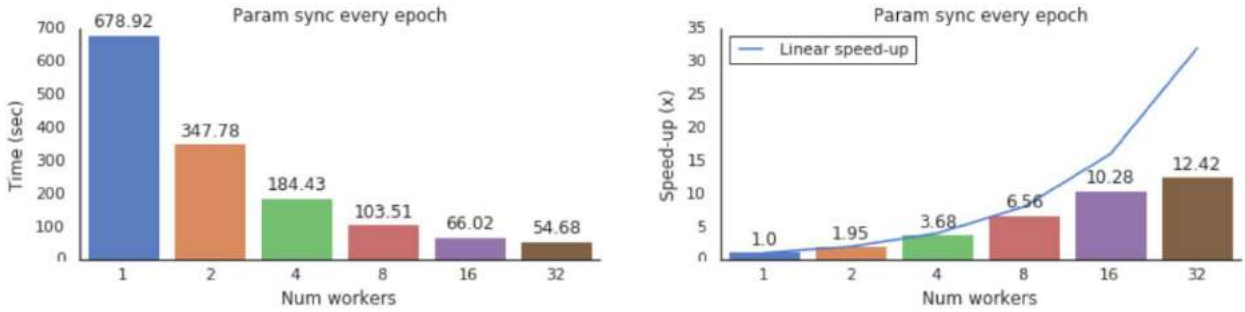

***Figure 7.2**. Preliminary evaluation of the distributed pyEDDLL training in Marenostrum.*

8 Conclusions and future work

This deliverable reports the activities performed so far until month M17 related to task T2.3 “EDDLL adaptation to heterogeneous HPC hardware”. The goal of this task is to deploy, analyze, and decide which are the most suitable strategies to use when porting the EDDLL to heterogeneous HPC platforms technologies, mainly focusing on CPUs, GPUs and FPGAs. The current deliverable will be updated in the beginning of the third year of the project.

Currently, most of the focus has been put on the development, testing, and characterization of EDDLL baseline. This way, much of the work has been put on developing a robust baseline and characterizing the performance of the library. This baseline version has been tested and profiled on different heterogeneous systems. As a result, the main bottlenecks of the EDDLL have been identified, along with the underlying causes of the performance loss and low exploitation of the available hardware resources.

For the CPU implementation, we have proposed and implemented algorithm optimizations that lead to significant improvements of the library performance (i.e. 47.3x compared to the EDDLL baseline; only 1.7x slower than the executions on the V100 GPU). At the same time, we provide a comprehensive characterization of the performance of the CPU-Optimized version of different HPC platforms. As a byproduct of this analysis, we have identified new bottlenecks and inefficiencies that will be addressed in the future in this task effort.

The profiling in terms of performance, accuracy, and power for the algorithms already adapted to GPUs is an on-going effort. In addition to the results presented in this deliverable, additional GPU performance profiles (decoupled from the EDDLL) are being performed, as well as profiling executions for the GPU version of the EDDLL.

Likewise, the adaptation of the main EEDLL algorithms to FPGAs is also an on-going effort. So far, the FPGA components to be ported have been successfully identified, and initial implementations of some the kernels have been presented. In the next few months we will focus our attention on the FPGA-based porting of the EDDLL targeting different platforms.

Another focus of our work will be the deployment of the heterogenous implementations of the EDDLL algorithms in HPC infrastructures, giving COMPSs the responsibility of managing and distributing the computation across the heterogeneous devices available in the system, including CPUs, GPUs, FPGAs or accelerators like DNeuro.

The advances and results of this work will be reported in the deliverable D2.4 “EDDLL Hardware algorithms and adaptation to HPC (II)” in month 31.