DEEPHEALTH

D3.3 ECVL 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:	WP3/T3.2
Dissemination level:	[PUBLIC]
Document due Date:	01/06/2020 (M17)
Actual date of delivery	12/06/2020 (M17)
Leader of this deliverable	BSC
Author (s)	Lluc Alvarez (BSC), Miquel Moreto (BSC), Carles Hernandez (UPV), Jose Flich (UPV), Michael Steinacker (PROD), Heiko Mauersberger (PROD), Federico Bolelli (UNIMORE), Constantino Grana (UNIMORE), Tomas Teijeiro (EPFL), Marina Zapater (EPFL)
Version	V4

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	13/05/2020	First draft contents to request for contributions
1.1	29/05/2020	Partner's contributions received
2	01/06/2020	Version for review with all partners contributions
3	05/06/2020	Updated version after internal peer-review
4	12/06/2020	Version reviewed by Technical Manager and Project Coordinator

Table of contents

1 Introduction

The European Computer Vision Library (ECVL)[1] facilitates the integration and exchange of data between existing state-of-the-art Computer Vision (CV) and image processing libraries. Moreover, it provides new high-level CV functionalities thanks to specialized/accelerated versions of some CV algorithms commonly employed in conjunction with Deep Learning (DL) algorithms. The algorithms of ECVL are adapted to hardware accelerators (GPUs and FPGAs) in a user transparent way: the Hardware Abstraction Layer (HAL) hides hardware specific implementations of image manipulation functions. The user only decides which device should be used for the computation, moving the concerned image objects and then calling different Application Programming Interface (API) handlers that are the same for all hardware infrastructures.

The main objective of WP3 is to develop and deploy the ECVL library that is going to be used in the use cases of the project. The work done for this deliverable is part of the task T3.2 “ECVL adaptation to heterogeneous HPC hardware”, that aims at optimizing and adapting the most relevant and time-consuming algorithms and methods deployed in the ECVL library to HPC infrastructures with heterogeneous computing elements, namely high-end CPUs, GPUs, and FPGAs. This task tackles the analysis of the algorithms of the ECVL library and their adaptation to heterogeneous HPC infrastructures.

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 pandemic. 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 D3.3 “ECVL Hardware algorithms and adaptation to HPC”, we study the performance characteristics of the main functionalities of the ECVL library on different HPC infrastructures. This is currently an ongoing work that will be finished in month 27 and fully described in the deliverable D3.4 “ECVL Hardware algorithms and adaptation to HPC (II)”. The organization of this deliverable is as follows:

- Section 2 describes the current support for testing and integration of the ECVL library.
- Section 3 characterizes the performance of the algorithms of the ECVL library on general purpose processors.
- Section 4 presents the FPGA-based implementation of the algorithms of the ECVL library and an early performance evaluation.
- Section 4 discusses the advances on the development of the DeepHealth FPGA board.
- Section 5 remarks the main conclusions of the work done for this deliverable.

[1] ECVL code publicly available at https://github.com/deephealthproject/ecvl

2 Support for continuous testing and integration

One of the goals of the DeepHealth project is to develop an ECVL library that supports heterogeneous hardware while keeping the same user interface. Therefore, hardware specific implementations of image manipulation functions are hidden under a Hardware Abstraction Layer (HAL). As a result, the user only decides which device should be used for the computation, moving the concerned image objects with the `Image.to(device)` method, and then calls API handlers that are common for all the types of hardware devices.

In order to make sure that the ECVL library is always in a correct state, we need to build it and run tests on different environments after every commit. Since we do not want to perform this task manually, we decided to use a continuous integration toolflow deployed on a specific server. There are many different integration systems available, all of them with advantages and disadvantages. Among them, we selected Jenkins[2] for the following reasons:

1. It is successfully adopted by many big projects such as PyTorch, Netflix, Mozilla, Ubuntu, Docker and many others.
2. It is completely open source and can be used to automate almost every process.
3. Jenkins is a self-contained, open source automation server which can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software. In the specific case of ECVL, the automation pipeline must include the following steps:
    a. Download the last version of the source code;
    b. Build the code on multiple platforms and environments;
    c. Perform tests on the same environments;
    d. Generate the documentation and upload it to the website.

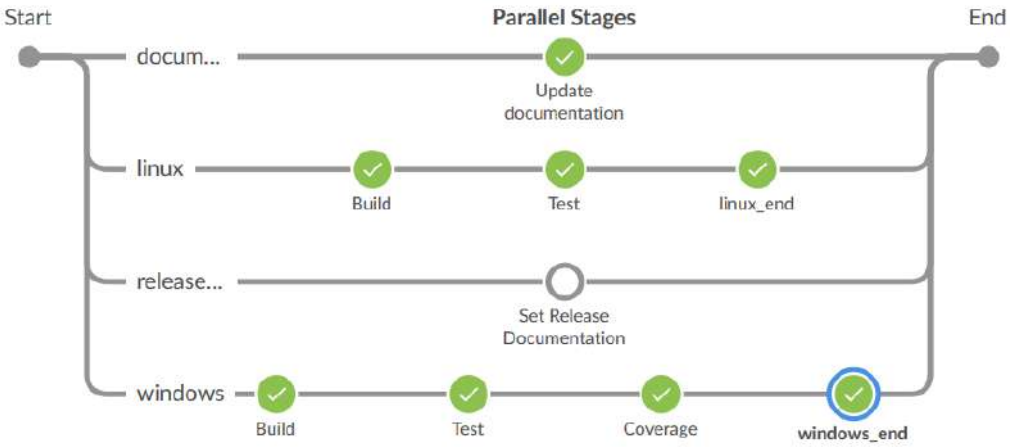

***Figure 1**: Example of Jenkins automated pipeline.*

An example of pipeline is depicted in Figure 1. In the figure, it is possible to see four parallel branches that are launched by the Jenkins master process and executed by slave containers. The documentation stage is run at each commit, so that the automatic documentation is always updated and aligned with the master branch. The third branch is run only at release time, to update the "official" release documentation, which will be also available in the future. Then, two stages cover Linux and Windows versions of the library. These consist of a building stage (with specific compilers), a test phase using GTest suite, and only for the Windows pipeline a coverage stage to analyse and report the percentage of code really tested in the previous stage. Each of these pipelines are run both on CPU and GPU platforms, so that we can test the deployability of the ECVL library on every platform. In the future, FPGA platforms will also be included in this pipeline.

[2] https://wiki.jenkins.io/pages/viewpage.action?pageId=58001258

The configuration file for the pipeline, called *Jenkinsfile*, is included in the GitHub repository[3], so that it can be easily modified if the necessity occurs. Figure 2 shows a code snippet of the Jenkinsfile.

```
stage('linux')
    agent
        docker
            label 'docker'
            image 'aimagelab/ecvl:latest'
    stages
        stage('Build')
            steps
                timeout(60)
                    echo 'Building..'
                    cmakeBuild buildDir: 'build',
                    cmakeArgs: '-DECVL_TESTS=ON -DECVL_BUILD_EDDL=ON -DECVL_DATASET=ON
                                -DECVL_WITH_DICOM=ON -DECVL_WITH_OPENSLIDE=ON',
                                installation: 'InSearchPath', sourceDir: '.', cleanBuild: true, steps: [
                                            [args: '--parallel 4', withCmake: true]]
        stage('Test')
            steps
                timeout(15)
                    echo 'Testing..'
                    ctest arguments: '-C Debug -VV', installation: 'InSearchPath', workingDir: 'build'
        stage('linux_end')
            steps
                echo 'Success!'
```

***Figure 2**: ECVL Jenkinsfile code example.*

The automation server receives a notification from GitHub right after every push event and triggers an execution of the pipeline. Detailed logs of the building and testing processes are available, and a badge is exposed on the GitHub page of the projects, stating whether the last build was successful or not. In order to build the project on more than one platform, we actually employ as many slaves as needed alongside the master. The validation of pull requests is another function of Jenkins that we use, to make sure that the master branch is not broken by the merging of other branches.

The diverse environments in which to build and test the ECVL library take the shape of Docker containers. A container is a unit of software that packages up code and all its dependencies, isolating them from the host environment. The reason for this choice is that it allows to choose, modify, and keep track of every build environment in a flexible way, and we can instantiate any of them in any machine where Docker can run. In fact, Jenkins downloads the appropriate Docker images on the chosen slaves before the beginning of the pipeline. The Jenkins master server itself runs inside a container, so that we can change the machine on which it runs with minimal effort.

Figures 3 and 4 list the currently tested environments in the ECVL library. As shown in these figures, the CPU implementation of ECVL has been successfully tested with diverse operating systems and compilers. In the next months, the GPU and the FPGA implementations will be integrated and tested.

Currently, most of the effort in this part of the work is being put on the support for GPUs, which is obtained by means of the NVIDIA Container Runtime. Jenkins can integrate with any tool that provides a command-line interface, so the integration with FPGA boards is certainly feasible and will be done in the future.

Of course, the lack of appropriate unit tests in the source code nullifies the effectiveness of a testing framework. For this reason, the Jenkins server is integrated with a tool for reporting code coverage, a measurement used to express which lines of code were executed by a test suite. For this task we chose Codecov[4]. Code coverage reports are generated and shown for each commit. Figure 5 shows an example of such a coverage report.

[3] Jenkinsfile available at https://github.com/deephealthproject/ecvl/blob/master/Jenkinsfile
[4] https://codecov.io/

Continuous integration (CPU)

Windows

OS	Compiler	OpenCV	EDDL	Infrastructure	Status
Windows 10 1903	VS 15.9.11	3.4.6	Latest Release	Jenkins	build passing

Linux

OS	Compiler	OpenCV	EDDL	Infrastructure	Status
Ubuntu 18.04.3	GCC 8.4.0	3.4.6	0.6.0	Jenkins	build passing
Ubuntu 18.04.4	GCC 6.5.0	3.4.10	0.6.0	Travis CI	build passing
Ubuntu 18.04.4	GCC 7.5.0	3.4.10	0.6.0	Travis CI	build passing
Ubuntu 18.04.4	GCC 8.4.0	3.4.10	0.6.0	Travis CI	build passing
Ubuntu 18.04.4	GCC 9.3.0	3.4.10	0.6.0	Travis CI	build passing
Ubuntu 18.04.4	Clang 5.0.2	3.4.10	0.6.0	Travis CI	build passing
Ubuntu 18.04.4	Clang 6.0.1	3.4.10	0.6.0	Travis CI	build passing
Ubuntu 18.04.4	Clang 7.1.0	3.4.10	0.6.0	Travis CI	build passing
Ubuntu 18.04.4	Clang 8.0.1	3.4.10	0.6.0	Travis CI	build passing
Ubuntu 18.04.4	Clang 9.0.0	3.4.10	0.6.0	Travis CI	build passing
Ubuntu 18.04.4	Clang 10.0.1	3.4.10	0.6.0	Travis CI	build passing

MacOS

OS	Compiler	OpenCV	EDDL	Infrastructure	Status
MacOSX 10.15.4	Apple Clang 11.0.3	3.4.10	0.6.0	Travis CI	build passing

***Figure 3**: CPU-based environments currently tested in ECVL.*

Continuous integration (GPU)

Windows

OS	Compiler	OpenCV	EDDL	Infrastructure	Status
Windows 10 1903	VS 16.2.0	-	-	Jenkins	Not available yet

Linux

OS	Compiler	OpenCV	EDDL	Infrastructure	Status
Linux (GPU)	GCC 8.4.0	3.4.6	-	Jenkins	Not available yet

***Figure 4**: GPU-based environments currently tested in ECVL.*

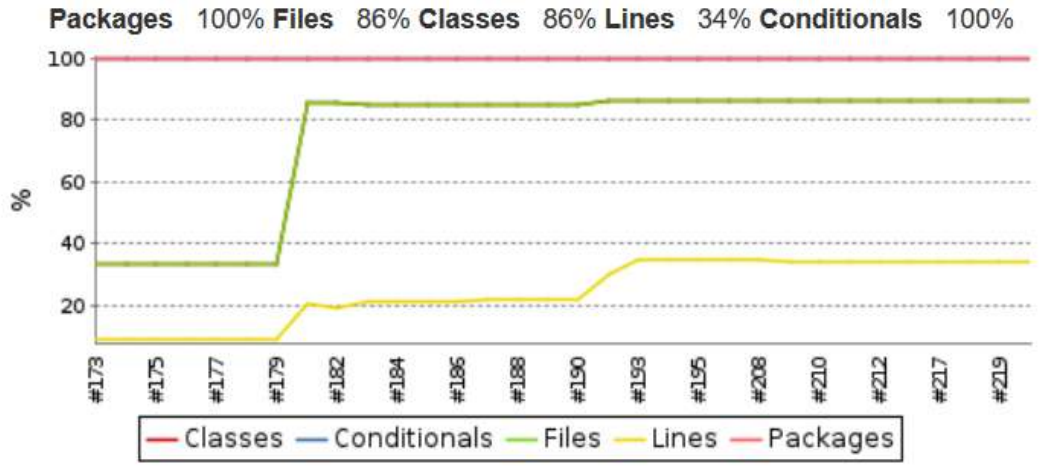

***Figure 5**: ECVL code coverage reports.*

3 Performance profiling and characterization with CPUS

This section presents a performance characterization of the ECVL library on HPC infrastructures with general purpose processors. The goal of this characterization is to identify the most time-consuming ECVL algorithms, to study their performance and bottlenecks, and to analyse their potential for being accelerated.

3.1 Experimental setup

The experimental setup used for the performance characterization is explained next, including the hardware description of the HPC infrastructures and the benchmark and input sets used in the experiments.

3.1.1 HPC infrastructure hardware and software environment

As described in the 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 removed 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, 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.

Both HPC infrastructures have the entire software ecosystem required to run ECVL. Table 1 shows the version of the GCC compiler, the OpenCV library and the OpenMP library used to perform the characterization experiments in both general purpose processors. In addition, several profiling tools have been used for the experimental evaluation and characterization of the ECVL library. 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 (e.g., 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.

***Table 1**: General purpose HPC processors for the ECVL evaluation*

Machine	Compiler	OpenCV	OpenMP
Intel Xeon	GCC 9.2.0	4.1.1	4.5 (201511)
IBM POWER9	GCC 9.2.0	4.1.1	4.5 (201511)

3.1.2 ECVL core functionalities and input sets

ECVL provides many functionalities to load, process and store images, which are essential components in a tool-chain of deep-learning models that focus on image classification and segmentation. In the context of DeepHealth, the ECVL library is used in the use cases mainly to perform data augmentation before feeding the deep-learning model. The data augmentation is a well-known and very important method to train visual recognition systems, and it gives the ability to increase the number of training examples in order to reduce overfitting and improve generalization.

The ECVL library implements 19 kernels that are used during the data augmentation phase. Some of the kernels are implemented inside the ECVL library, while some other kernels rely on the implementation of the underlying OpenCV library. Typically, during the data augmentation phase, various kernels will process a batch of dozens or even hundreds of images in parallel. Driven by this, we create a benchmark that mimics this behaviour and allows us to characterize the performance of each function provided by ECVL separately. To do so, the benchmark does thousands of parallel ECVL kernel calls to the same image. Note that, with this grain of parallelization, each ECVL kernel call runs in a single thread, and we restrict the underlying OpenCV library to use only one thread. The general flow of the benchmark operation is as follows:

```
#pragma omp parallel
{
  ImRead(Input)) // Open an Image
  #pragma omp for
  for 0 to NUMBER_OF_ITERATION:
    ecvl_library_kernel();
}
```

The size of the input images is one of the most important factors for the performance characterization of the ECVL library. To have a complete understanding of its behaviour and to characterize how it performs with images of various sizes, we use three images with different sizes and shapes, as described in Table 2.

***Table 2**: Input images size for the ECVL evaluation*

Image	Width x Height x Channels	Size
Small (Test.jpg)	675 x 900 x 3	0.132MB
Medium (Lena.png)	2048 x 2048 x 3	4.4MB
Large (img0015.png)	3072 x 2048 x 3	10.7MB

3.2 Performance characterization on Intel Xeon processors

This section presents the performance characterization of the ECVL library running on the HPC infrastructure with Intel Xeon processors.

3.2.1 Execution time and scalability

Figures 6, 7 and 8 show the performance scalability of the 19 kernels in the ECVL library when varying the number of OpenMP threads used in the benchmark for the small, medium and large images, respectively.

The results for the small image (see Figure 6) show that 9 out of the 19 ECVL kernels (ResizeDim, Rotate2D, ChangeColorSpace, FindCounters, Filter2D, SeparableFilter2D, GaussianBlur, AdditiveLaplaceNoise and GammaContrast) present perfect or very good scalability, achieving more than 40x speedup with 48 threads. After these, another group of 5 kernels (ResizeScale, Flip2D, Mirror2D, RotateFullImage2D and CoarseDropout) show a reasonable scalability of 30x to 40x with 48 threads. However, on the lower part of the plot, a group of 5 kernels (Threshold, ConnectedComponentsLabeling, Hconcat, Vconcat and Stack) suffer from poor scalability, as they achieve less than 20x speedup with 48 threads. These low-scaling kernels are very lightweight and take a very short time to process.

The results for the medium and for the large images (shown in Figures 7 and 8, respectively) follow similar trends to the ones discussed for the small image. The scalability of some kernels improves slightly when the size of the image increases, up to the point where 12 kernels achieve a speedup of more than 40x with 48 threads and a large image. However, even with large images, 4 kernels present a reduced scalability of less than 20x with 48 threads.

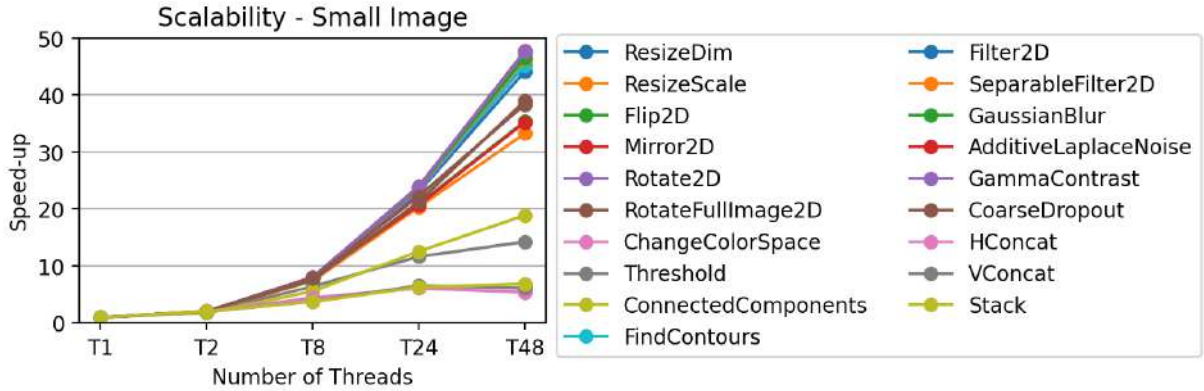

***Figure 6**: ECVL scalability results with the Intel Xeon processor and the small image input.*

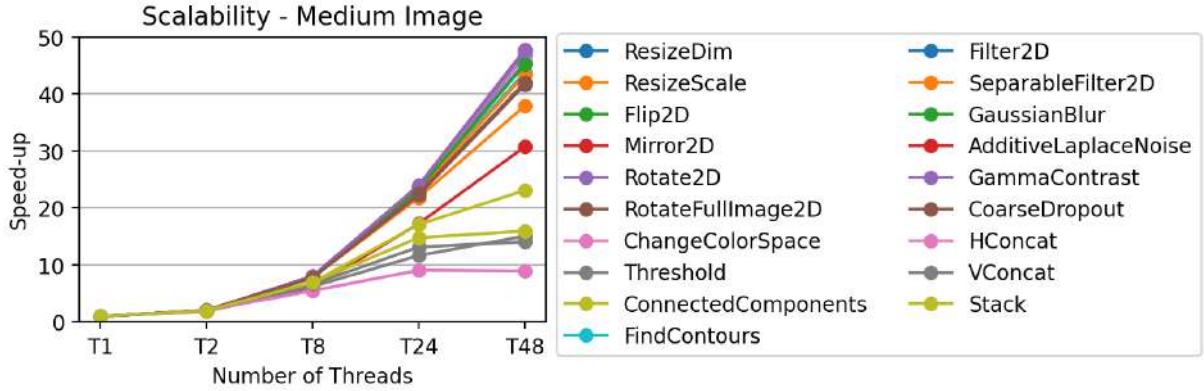

***Figure 7**: ECVL scalability results with the Intel Xeon processor and the medium image input.*

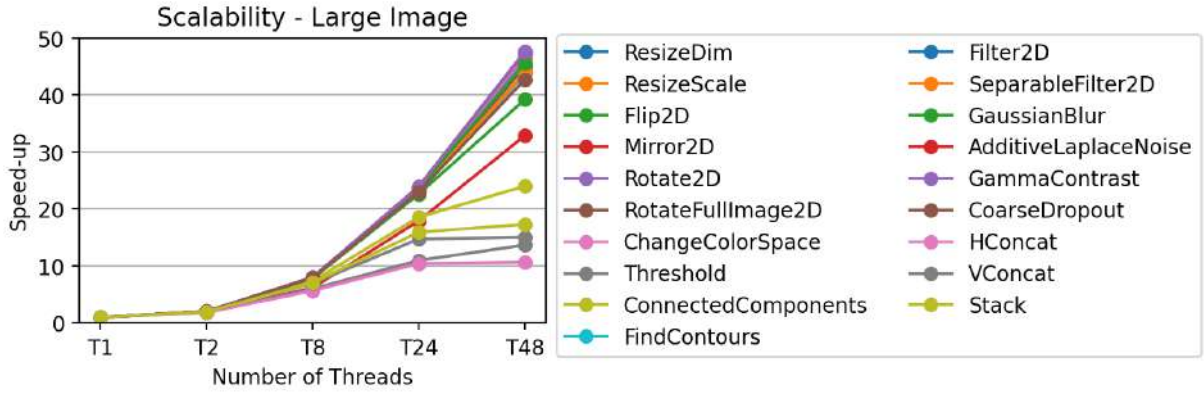

***Figure 8**: ECVL scalability results with the Intel Xeon processor and the large image input.*

The execution time per pixel is an interesting metric to study the weak scaling of the kernels of the ECVL library. This metric shows the time each kernel takes to process one pixel, and we compute it by measuring the execution time of each kernel and dividing the time by the number of pixels of the image that is being processed. Figure 9 shows the time per pixel of all the ECVL kernels for the three input sizes. The results show that all the kernels achieve very good weak scaling, as their time per pixel does not change significantly for the different input files. In addition, these results show that there are 2 kernels (AdditiveLaplaceN and GammaContrast) that have a much larger execution time than the rest of kernels, between 6 and 11 nanoseconds per pixel. Another group of 7 kernels (ResizeDim, ResizeScale, Rotate2D, RotateFullImage2, Filter2D, SeparableFilter2D, GaussianBlur) takes between 1 and 3 nanoseconds per pixel, while the rest of kernels take less than one nanoseconds per kernel. Note that the kernels that have a larger execution time are the same kernels that have a perfect or very good strong scalability, as shown in Figures 6, 7 and 8.

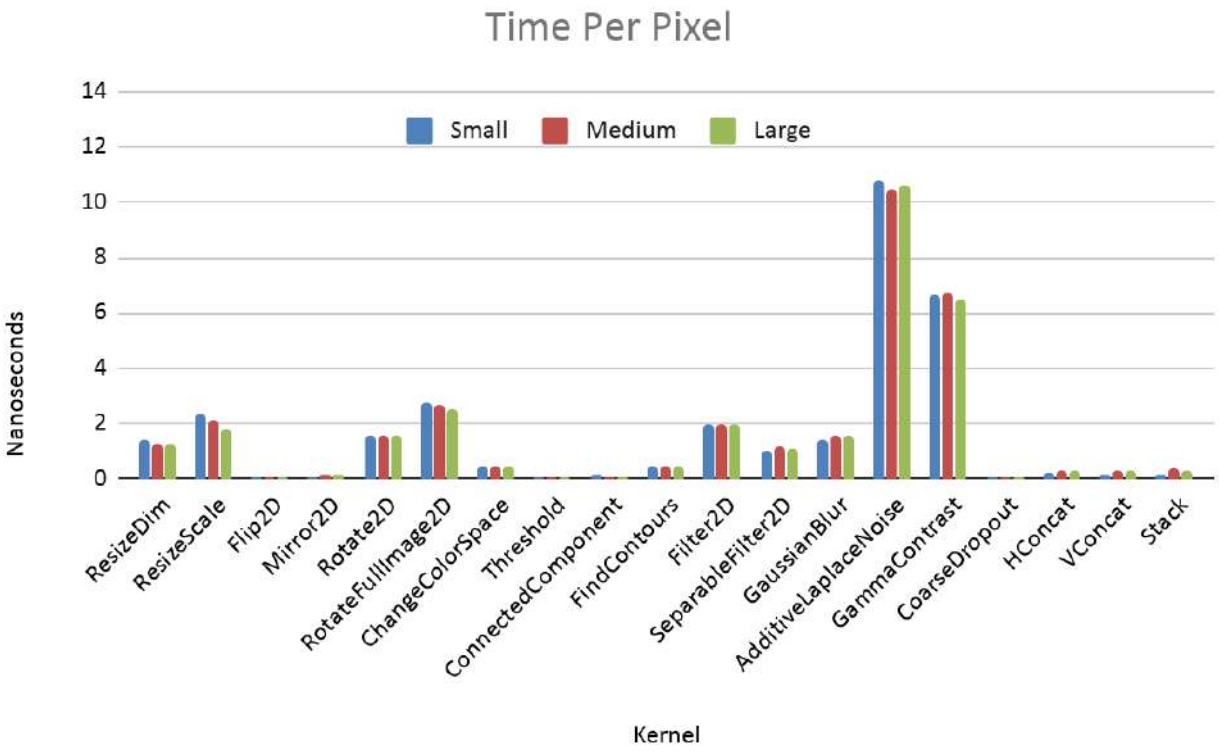

***Figure 9**: ECVL execution time per pixel with the Intel Xeon processor and different input image sizes (small, medium and large).*

3.2.2 Computing and memory intensity

Figure 10 shows the instructions per cycle (IPC) of the ECVL kernels when using different numbers of threads. These results are measured with the large input file. We also measure the IPC of the kernels with the medium and the small input file but, since the results are practically identical, they are not included in this document.

Results show that all the kernels but 3 (HConcat, VConcat and Stack) present large IPCs, more than 1.5 in all cases and up to 4.2 for Filter2D. In addition, in most of these kernels the IPC remains constant when increasing the number of threads, which indicates that the kernel has a very good scalability. Only 3 kernels (Mirror2D, ConnectedComponents and Threshold) suffer significant IPC degradations when increasing the number of threads.

These results also show that 3 kernels (HConcat, VConcat and Stack) present an extremely low IPC of less than 0.2. These 3 kernels use two images as input and create one image as output, so they are very memory intensive and the computing intensity is very low. In addition, these 3 kernels have been identified before as very little time consuming and present a very poor scalability.

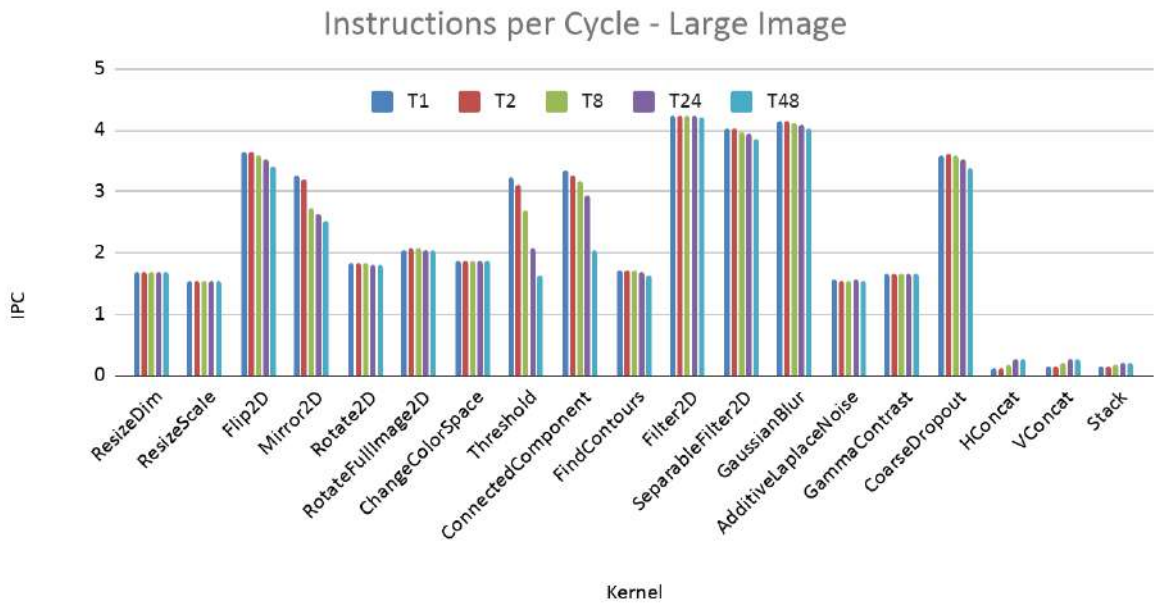

***Figure 10**: ECVL instructions per cycle with the Intel Xeon processor and the large input image.*

Figure 11 shows the cache misses per 1 thousand instructions (MPKI) of the ECVL kernels when using different numbers of threads. These results are measured with the large input file. We also measure the MPKI of the kernels with the medium and the small input file, which are not included in this document because the results are practically identical.

The last-level cache of the Intel Xeon processor is 32MB, so it can store the whole small, medium, and large input files. However, the private caches of each core in the Intel Xeon processor consist of a L1 of 32KB and a L2 of 1MB, so they can store the whole small image (132kB) but not the medium image (4.4MB) nor the large image (10.7MB).

Results show that all the kernels included in the figure (HConcat, VConcat and Stack are not included) present very low MPKI ratios, 8 in ConnectedComponents and less than 4 in the rest of kernels. This indicates that the vast majority of the memory accesses are served by the cache hierarchy with a very low latency, providing data on time for the CPU to achieve very high IPC, as shown previously. It can also be observed that the MPKI experiences a minor growth when increasing the number of threads, which indicates that the kernel presents a very good scalability.

Only 3 kernels (Mirror2D, Threshold and ConnectedComponents) suffer significant IPC degradations when increasing the number of threads.

3 of the ECVL kernels (HConcat, VConcat and Stack) present much larger MPKI ratios. These kernels are not included in Figure 11 for readability reasons, but their results are shown in Table 3. These 3 kernels use two images as input and create one image as output, so they are very memory intensive. The MPKI of these kernels varies between 45 and 210 when executing with 48 threads and different input sizes. Stack has the worst MPKI of 210 for the small file, and goes down to 72 and 69 for medium and large inputs. HConcat and VConcat behave similarly, with respective MPKI of 72 and 80 for the small input and 45 to 50 for the medium and large inputs.

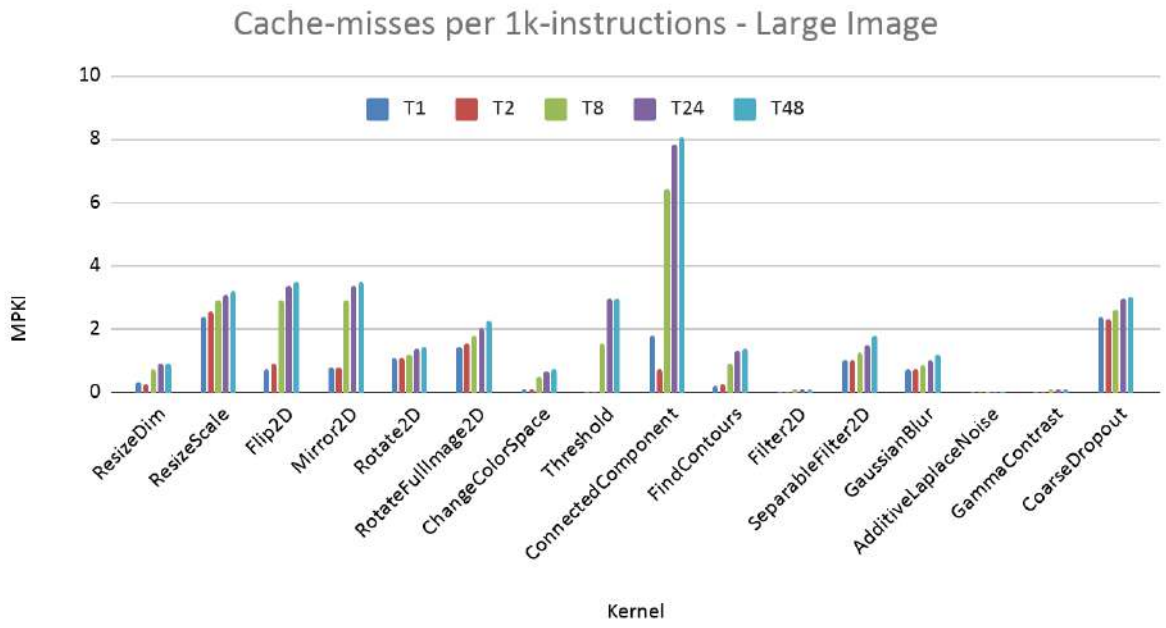

***Figure 11**: ECVL cache misses per 1K instructions with the Intel Xeon processor and the large input image.*

***Table 3**: ECVL cache misses per 1K instructions with the Intel Xeon processor and different input image sizes (small, medium and large).*

	Cache misses per kilo instruction		
Kernel	Small	Medium	Large
HConcat	72.08	47.04	45.45
VConcat	80.39	50.32	50.11
Stack	210.14	72.52	69.89

3.2.3 Scalability of fine-grain parallelization

Next, we explore the possibility to exploit fine-grain parallelization. In this parallelization strategy we exploit parallelism inside the ECVL kernels, relying on the OpenCV parallel implementations of each kernel. To do so, we configure our benchmark with a single thread that sequentially calls the ECVL kernels, and we vary the number of OpenCV threads.

Figure 12 shows the speedup obtained by the ECVL kernels when exploiting fine-grain parallelism with the large input image. It can be observed that the scalability of all the kernels is extremely poor. All kernels except 3 do not benefit from additional threads at all, while Rotate2D and ResizeScale

achieve less than 20% speedup with 48 threads, and RotateFullImage2D achieves 60% speedup with 48 threads. These 3 kernels use ECVL as a wrapper to call the OpenCV library.

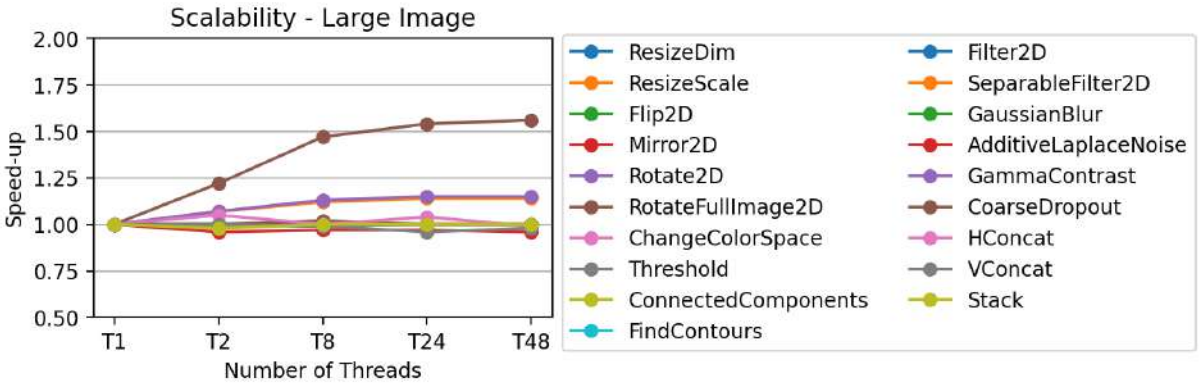

***Figure 12**: ECVL scalability results when exploiting fine-grain parallelization with the Intel Xeon processor and the large image input.*

3.3 Performance characterization on IBM POWER9 CPUs

This section presents the performance characterization of the ECVL library running on the HPC infrastructure with IBM POWER9 processors.

3.3.1 Execution time and scalability

Figures 13, 14 and 15 show the performance scalability of the 19 kernels in the ECVL library when varying the number of OpenMP threads used in the benchmark for the small, medium and large images, respectively.

The results for the small image (see Figure 13) show that the benchmarks scale moderately up to 40 threads, reaching speedups of up to 25x. Adding more than 40 threads does not increase performance in any kernel but 2, AdditiveLaplaceNoise and GammaContrast, which achieve up to 40x with 160 threads. Enlarging the input size improves the scalability significantly, as can be observed in Figure 14 and Figure 15. With the large image, 11 kernels benefit from adding more than 40 threads (IResizeDim, ResizeScale, Flip2D, Rotate2D, RotateFullImage2D, ChangeColorSpace, Filter2D, SeparableFilter2D, GaussianBlur, AdditiveLaplaceNoise and GammaContrast), reaching up to 45x with 160 threads in Rotate2D.

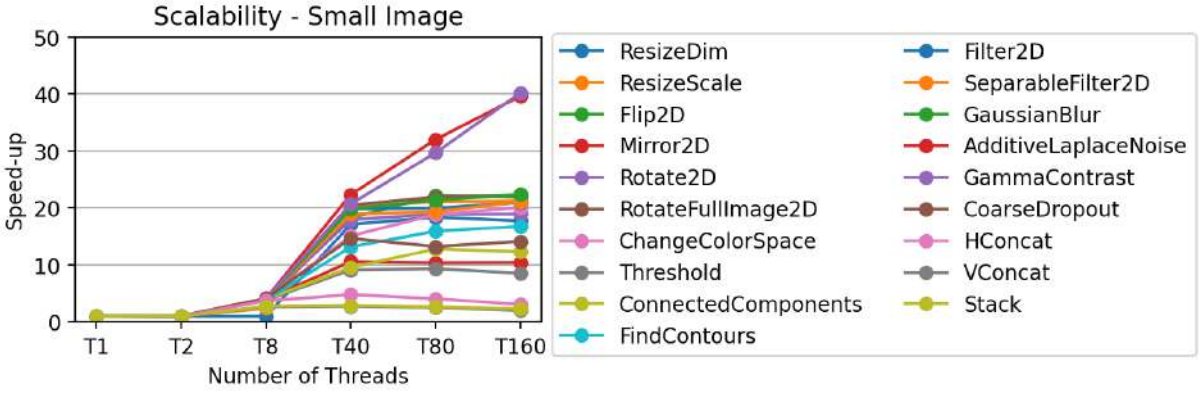

***Figure 13**: ECVL scalability results with the IBM POWER9 processor and the small image input.*

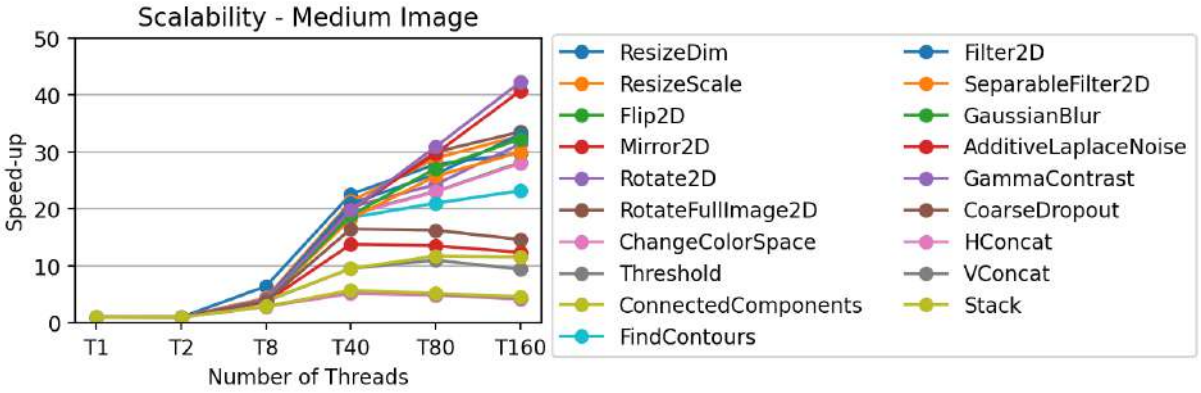

Figure 14: *ECVL scalability results with the IBM POWER9 processor and the medium image input.*

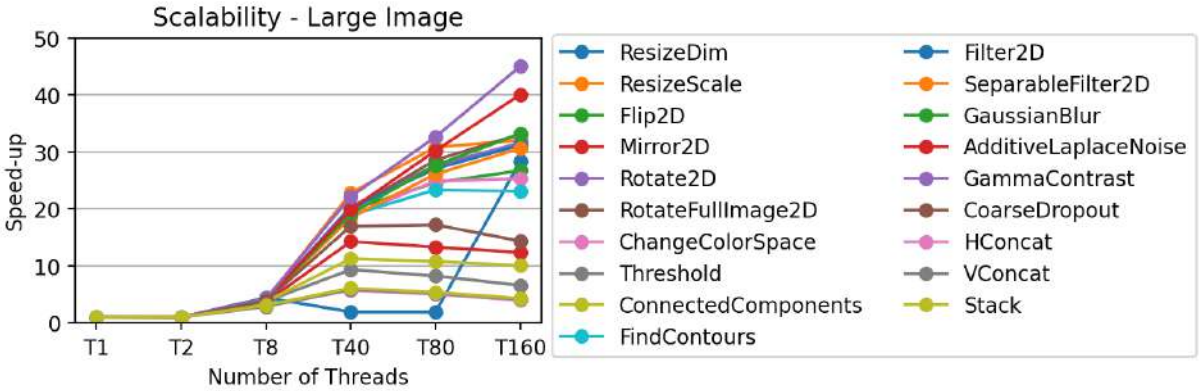

Figure 15*: ECVL scalability results with the IBM POWER9 processor and the large image input.*

To measure the weak scaling of the kernels we compute the execution time per pixel by measuring the execution time of each kernel and dividing the time by the number of pixels of the image that is being processed. Figure 16 shows the time per pixel of all the ECVL kernels for the three input sizes. The results show that most kernels achieve very good weak scaling, as the size of the input does not impact the time per pixel. As in the case of the Intel Xeon, there are 2 kernels (AdditiveLaplaceN and GammaContrast) that take much longer than the rest, between 6 and 12 nanoseconds per pixel. Another group of 7 kernels (ResizeDim, ResizeScale, Rotate2D, RotateFullImage2, Filter2D, SeparableFilter2D, GaussianBlur) takes between 1 and 4 nanoseconds per pixel, and are the ones that show a worse weak scaling. The rest of the kernels take less than one nanoseconds per pixel.

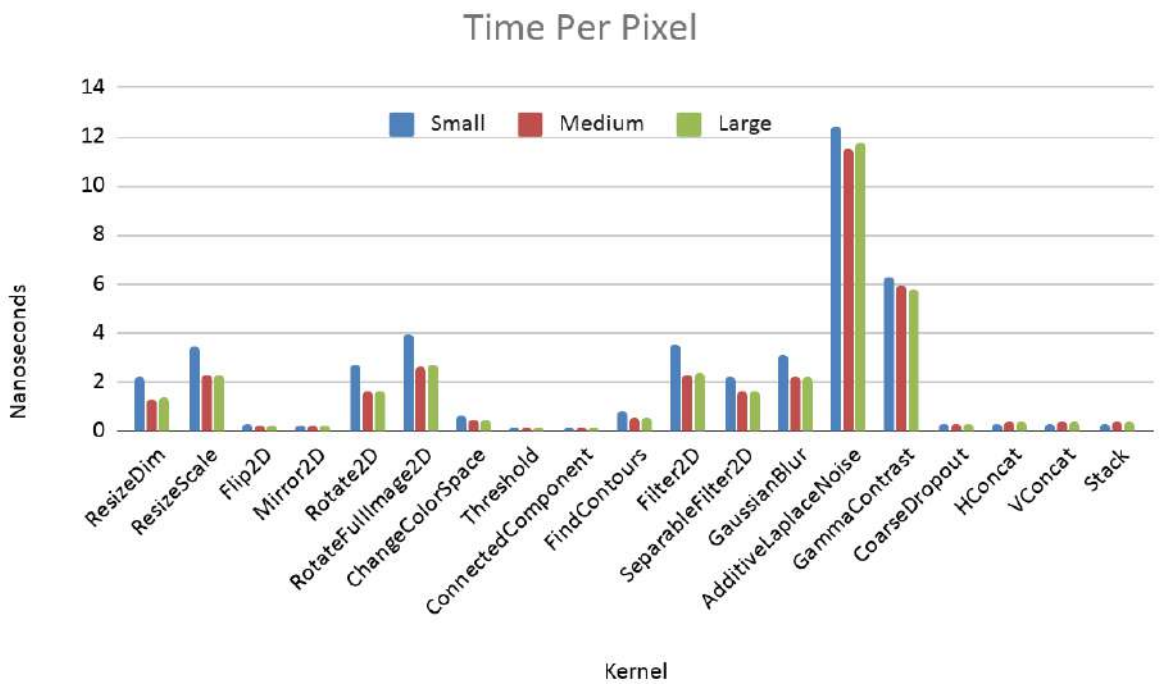

***Figure 16**: ECVL execution time per pixel with the IBM POWER9 processor and different input image sizes (small, medium and large).*

3.3.2 Computing and memory intensity

Figure 17 shows the instructions per cycle (IPC) of the ECVL kernels when using different numbers of threads. These results are measured with the large input file. We also measure the IPC of the kernels with the medium and the small input file but they are not presented in this document because the results are practically identical to the ones obtained with the large file.

Results show that all the kernels but 3 (HConcat, VConcat and Stack) present large IPCs. With execution of up to 80 threads, the IPC is more than 1 in all cases and reaches a peak of 3.4 for ConnectedComponents. Adding 160 threads causes an IPC drop in many kernels, as expected from the bad scalability observed before. These results also confirm that HConcat, VConcat and Stack have very low IPC of less than 0.5, they are very little time consuming, and they present a very poor scalability.

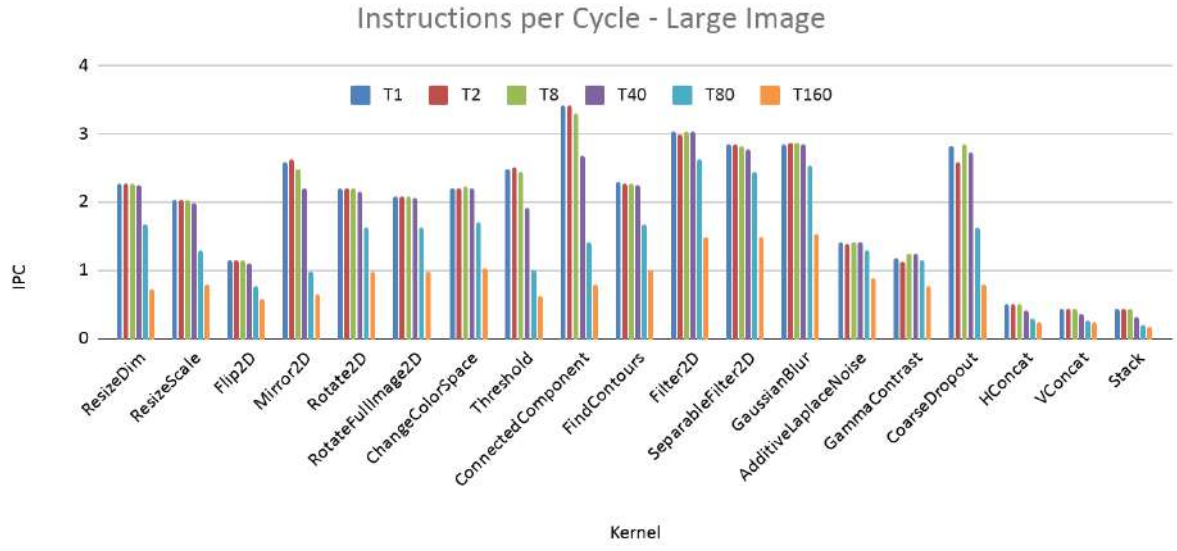

***Figure 17**: ECVL instructions per cycle with the IBM POWER9 processor and the large input image.*

Figure 18 shows the cache misses per 1 thousand instructions (MPKI) of the ECVL kernels when using different numbers of threads. These results are measured with the large input file. The MPKI results with the medium and the small input file are practically identical to the ones presented with the large input file, so they are not included in this document.

The last-level cache of the POWER9 processor is 10MB, so it can store the whole small, medium, and large input files. However, the private caches of each core consist of a L1 of 32KB and a L2 of 512KB, so they can store the whole small image (132kB) but not the medium image (4.4MB) nor the large image (10.7MB).

Results show that, with 40 threads or less, all the kernels except 3 have MPKI ratios lower than 3. HConcat, VConcat and Stack present moderately larger MPKI ratios, between 5 and 16 with no more than 40 threads. In all kernels, the MPKI grows significantly when increasing the number of threads to 80 and, specially, 160, reaching up to 36 MPKI in AddaptiveLaplace. This situation, that did not happen with the Intel Xeon processor, could be due to the synchronization overheads of adding threads that are idling.

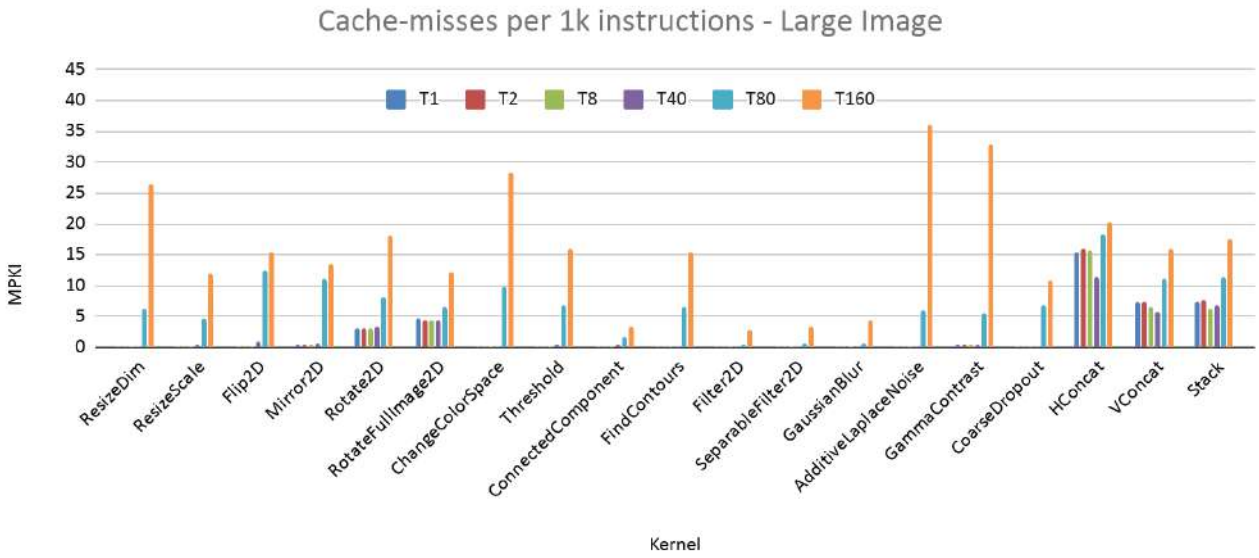

***Figure 18**: ECVL cache misses per 1K instruction with the IBM POWER9 processor and the large input image.*

3.4 Summary of the main characterization results

The presented characterization of the kernels of the ECVL library has shown that most of the kernels present a good or very good performance on both Intel Xeon and IBM POWER9 processors. In particular, most of the kernels are very computationally intensive and have low memory requirements, so they can be efficiently executed on high-performance CPUs and they achieve good or very good scalability when multiple kernel calls are executed in parallel on different cores. However, the multithreaded capabilities of the cores of IBM POWER9 processor do not provide significant advantages. On the other hand, we have identified three kernels that are less computationally intensive and present poor scalability, mainly because they are dominated by memory accesses that cannot be efficiently served by the cache hierarchy, as shown by their large MPKI ratio.

Based on this analysis, we can conclude that most of the kernels can greatly benefit from the enhanced computational capabilities of heterogeneous systems such as GPUs and FPGAs. In addition, the heterogeneous HPC infrastructures used in the DeepHealth project equip advanced memory technologies like HBM, which can alleviate the bottleneck observed in the memory intensive kernels.

4 FPGA-based accelerated algorithms

4.1 FPGA adaptation approach

For the adaption of the ECVL to the FPGA we have used a similar approach to the one we have used for the EDDL (see D2.3). The ECVL offers a hardware abstraction layer that facilitates the deployment of ECVL in different hardware devices with a single interface. In particular, methods that operate on ECVL images will check the device where the image is located, and execute the transformation there.

To ease the adaptation of ECVL to FPGAs we rely on the tools provided by Xilinx. We exploit the support for OpenCL provided by Xilinx to support the offloading process of ECVL kernels to the FPGA. However, the same philosophy can also be applied to other FPGA vendors providing OpenCL support.

Usually, when programming heterogeneous computing platforms, a program is composed of two pieces of code. One piece of code runs on the host CPU, it is coded using a high-level programming language (e.g., C/C++) and it is in charge of initializing the device, transferring data, offloading kernels to the FPGA and getting kernel results back to the host. The other software part, also known as kernel, is the part of the application that runs on the FPGA devices, and it can be coded in a high-level programming language like OpenCL or C++ or with lower-level programming models like Verilog or VHDL. The kernel code must be compiled with a specific compiler for the target hardware platform. In particular, the Xilinx and Intel compilers are *xocc* and *aoc*, respectively.

4.1.1 Target FPGA hardware

One of the target HPC infrastructures for the ECVL/EDDL libraries is the MANGO prototype, which is documented in Deliverable D5.1. The prototype is made up of 96 high-end FPGAs connected through five end nodes. Currently, the prototype is being adapted to the project. Practically all the FPGAs are manufactured by Xilinx, which imposes the adoption of OpenCL as the programming language both for the host and for the kernels. The FPGA being manufactured in the project will target, however, an Intel FPGA. In this sense, the adoption of OpenCL is still valid as it can be used for the host, although the kernels will need to be refactored to HLS. In order to speed up kernel developments we also target a Xilinx Alveo board where kernels are coded and tested.

4.1.2 Xilinx XfOpenCV

The Xilinx XfOpenCV library provides a software interface for computer vision functions accelerated on an FPGA device. The main advantage of this library lies in that their functions are mostly similar in functionality to their OpenCV equivalent, apart from some documented deviations. As the CPU implementation of the ECVL library provides many functionalities that are also present in OpenCV, we use XfOpenCV to accelerate some of the ECVL kernels, since XfOpenCV provides highly optimized implementations of these kernels.

To incorporate XfOpenCV in the Alveo board, and in general in any regular Xilinx board that is connected to a CPU using PCI express (like the ones in the MANGO cluster), we create a wrapper layer to convert xfOpenCV functions into Vitis kernels. As already explained in D5.1, Vitis is the latest software development tool provided by Xilinx to facilitate the use of FPGA-based acceleration.

It is important to understand that, due to the inherent characteristics of FPGAs, XfOpenCV uses input images (xf::Mat instances) of a fixed size. Since we want to support different image sizes with the same hardware configuration, we call the XfOpenCV functions using a worst-case image size, thus generating the acceleration hardware for the biggest image size that we want to support. Then, in the wrapper layer we zero-pad the input image, bringing it to the maximum size and making it supported by the accelerator. Figure 19 illustrates how the wrapper works. The code inside the wrapper box is executed in the FPGAs. This code also includes profiling capabilities to measure times. The wrapper allows the user to pass parameters, call the XFOpencCV kernels, and store the results.

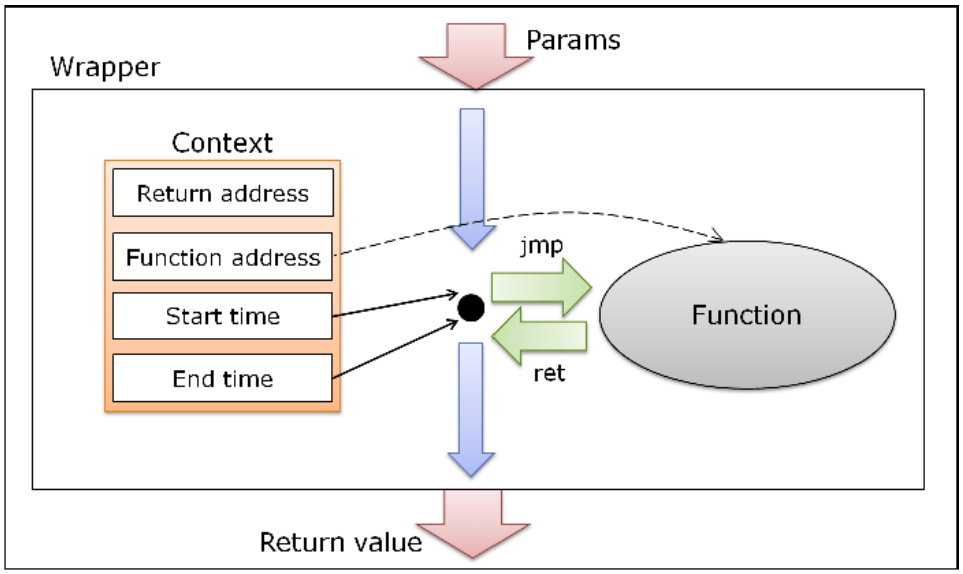

***Figure 19**: XfOpenCV wrapper.*

4.1.3 Types of kernels

Currently, we use 6 implementations of kernels used in the ECVL library, which were briefly introduced in Deliverable D5.1. With these kernels we were able to identify the major types of kernels we need to implement and the major issues we faced when adapting the platform to the expected operability of the ECVL library (the same occurs for EDDL which is documented in Deliverable D2.3). Indeed, we found that some kernels can be efficiently coded based on the OpenCV library. Those kernels perform exactly the same functionality, but instead of using OpenCV, we use XFOpenCV, the Xilinx library optimized for FPGAs.

We also found kernels that need to be coded manually as the implementations found in the ECVL library for CPU support are indeed coded manually. In this situation, we had to understand the functionality of the kernels and to find a compatible kernel in XFOpenCV library that performs the same work. Last, there are situations where we need to design totally custom kernels because there is no compatible support in XFOpenCV or because the XFOpenCV implementation cannot be used in our FPGAs due to lack of resources.

Below we provide the list of kernels we have currently ported to FPGA and classify them as based on OpenCV, using XFOpenCV, and full-custom implementations.

- Implemented kernels based on OpenCV:
    1) ResizeDimension based on cv::resize
    2) ResizeScale based on cv::resize
    3) Threshold based on cv::threshold
- Implemented kernels only using XFOpenCV
    4) GaussianBlur based on cv::resize
    5) Rgb2gray based on cv::resize
    6) OtsuThreshold based on cv::resize
- Custom kernels (in development for FPGA hardware, working in emulation):
    7) Flip2D
    8) Mirror2D

4.2 Description of the execution flow

For the sake of understanding, we illustrate the flow we currently use to run the ECVL library on the FPGA. As an illustrative example, we select one kernel which is the ResizeDim kernel.

We show three figures representing the execution flow. Figure 20 shows part of the code of the 'imgproc.cpp' file. In this file all the kernels needed by the ECVL library are coded. This means a single file embeds all the expected kernels to be coded in any of the target devices (CPU, GPU and FPGA). In the case of FPGA support we added a compilation flag (through a define in C) that allows us to switch on all the functionality for the FPGA support. This flag is a temporal strategy that will be substituted by the HAL strategy defined within the DeepHealth project. Notice that the adaptation effort to the HAL will be minimal since all the kernels implemented for FPGAs will not be affected, and only the host side part will need some rework. However, for every kernel instantiated in the FPGA, the host side part will need the same type of modifications. Using the compilation flags allows us to start our activities even before the HAL strategy was defined within the project.

The compilation switch, when enabled, allows all the host side parts for the FPGA support to be compiled. Instead, when the switch is disabled, the original code for the CPU is compiled. This enables us to compare the performance of both target devices for every kernel.

For the ResizeDim kernel in the figure we perform the following adaptations. First we transform the incoming image into a matrix (cv::Mat). This enables both XFOpencv and OpenCV working seamlessly with the image. For this adaptation we need to carefully understand the format of the incoming image and the expected format for the XFOpencv and OpenCV library kernels. In our example, the incoming image has three channels (RGB) and the data types of each channel is unsigned char (8 bits) per pixel (CV_8UC).

Once the input image is adapted, the OpenCV kernel is called. On completion, the result produced by the kernel is converted back to the expected format by ECVL. In fact, the produced matrix is converted back to an image format.

```
void ResizeDim(const ecvl::Image& src, ecvl::Image& dst, const std::vector<int>& newdims, InterpolationType interp)
{
    if (src.IsEmpty()) {
        ECVL_ERROR_EMPTY_IMAGE
    }

    if (src.channels_ == "xyc") {
        if (newdims.size() != 2) {
            throw std::runtime_error("Number of dimensions specified doesn't match image dimensions");
        }

#ifdef ECVL_WITH_FPGA
        cv::Mat src_mat = ImageToMat(src);
        cv::Mat m = cv::Mat::zeros(cv::Size(newdims[0], newdims[1]), CV_8UC(src_mat.channels()));
        ResizeDim_FPGA(src_mat, m, cv::Size(newdims[0], newdims[1]), GetOpenCVInterpolation(interp));
        dst = ecvl::MatToImage(m);

#else
        using namespace std::chrono;
        cv::Mat m;
        high_resolution_clock::time_point t1 = high_resolution_clock::now();
        cv::resize(ImageToMat(src), m, cv::Size(newdims[0], newdims[1]), 0.0, 0.0, GetOpenCVInterpolation(interp));
        high_resolution_clock::time_point t2 = high_resolution_clock::now();
        dst = ecvl::MatToImage(m);
        duration<double> time_span = duration_cast<duration<double>>(t2 - t1);
        std::cout << "Tiempo de ejecucion ResizeDim en cpu: " << time_span.count() << " seconds.";
        std::cout << std::endl;
#endif
    }
    else {
        ECVL_ERROR_NOT_IMPLEMENTED
    }
}
```

***Figure 20**: Host code in imgproc.cpp, ResizeDim function.*

The second step when coding FPGA support for ECVL is to provide a wrapper to every function that sends arguments to its kernel. The wrappers are coded in the 'imgproc_fpga' file and are executed on the host side prior to calling the kernels running on the FPGA. Every coded wrapper is implemented with a method. Figure 21 shows the function that implements the wrapper for the ResizeDim kernel.

```
void ResizeDim_FPGA(const cv::Mat& src, cv::Mat& dst, cv::Size dsize, int interp)
{
    /* The interp parameter is ignored at the moment.
     * The xfOpenCV generates an accelerator for the Area interpolator
     * To change the accelerator interpolation strategy, its header needs to be changed,
     * and the hardware resynthesized
     */
    (void) interp;

    std::vector<cl::Device> devices = xcl::get_xil_devices();
    cl::Device device = devices[0];
    cl::Context context(device);

    cl::CommandQueue q(context, device,CL_QUEUE_PROFILING_ENABLE);

    std::string device_name = device.getInfo<CL_DEVICE_NAME>();
    std::string binaryFile = xcl::find_binary_file(device_name,"ecvl_kernels");
    cl::Program::Binaries bins = xcl::import_binary_file(binaryFile);
    devices.resize(1);
    cl::Program program(context, devices, bins);
    cl::Kernel krnl(program,"resize_accel");

    cl::Buffer imageToDevice(context,CL_MEM_READ_ONLY, src.rows * src.cols * src.channels()); // TODO check src datatype
    cl::Buffer imageFromDevice(context,CL_MEM_WRITE_ONLY, dst.rows * dst.cols * dst.channels());

    /* Copy input vectors to memory */
    q.enqueueWriteBuffer(imageToDevice, CL_TRUE, 0, src.rows * src.cols * src.channels(), src.data);

    krnl.setArg(0, imageToDevice);
    krnl.setArg(1, imageFromDevice);
    krnl.setArg(2, src.rows);
    krnl.setArg(3, src.cols);
    krnl.setArg(4, dsize.height);
    krnl.setArg(5, dsize.width);

    // Profiling Objects
    cl_ulong start= 0;
    cl_ulong end = 0;
    double diff_prof = 0.0f;
    cl::Event event_sp;

    printf("Launching kernel: Resize \n");
    q.enqueueTask(krnl,NULL,&event_sp);
    clWaitForEvents(1, (const cl_event*) &event_sp);
    printf("Launched kernel: Resize \n");

    event_sp.getProfilingInfo(CL_PROFILING_COMMAND_START,&start);
    event_sp.getProfilingInfo(CL_PROFILING_COMMAND_END,&end);
    diff_prof = end-start;
    std::cout<<(diff_prof/1000000)<<"ms"<<std::endl;

    q.enqueueReadBuffer(imageFromDevice, CL_TRUE, 0, dst.rows * dst.cols * dst.channels(), dst.data);
```

***Figure 21**: code in imgproc_fpga.cpp, ResizeDim function.*

The structure of this function is the one recommended by Xilinx in its XFOpenCV User Guide (2019.1 in our case). Our code is compiled and run on the host and provides the data and control signals to the attached hardware within the FPGA. The code in Figure 21 is written using OpenCL constructs and provides capabilities for setting up, and running a kernel on the FPGA.

The code performs the following actions in the described order:

1) Loading the kernel binary on the FPGA – xcl::import_binary_file() loads the bitstream and programs the FPGA to enable the required processing of data.

2) Setting up memory buffers for data transfer – Data needs to be sent and read from the DDR memory on the hardware. cl::Buffers are created to allocate required memory for transferring data to and from the hardware.
3) Transfer data to and from the hardware –enqueueWriteBuffer() and enqueueReadBuffer() are used to transfer the data to and from the hardware at the required time.
4) Execute kernel on the FPGA – There are functions to execute kernels on the FPGA. There can be single kernel execution or multiple kernel execution that could be asynchronous or synchronous with each other. Commonly used command is enqueueTask().
5) Profiling the performance of kernel execution – The host code in OpenCL also enables measurement of the execution time of a kernel on the FPGA. The function used in our examples for profiling is getProfilingInfo().

Having seen these features it is easy to identify the process in the figure. However, from completeness, the use of the next elements must be noted as well:

a) To detect the device 'u200' it is used "xcl::get_xil_devices" with cl::Device and cl::Context.
b) The kernel to run is selected with "cl::Kernel krnl(program,"resize_accel")", knowing that the kernel is in the bitstream.
c) Buffers "imageToDevice" and "imageFromDevice" respectively transfer the source image to the FPGA and collect the produced image when the kernel has finished. It is important to focus on the size of the buffer. In this case both have the same size: cols * rows * number of channels of the image (R+G+B). If the size is not exact, errors will be thrown.
d) Buffers can also be instantiated to pass a value (e.g. an integer or a float variable) in which we will receive the result.
e) The arguments received from the host code in 'imgproc' file are passed with "krnl.setArg" and the number of the argument.

Notice that this example stresses the case of running one single kernel. For the practical case of ECVL running in a training process all the kernels needed will be loaded to the FPGA at the same time and the memory buffers will already reside in the FPGA memory. Thus, all these steps will be factored out. This integration will come with the FPGA support part for the EDDL library..

The last step is to define the wrapper code. This code will be loaded in the bitstream and therefore, executed on the FPGA. An example of this code is shown in Figure 22.

All xfOpenCV kernels are provided with C++ function templates with image containers as objects of xf::Mat class. In addition, these kernels work either in stream based (where complete image is read continuously) or memory mapped (where image data access is in blocks), which is our case.

SDAccel flow (OpenCL) requires kernel interfaces to be memory pointers with a width in power of 2. Thus, glue logic is required for converting memory pointers to xf::Mat class data type and vice-versa when interacting with XFOpenCV kernels. Wrappers are built over the kernels with this glue logic.

To facilitate the conversion of pointers to xf::Mat and vice versa, two adapter functions are used as part of XFOpenCV: xf::Array2xfMat() and xf::xfMat2Array(). It is necessary for the xf::Mat objects to be invoked as streams using HLS pragmas with a minimum depth of 2. This is the reason why in the figure we convert 'img_inp' and 'img_out' arguments into 'in_mat' and 'out_mat' of xf::Mat. The Data Types 'ap_uint' are used because it is an enhancement type made for HLS with C++. It can be used as fixed point types like 'ap_fixed'. However, it is not useful as Array2xfMat and xfMat2Array only accept unsigned or integer types, not float types.

Other noticeable elements that can be seen in Figure 22 are HLS Interface pragmas at the beginning. The arguments of the wrapper are pointers and all of them are mapped to global memory. The data is accessed through AXI interfaces which can be mapped to different banks, which are buffer-like structures where the FPGA kernel will receive the data. The memory interface specification can be one of the following two:

1) The first approach is to define which argument the AXI (m_axi) memory map interface is accessed, that is why only the images or variables to load in the FPGA must be in this part. An offset is always required. The 'offset=slave' means that the offset of the array <variable_name> will be made available through the AXI slave interface of the kernel. The 'port' means the argument that it is mapped to and 'bundle=gmem' refers to the DDR memory.
2) The second approach is a pragma for the AXI Slave interface. Scalars (and pointer offsets) are mapped to one AXI Slave control interface which must be named control. Scalars are considered constant inputs and should also be mapped to s_axilite, such as integers, pointers to integers or other types, but always arguments not loaded in the FPGA.

Finally, it is the call to the XFOpenCV Kernel, in Figure 22 "xf:resize". The kernel does its work and returns the result in an "xf::Mat" structure, so through 'xfMat2Array', it is transformed and sent back to the host.

The arguments of each kernel are different and can have variations, but in this example we use different constant variables like 'HEIGHT', 'WIDTH', 'NEWHEIGHT' and 'NEWWIDTH' that respectively specify the maximum input rows, input columns, output rows and output columns that the kernel can accept.

Alternatively, if a grayscale image needs to be manipulated, the constant variable 'TYPE' will be XF_8UC1 (1 channel) and, depending of the number of pixels to be processed per cycle (NPC_T), possible options are XF_NPPC1 and XF_NPPC8 for 1 pixel and 8 pixel operations, respectively.

```
extern "C" {
void resize_accel(ap_uint<INPUT_PTR_WIDTH> *img_inp, ap_uint<OUTPUT_PTR_WIDTH> *img_out, int rows_in, int cols_in, int rows_out, int cols_out)
{
#pragma HLS INTERFACE m_axi     port=img_inp  offset=slave bundle=gmem1
#pragma HLS INTERFACE m_axi     port=img_out  offset=slave bundle=gmem2
#pragma HLS INTERFACE s_axilite port=img_inp               bundle=control
#pragma HLS INTERFACE s_axilite port=img_out               bundle=control
#pragma HLS INTERFACE s_axilite port=rows_in               bundle=control
#pragma HLS INTERFACE s_axilite port=cols_in               bundle=control
#pragma HLS INTERFACE s_axilite port=rows_out              bundle=control
#pragma HLS INTERFACE s_axilite port=cols_out              bundle=control
#pragma HLS INTERFACE s_axilite port=return                bundle=control

    const int pROWS_INP = HEIGHT;
    const int pCOLS_INP = WIDTH;
    const int pROWS_OUT = NEWHEIGHT;
    const int pCOLS_OUT = NEWWIDTH;
    const int pNPC = NPC_T;

    xf::Mat<TYPE, HEIGHT, WIDTH, NPC_T> in_mat;
#pragma HLS stream variable=in_mat.data depth=pCOLS_INP/pNPC

    xf::Mat<TYPE, NEWHEIGHT, NEWWIDTH, NPC_T> out_mat;
#pragma HLS stream variable=out_mat.data depth=pCOLS_OUT/pNPC

    in_mat.rows = rows_in;  in_mat.cols = cols_in;
    out_mat.rows = rows_out;  out_mat.cols = cols_out;

#pragma HLS DATAFLOW

    xf::Array2xfMat<INPUT_PTR_WIDTH,TYPE,HEIGHT,WIDTH,NPC_T>(img_inp,in_mat);
    xf::resize<INTERPOLATION,TYPE,HEIGHT,WIDTH,NEWHEIGHT,NEWWIDTH,NPC_T,MAXDOWNSCALE> (in_mat, out_mat);
    xf::xfMat2Array<OUTPUT_PTR_WIDTH,TYPE,NEWHEIGHT,NEWWIDTH,NPC_T>(out_mat,img_out);
}
}
```

Figure 22: *code in ResizeDim wrapper.*

4.3 FPGA compilation tool flow

The details about how to compile FPGA kernels are given in D5.1. However, it is important to mention that, for the examples provided in this deliverable, we have compiled ECVL kernels in a stand-alone manner. This means that each time we use a kernel we have to program the FPGA with the corresponding bitstream. For the final version of the FPGA support in the ECVL the bitstream will be programmed only once and it will contain all the kernels needed for a particular example. In

fact, we expect to co-host EDDL and ECVL kernels in the same bitstream, being the EDDL library the one in charge of loading the appropriate bitstream at initialization time.

4.4 FPGA kernels performance

We have evaluated the default performance of the kernels we have already successfully implemented in the FPGA. To ease the comparison with CPU results we have computed the time required per pixel for each of the kernels.

Figure 23 shows time per pixel in nanoseconds for three different image sizes. The first thing we observe is that the performance per pixel is quite similar regardless of the image. Additionally, we also observe that the values for the FPGA implementation are similar than the ones provided by the CPU. However, while CPU relies on using threads to reduce the time to process an image, the FPGA offers other much fine grain optimizations that are able to improve performance in the ECVL kernels.

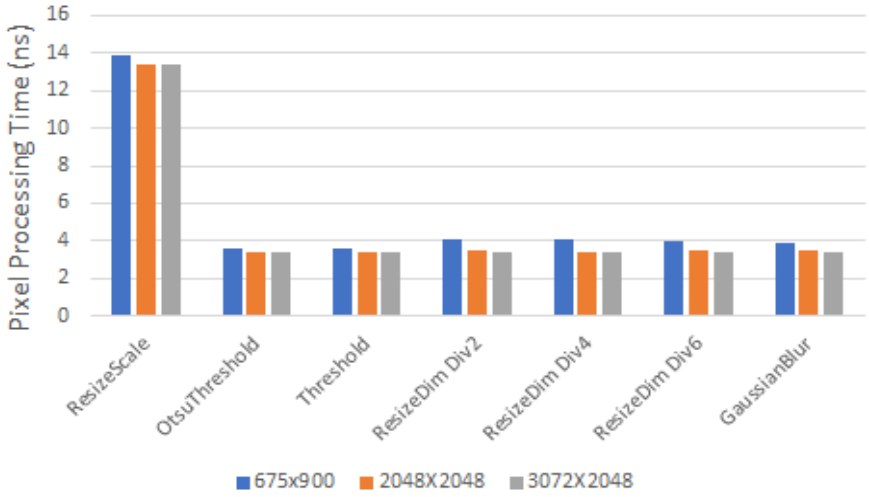

***Figure 23**: ECVL execution time per Pixel with the FPGA and different input image sizes.*

Figure 24 shows the speedup obtained when implementing kernels using operators of 8 pixels width. To do so, we have to re-implement the kernels in the FPGA varying this parameter. As shown in the plot when increasing the parallelism at the pixel level we are able to improve performance significantly. In particular, we get an average speedup of 21,4 when using 8-pixel operators.

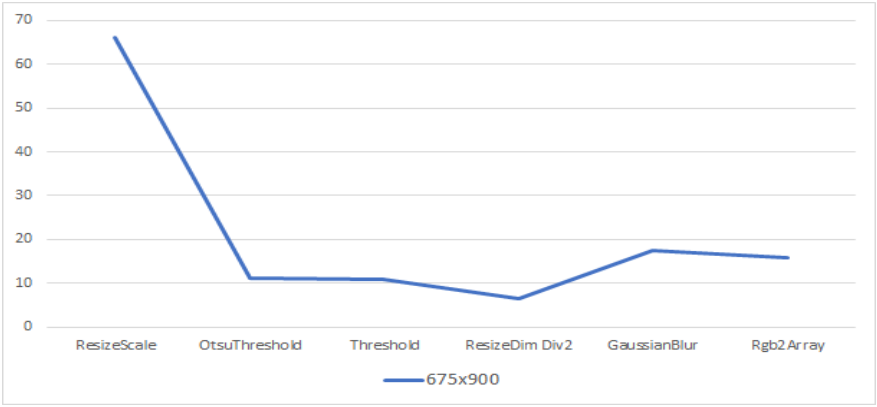

***Figure 24**: Speedup using operators of 1 pixel/cycle against operators of 8 pixel/cycle.*

It is important to mention that, for the results shown above, we have also used two additional basic optimization options. The first one is the utilization of the HLS_DATAFLOW pragma. This pragma is useful to force the synthesis tool to implement concurrent kernel computations in a pipelined fashion allowing exploiting the maximum throughput provided by the kernel frequency. Time per pixel

results shown above do not benefit from this optimization since the results have been computed using only a single image. We will explore the impact of using batches of hundreds of images when integrating the ECVL library with the EDDL. The second optimization we have implemented is the utilization of two separate memory banks for input and output images to allow maximizing memory bandwidth. To do so, we have to specify different bundles to the parameter interfaces to ensure input and output buffers are mapped to different memory ports.

4.5 FPGA resources used by kernels

Table 4 summarizes the resources required by the kernels we have implemented and the maximum frequency we can achieve with each of them. These values have been collected for the 1 pixel operations in the ALVEO U200 using a target frequency of 300 MHz. As shown in the table, the amount of resources used by the kernels are rather limited. Note that the ALVEO U200 board has 892 KLUTs, which means that none of the kernels requires more than 6% of the available resources. Additionally, we observe that some kernels are able to operate at higher frequencies. However, this usually comes at the expense of an increase in the hardware resources.

***Table 4**: FPGA resource utilization by the ECVL kernels.*

Wrapper Name	Target Frequency	Estimated Frequency	FF	LUT	DSP	BRAM
gaussian_accel	300.300293	300.029999	39472	53605	298	86
resize_accel	300.300293	411.015198	18894	37320	21	87
threshold_accel	300.300293	411.015198	6970	26075	16	40
otsuThreshold_accel	300.300293	335.008392	11245	19465	65	34
rgb2gray_accel	300.300293	411.015198	12356	51027	40	133

4.6 Problems found and mitigation solutions

The aim of this section is to explain some differences we had when trying to implement the wrappers. It is commented before the general example of a wrapper and its execution flow. However, in some situations, the code changes due to lack of resources on our FPGAs or special kernels that do not require images. That is the case of our Custom Kernels.

To implement both cv:Flip and cv:Mirror OpenCV kernels there is an equivalent kernel in XFOpenCV called xf::Remap. However, when compiling, Remaps throws an error because there are not enough RAMB18 and RAMB36/FIFO cells (5264 are required but only 4320 compatible sites are available). Thus, we decided to implement these kernels manually and name them as ‘custom’.

The difference is basically that, inside the wrapper (Figure 25), we relocate the rows and columns so the output image is the mirror or flip. In addition, the kernel xf::Remap only works with images in grayscale and not with RGB. This eventually could be a problem for losing time converting a RGB image to grayscale and later from grayscale to RGB. This problem is solved with this custom implementation, but it has only been tested in software emulation. On top of this, the kernel xf::Remap does not support unsigned/signed integers as pixel Type of input images. This problem becomes important knowing that the functions ‘Array2xfMat’ and ‘xfMat2Array’ only accept

unsigned/signed integers, therefore the use of the kernel and the functions that transform the image into arrays to be processed are incompatible.

```
#pragma HLS DATAFLOW

    xf::Array2xfMat<INPUT_PTR_WIDTH,TYPE,HEIGHT,WIDTH,NPC_T>(img_inp,in_mat);
    for(int i = 0; i<rows_in;i++){
        for(int j = 0; j<cols_in;j++){
            out_mat.data[i*cols_in +j] = in_mat.data[i*cols_in + (cols_in - 1 -j)];
        }
    }
    xf::xfMat2Array<OUTPUT_PTR_WIDTH,TYPE,HEIGHT,WIDTH,NPC_T>(out_mat,img_out);
}
}
```

***Figure 25**: Wrapper for the Flip kernel.*

4.7 Expected enhancements in the following months

In the following months we will extend the FPGA support to the ECVL kernels that have not been yet integrated in the FPGA flow. Additionally, we will work on the performance optimization of the kernels by exploiting pixel-level parallelism beyond what the XFOpenCV is able to provide by developing full-custom implementation for certain computationally demanding kernels. Finally, we also expect to devote significant efforts in finding the most appropriate strategy to co-host EDDL and ECVL kernels in the FPGA.

5 FPGA-based cluster

Within the DeepHealth Project, there are different FPGA-based hardware infrastructures that we plan to evaluate as of an FPGA cluster:

- MANGO hardware / Xilinx FPGAs + PCI Express extension kit
- MANGO hardware / Intel FPGAs + PCI Express extension kit
- DeepHealth PCI Express board

The MANGO hardware is an outcome of the MANGO European project, which is currently being adapted to be used in the context of DeepHealth. Therefore, this section is focused on the DeepHealth PCI Express board.

A high memory bandwidth of mid-size memories was identified as a key requirement for the new FPGA board to be developed for DeepHealth, especially for the acceleration of the functionalities of the EDDL library. Thus, we decide to use an FPGA with on-chip High Bandwidth Memory (HBM) for high bandwidth, as well as on-chip DDR4 memory for high capacity requirements.

A second key requirement is the communication bandwidth between the FPGA and the host, as well as between several FPGA boards. PCI Express was selected as the communication interface with the host. This interface is supported by almost all HPC servers.

The DeepHealth PCI Express board will implement the Intel Stratix-10 MX1650 or MX2100 FPGA. Both types are package and pin compatible. The MX2100 provides 16GByte of HBM memory and the MX1650 provides 8GByte of HMB memory. The MX2100 is the preferred choice and will be used for the first boards. The concept of the board is shown in Figure 26. To adopt the board to different applications, it provides several connectors to extend the capabilities in a modular way.

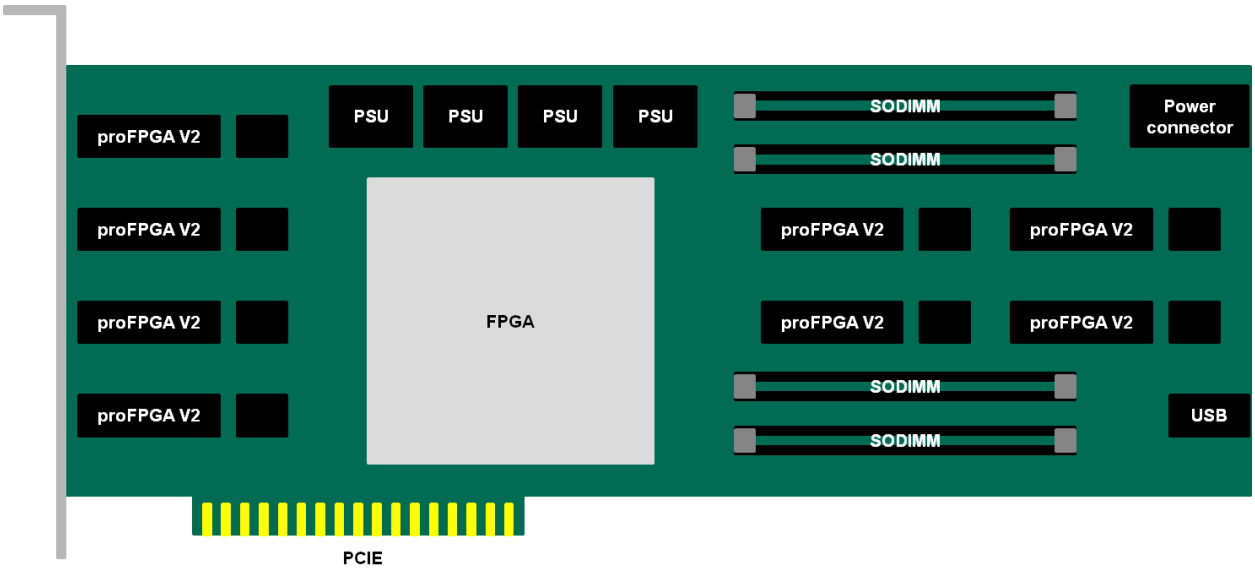

***Figure 26**: Concept of the DeepHealth FPGA board.*

The SODIMM connectors are used for memories and peripherals which are connected to regular FPGA I/Os. In DeepHealth, these SODIMM extension board sites can be used to attach additional memories to the FPGA depending on the application requirements. Examples of such memories are DDR4 memory or high speed SRAM memories to support random memory access with low latency.

The Gigabit transceivers of the FPGA will be made available using the “proFPGA V2” connectors. These connectors allow data rates with more than 100 GBit/s per differential pair signal. Using these V2 connectors, interfaces like QSFP28 (see Figure 27) and Firefly can be attached, or connections to other FPGA boards using dedicated cables can be established. Those connectors allow scalability of the hardware in terms of capacity.

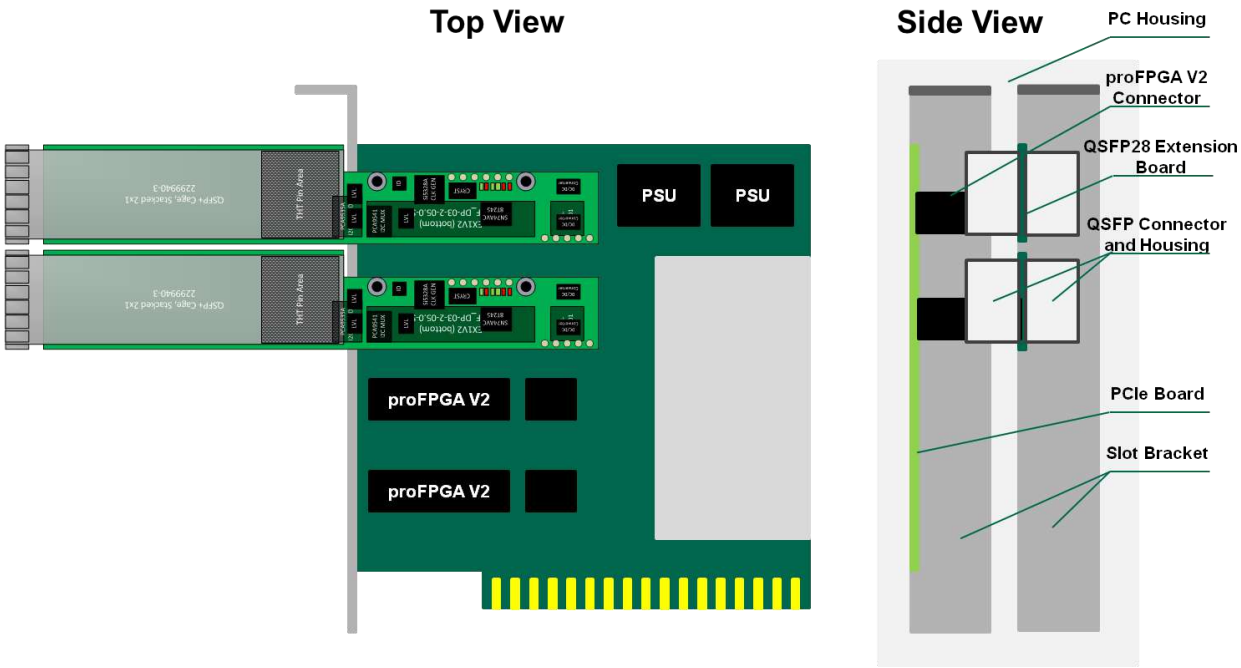

***Figure 27**: Case study - Extensibility with QSFP28 interfaces.*

For the communication with the host computer, two approaches will be supported:

1. A performance and latency optimized PCI Express interface IP, which consists of a PCI Express controller, a Linux device driver, a software API and some service tools for configuration and status monitoring. The interface supports Xilinx and Intel in a way that the hardware API and software API are FPGA-type independent.
2. The DeepHealth libraries also require using OpenCL and High-Level Synthesis (HLS) tools provided by the FPGA vendor. It was confirmed by Intel that these vendor tools can be adapted to a custom hardware. Based on this, it is assumed that OpenCL/HLS is supported by this PCI Express board. Some initial tests performed on the MANGO hardware prototype support this assumption.

The board is currently under development. On-going tasks are circuit design, schematic and Printed Circuit Board (PCB) layout. Once the PCB layout is finished, the first boards will be manufactured and initially tested. In parallel, the implementation of the PCI Express controller is in progress. The upstream and downstream communication is up and running for Xilinx FPGAs. The next step is the implementation of the user interrupt and event handling. Finally, the Intel FPGA technology will be added. Both the board and the PCI Express controller are expected to be ready on Q4/2020.

6 Conclusions

This deliverable reports the activities performed so far in the task T3.2 “ECVL adaptation to heterogeneous HPC hardware”. The goal of this task is to deploy, analyse, and decide which are the most suitable strategies to use when porting the ECVL library to heterogeneous HPC platforms technologies, mainly focusing on CPUs, GPUs and FPGAs. The current deliverable will be updated in month 27 in the deliverable D3.4 “ECVL Hardware algorithms and adaptation to HPC (II)”.

Part of the work done focuses on the development of a solid ECVL library that can be automatically tested on different heterogeneous systems. Currently the ECVL library has been successfully deployed and tested on CPU-based systems, while the support for GPUs and FPGAs is under development.

Multiple performance characterization studies have been done on HPC infrastructures with CPUs. As a result, the most time consuming algorithms of the ECVL library have been identified, and their scalability and main bottlenecks have been highlighted. Although the ECVL toolflow does not support GPUs and FPGAs yet, we have already started with the adaptation of some kernels of the ECVL library to FPGA-based systems, and the initial results show that the prototypes for these kernels are functionally correct and ready to be integrated in the ECVL library toolflow. The adaptation and characterization of the kernels to GPU-based systems will start once the ECVL toolflow supports these systems.

In the next months we will focus our attention on GPU-based and FPGA-based implementations of the kernels that have not been adapted yet and on the integration of the different implementations in the ECVL library toolflow. The outcome of this work will be a complete implementation of the ECVL library for different heterogeneous HPC infrastructures, with special attention to the MANGO prototype and the DeepHealth FPGA board. The results of this work will be reported in the deliverable D3.4 “ECVL Hardware algorithms and adaptation to HPC (II)” in month 27.