eFlows4HPC

Enabling dynamic and Intelligent workflows in the future EuroHPC ecosystem

D3.6 Optimised kernels for EPI

Version 1.0

Contract Number	9555558
Project Website	www.eFlows4HPC.eu
Contractual Deadline	29.02.2024
Dissemination Level	PU
Nature	R
Author	BSC
Contributors	Rosa M. Badia (BSC), Enrique S. Quintana-Ortí (UPV), Andrés Tomás (UPV), Jorge Ejarque (BSC)
Reviewer	Fabrizio Marozzo (DtoK)
Keywords	Workflows, application kernels, ARM, RISC-V, SIMD units, EPI, Vector Processing Unit

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955558. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Germany, France, Italy, Poland, Switzerland, Norway.

Change Log

Version	Description Change
V0.0	Draft for internal review
V0.1	Draft revised after internal review
V1.0	Final version, formatted for submission

Table of Contents

1 Executive Summary

As part of the European Processor Initiative (EPI), the Barcelona Supercomputing Centre (BSC) together with other partners is developing a high-performance processor based on the RISC-V architecture and equipped with a vector processing unit (VPU). At present time, the design of the EPI processor is still under development, access to the actual hardware is limited, and the software is still evolving. Compared to the EPI hardware-software ecosystem, ARM provides a good basis for assessing whether the VPU accelerator integrated in the EPI processor can deliver relevant gains for the type of computations that arise in the eFlows4HPC workflows. Therefore, this deliverable describes the work and experimental results with the final versions of the projects' computational kernels optimised for ARM-based architectures equipped with "narrow" SIMD (single-instruction, multiple-data) units. In addition, we take advantage of the recent commercialisation of RISC-V-based boards equipped with narrow floating-point SIMD units to demonstrate the portability of the solutions to this type of architecture. Finally, we evaluate some prototype implementations for two of the project's kernels on an FPGA-emulated version of the EPI processor.

This deliverable describes the development of optimised kernels for DL inference and the singular value decomposition (SVD), collecting the work in task T3.6.

2 Introduction

The European Processor Initiative (EPI) aims to create a high-performance, energy-efficient processor by integrating vector instructions and specialised accelerators with high-bandwidth memory access. This processor will be seamlessly integrated into a System-on-Chip (SoC) with general-purpose cores and vector accelerators based on the RISC-V architecture. The Barcelona Supercomputing Centre (BSC) is playing a key role in the design and development of the EPI, in particular, of the European Processor Accelerator (EPAC).

The ongoing design phase of the EPAC, the limited access to the EPAC hardware, and the evolving nature of the EPAC software led us to focus Task 3.3 on developing prototype kernels for ARM-based architectures equipped with standard (i.e., 512-bit or less) SIMD (single-instruction multiple-data) arithmetic units. Such hardware is available from several vendors, along with an ample collection of well-tested software tools. Fortunately, ARM architectures provide a basis for assessing whether the type of specific accelerator that BSC is integrating into the EPAC can be expected to deliver any performance gains for the type of computations that arise in the workflows. Consequently, the approach in the first half of the eFlows4HPC project (corresponding to the period of Task 3.3) was to design and evaluate prototype kernels to gain insights on whether a substantial improvement could be expected when eventually moving to a vector processor with "deep" arithmetic units.

While the EPI hardware-software ecosystem made considerable progress concurrently with the second half of the eFlows4HPC project (corresponding to the period of Task 3.6), the ARM ecosystem is still much more mature and rich. Consequently, Task 3.6 was primarily focused on developing, optimising and experimenting with the project kernels on ARM-based architectures equipped with SIMD units, yet with a strong focus on portability to facilitate the extension to other processor families.

Interestingly, in the last few months of the eFlows4HPC project, we identified a few new commercial processors implementing the RISC-V instruction set architecture (ISA) and equipped with floating-point narrow SIMD extensions. These baseline realisations of the RISC-V Vector Extension (concretely, RVV-v0.7) thus provided the means to test whether our work on the ARM with an eye to improving portability paid off.

In the final phase of the project, the optimisation of computational kernels for the EPAC still progressed at a moderate pace, basically dictated by the advances in the EPI project. During the last part of the eFlows4HPC project, the EPI ecosystem eventually offered a stable FPGA-SVD (Field Programmable Gate Array-Software Development Vehicle) that emulates the key functional components of the EPAC design. In particular, the FPGA-SDV can now be utilised to emulate a RISC-V superscalar core tightly coupled to a variable-length vector processing unit (VPU) supporting RVV-v.0.7.1. As part of the eFlows4HPC project, we used this hardware-software infrastructure to migrate and optimise prototype implementations for two of the project kernels onto an FPGA-emulated version of the EPI.

The rest of the document is structured as follows. In Section 3 we present a portable library of deep learning (DL) kernels for convolutional neural networks (CNNs) on ARM and RISC-V processors. In Section 4 we review the EPI ecosystem and describe our kernels for DL inference as well as the singular value decomposition (SVD).

3 ARM/RISC-V Architectures with Narrow SIMD Units

3.1 A library of convolution kernels for “narrow” SIMD processors

The most popular convolutional neural networks (CNNs) for computer vision and signal processing tasks contain dozens of convolutional layers which, in many cases, concentrate a large part of the computational cost when training or inferencing the model. It is therefore natural that, in recent years, a considerable effort has been devoted to optimising the convolution operator for virtually every existing computer architecture, from conventional multi-core processors from Intel, AMD and ARM, to hardware accelerators such as graphics processing units (GPUs) from AMD and NVIDIA, and Tensor Processing Units (TPUs) from Google.

In the case of the eFlows4HPC workflows for Tropical Cyclone detection (Pillar II) and Urgent Computing for Tsunamis (Pillar III), an initial analysis determined that a solution based on CNNs provides a computational tool that, from the point of view of accuracy, could be leveraged as an alternative to the numerical simulations appearing in the workflows to reduce the high computational cost of these simulations.

The direct convolution as well as the lowering (also known as the im2col- or im2row-based) approach provide flexible methods to compute the convolution that do not degrade the numerical accuracy, while being based on the general matrix multiplication (GEMM), a kernel that is well-known to provide high performance on current multicore architectures with a layered memory hierarchy.

As part of the work in Task 3.6, we have developed ConvLIB, a high performance, multi-threaded and portable library for computing the convolution on commodity multicore processors. The source code was released in December 2023 in Gitub.[1] The library presents some relevant features:

- The methods comprise three GEMM-based alternatives for the convolution that trade off portability for memory consumption: ConvDirect, ConvGEMM and ConvLow.
- The convolution routines exploit loop parallelism to target a multicore processor as well as SIMD extensions for ARM NEON v8.2 and RISC-V RVV 0.7.1 architectures.
- The library is thread-safe.
- There are Python scripts for the automatic generation of micro-kernels to deal with the edge cases encountered during the execution of GEMM.
- The cache configuration parameters (CCPs) can be configured automatically via an analytical model to adjust the algorithm to the cache hierarchy.
- Between calls to the convolution routines, it is possible to dynamically vary four hyper-parameters: micro-kernel, CCPs, parallel loop, and GEMM algorithm, without recompiling the library.
- The library is augmented with a driver to automatically detect the best values of the hyper-parameters, depending on target architecture and convolution dimensions.

In the following subsections we offer a short overview of the main characteristics of ConvLIB.

3.1.1 Family of micro-kernels

Current implementations of GEMM in high performance open-source linear algebra libraries (OpenBLAS, BLIS) as well as their commercial counterparts (ARMPL, AMD AOCL, etc.) mimic GotoBLAS [Got08] to formulate this computational kernel as five nested loops around two packing routines and a micro-kernel (comprising an additional, sixth loop). This last component is usually encoded directly in assembly or in C with vector intrinsics. At each iteration of this innermost loop, the micro-kernel updates a small micro-tile of the output matrix via a collection of SAXPY (scalar α times x plus y) operations which are mapped into the appropriate SIMD instructions of the target architecture.

The implementation of GEMM in current high performance libraries (BLIS, OpenBLAS, ARMPL, AMD AOCL, etc.) includes a single micro-kernel per architecture, written either in assembly or in C with vector intrinsics. In general, these micro-kernels deliver high performance for large GEMM problems but they are often suboptimal for the size of the operands appearing in the convolutions.

In contrast, as part of the configuration of ConvLIB, the user specifies two SIMD parameters of the target processor architecture via a configuration file: the number of vector registers and their length. The user also indicates in a configuration file the target data type as well as whether the micro-kernels should apply loop unrolling and/or software pipelining. During the installation, the library then takes into account the data provided by the user in these two configuration files to generate a collection of possible assembly micro-kernels.

[1] ConvLIB is available at https://github.com/hpca-uji/ConvLIB.

3.1.2 Cache configuration

In addition to the micro-kernel, the performance of the GEMM routine is strongly determined by the specific values that are chosen for the CCPs. In current high performance implementations of GEMM, these parameters are usually set by an expert to maximise performance for large GEMM problems. However, they can be suboptimal for the type of operations that are associated with convolution layers, especially for non-squarish problems.

ConvLIB employs an analytical model in order to set the values for CCPs. The model takes into account a few key architecture parameters of the cache memory subsystem, namely the number of cache levels as well as their capacity and associativity. Alternatively, ConvLIB allows the user to set these parameters to any other desired value at execution time, by specifying them in the list of parameters in the call to the GEMM routine. This is in contrast with the realisation of GEMM in high performance libraries, for which the parameters are fixed at compile time, and any change requires re-compiling the full contents of the library.

3.1.3 Multi-threaded parallelism

Together with the micro-kernel and cache memory usage, the third aspect with strong impact on the performance on a multicore processor is the parallelization scheme. For an algorithm consisting of a number of loops that concentrate most of the computational workload, with no dependencies, a fair option to attain high performance is to exploit loop parallelism.

Like the selection of the micro-kernel, the CCPs and the GEMM algorithm, the loop to parallelize in ConvLIB can be set dynamically, when invoking the GEMM routine, allowing a layer-dependent customisation of the call.

The implementation of ConvLIB is thread-safe. In particular, it allows to exploit thread-level parallelism within the library components (mainly GEMM). In addition, the library routines can also be invoked simultaneously from two or more application threads. This has some special implications for the implementation of the library that have to be addressed with care because, in order to reduce overhead, the internal buffers are allocated only once, during the first call to the library; and the threads are created once, outside the GEMM loops. Furthermore, the iteration space for the parallel loop is distributed evenly to a certain extent.

3.1.4 “Bestof” driver

While the CCPs can be automatically set by ConvLIB, choosing the best micro-kernel can be challenging for the library user as the optimal option depends on the convolution algorithm, target architecture, problem dimensions, CCPs and, in addition, the number of threads (up to a maximum, specified by the user).

To tackle this caveat, the library includes a driver routine that, given the dimension parameters of a convolution and target number of threads, tests all possible micro-kernels for a given architecture and the three GEMM-based algorithms, identifying the best option for each. The result from this evaluation is summarised into a CSV file containing the best option for each CNN layer.

Experimental overview

The results shown in this subsection were obtained using single precision (FP32) arithmetic for the ResNet-50 v.15 and VGG16 CNNs, combined in both cases with the ImageNet dataset (https://www.image-net.org/) with single input (batch size equal 1). The execution of each case

was repeated for at least 5 seconds and the results were calculated using the average execution times per run.

Table 1 shows the main characteristics of the ARM and RISC-V processors present in the boards targeted in the evaluation and the cache architecture of their respective memory subsystems. In all cases we use one thread per core and the maximum number of processor cores for each platform. When necessary, the frequency is fixed so that it does vary during the execution of the workload.

Table 1. Target boards. Top: Architecture of the processors and RAM. All architectures are equipped with 32 vector registers of 128 bits each. Bottom: Cache architecture for the target processors. NLX, WLX and SLX respectively refer to the number of sets, line size (in bytes) and associativity degree of the LX cache. The cache line size is 64 bytes for all cache levels and architectures.

Board	Processor	Freq. (GHz)	#Cores	ISA	RAM (GB)	Type
NVIDIA Jetson AGX Xavier	NVIDIA Carmel	2.26	8	NEON v8.2	32	DDR4
NVIDIA Jetson Nano	ARM A57	1.43	4	NEON v8.2	4	DDR4
NVIDIA Jetson AGX Orin	ARM A78AE	2.20	12	NEON v8.2	64	DDR5
Sipeed LicheePi 4a	XuanTie C910	1.85	4	RVV 0.7.1	4	DDR4
Sipeed LicheeRv	XuanTie C906	1.00	1	RVV 0.7.1	0.5	DDR3

Processor	N_{L1}	W_{L1} (bytes)	S_{L1} (KB)	N_{L2}	W_{L2} (bytes)	S_{L2} (MB)	N_{L3}	W_{L3} (bytes)	S_{L3} (MB)
NVIDIA Carmel	256	4	64	2,048	16	2	4,096	16	4
ARM A57	256	16	32	2,048	16	2	–	–	–
ARM A78AE	256	16	64	2,048	16	256 (KB)	4,096	16	6
XuanTie C910	512	2	64	1,024	16	1	–	–	–
XuanTie C906	128	4	32	–	–	–	–	–	–

The following implementations are included in the comparison:

- ConvDirect. The direct algorithm for the convolution in ConvLIB.
- ConvLow. The lowering approach in ConvLIB.
- im2row+BLIS/im2row+OpenBLAS. The im2row transform in ConvLIB linked with the implementation of GEMM in either BLIS 0.9.0 or OpenBLAS 0.3.23.
- ConvGEMM. The implementation of the merged im2row+Pack A in ConvLIB.
- ConvWinograd. An optimised implementation of the Winograd algorithm for ARM NEON. This option is available only for the ARM-based processors and convolutional layers with 3 × 3 filters. We observe that the Winograd algorithm usually produces a decay on the numerical accuracy of the results.

For ConvDirect, ConvLow and ConvGEMM, the results correspond to the best micro-kernel, GEMM algorithm, and parallel loop. Furthermore, the CCPs are set using the analytical model. Changing these hyper-parameters is not possible for im2row+BLIS/im2row+OpenBLAS, because they only feature one micro-kernel per architecture. Furthermore, the CCPs are set internally so that modifying them requires recompiling the library to adjust them for each individual layer. For all algorithms, we use the nominal number of floating point operations (flops) for a particular layer in order to calculate the GFLOPS (billions of flops per second) rate.

Although Winograd performs a number of flops that may be lower than this, by normalising the rates to the same flop count, we can directly relate GFLOPS to execution time for all convolution methods.

The results from our experimental evaluation are displayed in Figures 1 and 2 for the NVIDIA Jetson AGX Xavier and the Speed Lichee Pi 4a. Similar results were obtained for the other platforms. Observing the plots there, we make the following general comments about the performance of the GEMM-based algorithms for the convolution on these two boards:

- **NVIDIA Jetson AGX Xavier** (Figure 1). For ResNet-50 v1.5, ConvDirect is clearly superior to the other options for the “leftmost'' layers (C2–C9), (except for C1, for which ConvLow is a better choice,) as well as the “right-most” ones (C17–C20). In the remaining cases, the performance is similar for ConvDirect and ConvLow, yet still superior to that obtained with the BLIS/OpenBLAS alternatives.

  The behaviour is similar for VGG16, with ConvDirect delivering the highest performance for the leftmost layers (C2–C6), again except for C1 where ConvLow is best; and ConvDirect, ConvLow showing similar performance for the remaining layers, superior to that attained with BLIS/Open- BLAS.

- **Sipeed LicheePi 4a** (Figure 2). In this board, ConvDirect offers much higher GFLOPS rates in all cases. We note that BLIS and ConvWinograd have not been ported to this platform and, therefore, we cannot include them in the evaluation with this board.

Because of its different nature, we leave ConvWinograd out of the discussion, yet we include this option in the plots for reference.

The GFLOPS rate offers a normalised metric to evaluate the performance of the distinct convolution algorithms, but it does not expose the contribution of the layers to the total execution time and, therefore, the relevance of the differences. The last experiment, with the results in Figure 3, shows the aggregated time on four of the boards and the two CNN models. The execution times are obtained by integrating the convolution algorithms into the PyDTNN[2] framework for deep learning. We only include one of the RISC-V boards, as the behaviour of the remaining one is similar. Also, since ConvWinograd can be only applied to a few convolutional layers, we exclude it from this final evaluation. We report an additional solution, labelled as “Bestof”, that selects the best among the three algorithms in ConvLIB for each layer.

From top to bottom, the experiment shows that ConvDirect is the best solution on the NVIDIA Jetson AGX Xavier, with a larger gain with respect to other alternatives for VGG16. For the NVIDIA Jetson Nano, ConvDirect is also the best option for about the first half of layers, but from there, a hybrid approach (see the line for Bestof) offers better results. For the NVIDIA Jetson AGX Orin and ResNet-50 v1.5, the differences between all convolu-tion algorithms are very small while, for VGG16 on the same platform, again ConvDirect is the preferred option. Finally, for the Sipeed LicheePi 4a ConvDirect outperforms all other alternatives by a wide margin on both CNN models.

[2] PyDTNN is avaiable at https://github.com/hpca-uji/PyDTNN.

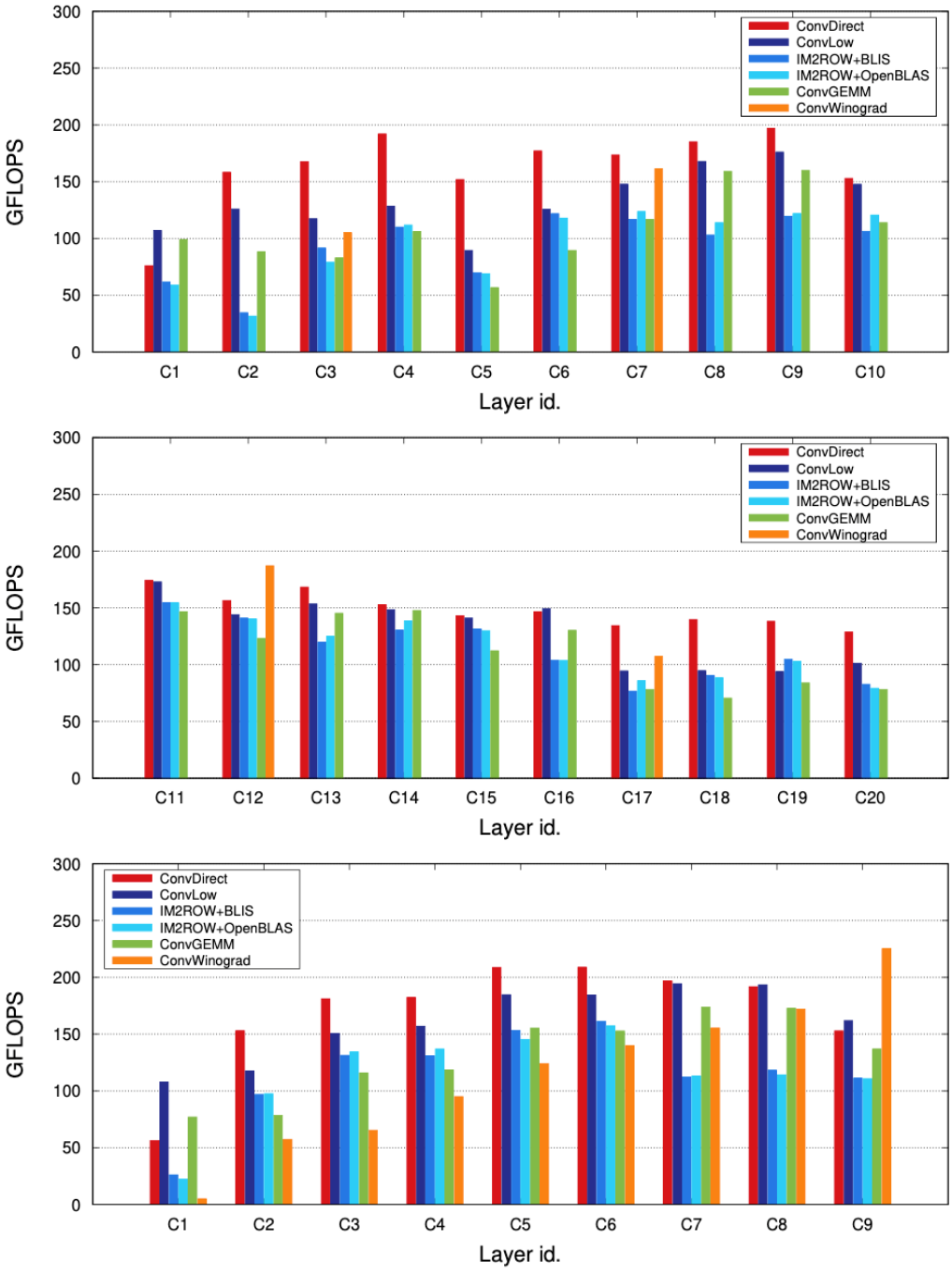

Figure 1. Evaluation of the convolution algorithms on the NVIDIA Jetson AGX Xavier board, using 8 threads/cores. Top and middle: ResNet-50 v1.5; Bottom: VGG16.

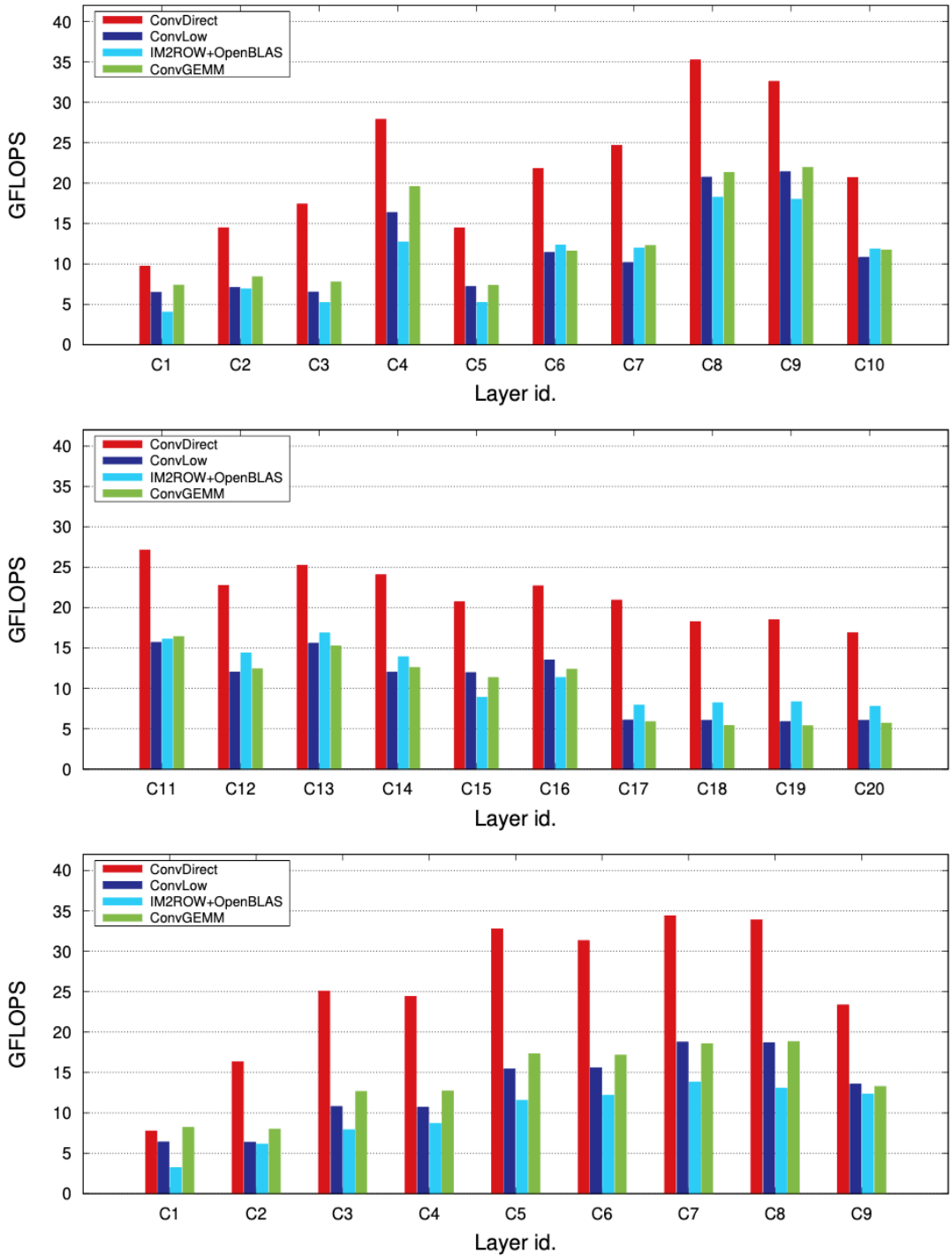

Figure 2. Evaluation of the convolution algorithms on the Sipeed LicheePi 4a board, using 4 threads/cores. Top and middle: ResNet-50 v1.5; Bottom: VGG16.

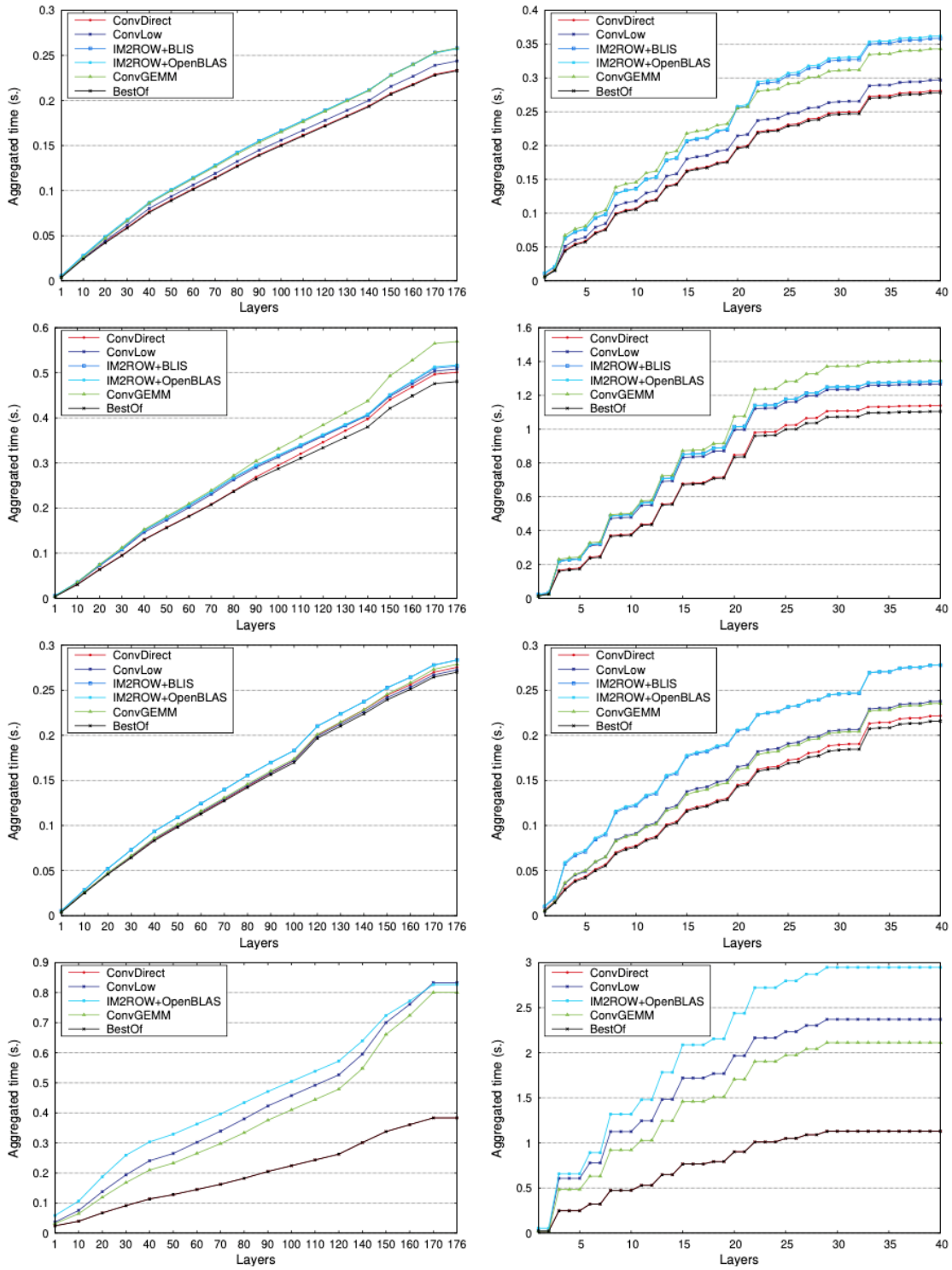

Figure 3. Aggregated execution time (in seconds) of the convolution algorithms. Left: ResNet-50 v1.5; Right: VGG16. From top to bottom: NVIDIA Jetson AGX Xavier, NVIDIA Jetson Nano, NVIDIA Jetson AGX Orin, and Sipeed LicheePi 4a. We use one thread per core on each platform.

4 RISC-V Architectures with Long SIMD Units: EPI

4.1 EPI hardware-software ecosystem

The EPI project is developing a standard general-purpose processor based on ARM and a set of RISC-V based accelerators called EPAC (EPI ACcelerator). EPAC comprises:

- EPAC-VEC: A RISC-V based core tightly coupled with a vector processing unit with vector length of 256 double precision (i.e., 64-bit) elements.
- EPAC-STX: A RISC-V based accelerator focusing on stencil computation.
- EPAC-VRP: A RISC-V based accelerator targeting variable and extended precision computation.

The EPI project has provided a set of hardware/software tools for testing EPAC-VEC. In the rest of the document, we report the experience and results of testing the eFlows4HPC infrastructure on EPAC-VEC prototypes.

4.1.1 EPI software development Vehicle (SDV)

The Software Development Vehicle (SDV) comprises a set of software/hardware tools designed within the EPI project to facilitate experiments and measurements with vectorized codes on EPAC-VEC. The EPI methodology is structured into three levels:

Level 1: Run on a commercial RISC-V platform (scalar CPU). Codes can be compiled using any RISC-V compiler and executed on a scalar RISC-V CPU (e.g., SiFive Unmatched).

Level 2: Run on a commercial RISC-V platform (scalar CPU) + Vehave emulator. This step involves compiling the code with a compiler that supports the vector extension of RISC-V (in its version 0.7.1 or 1.0), enabling auto-vectorization. The binary can be executed using the Vehave emulator to study the achieved vectorization. This level allows refining the code to better exploit the long vectors of the VPU and/or providing feedback to the compiler team for additional improvements in vectorization.

Level 3: Run on RTL mapped into FPGA (FPGA-SDV). After achieving a satisfactory level of vectorization in Level 2, the code can be executed on FPGA-based prototypes implementing the RISC-V vector CPU developed within the EPI project.

The FPGA-SDV platform emulates a single core of the EPAC-VEC from the EPAC accelerator, with the following components:

- The Avispado RISC-V core (SemiDynamics), with support for the RV64GCV ISA.
- The Vitruvius VPU, implementing the RVV-v0.7.1 Extension (BSC and University of Zagreb), and tightly coupled to the RISC-V core.
- A Network on Chip (NoC XP) with 2D Mesh topology (EXTOLL).
- A shared 256-KB L2 Cache tightly coupled with a MESI coherence Home Node, L2HN (FORTH-ICS and Chalmers).

The interconnection of these elements is graphically illustrated in Figure 4 (from Viz23).

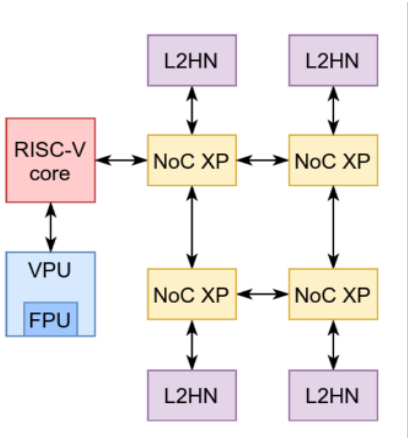

Figure 4. Simplified architecture of the FPGA-SDV, reproduced from [Viz23].

Among these elements, the Vitruvius VPU emulated in the FPGA-SDV is particularly interesting to the eFlows4HPC project, because it contains eight lanes, each equipped with a Floating Point Unit (FPU), which can operate on vectors of 256 double-precision elements (i.e., 16384-bit wide vector registers). The peak performance of the VPU is 32 flops per cycle in single precision and 16 in double precision. Hereby, in principle, the VPU is a very appealing accelerator for scientific computing in general, and dense linear algebra in particular.

4.1.2 RISC-V Vector Extension

The RVV Extension possesses a couple of distinctive characteristics that set it apart from other vector extensions like Intel and AMD's AVX or ARM Neon. Notably, the Vector Length (VL) is not bound by the Instruction Set Architecture (ISA); instead, it is implementation-specific, akin to ARM's SVE extension. An additional noteworthy feature is that the VL can be dynamically altered at runtime.

This flexibility allows an application to execute a segment of code with a specified VL and another segment with a different VL. Consequently, the architecture seamlessly adapts to various application phases without the need for predicates or loop prologues/epilogues. Moreover, this flexibility empowers the compiler to generate code that is agnostic to the VL, eliminating the necessity for binaries to be aware of the machine's maximum implemented VL. This characteristic enhances portability, as the compiled code can be executed on different machines without modification.

4.1.3 FPGA-based emulation board

The FPGA-based emulation platform utilised for the experiments in the eFlows4HPC project consists of an FPGA evaluation board and a host x86 server. The server is a standard commodity system equipped with an AMD Ryzen 5-5600 processor and 32 GB of DDR4-3200 memory, both integrated into a Mini-ITX motherboard. It operates a regular Ubuntu Server 20.04, featuring local storage and a mounted Network Filesystem.

The FPGA board employed is the Virtex UltraScale+ VCU128 FPGA Evaluation Kit, commonly referred to as VCU128. This board incorporates a VU37P FPGA with 8 GB of integrated High Bandwidth Memory (HBM). Additionally, there are five on-board DDR4 memory modules, contributing a combined memory capacity of 4.5 GB.

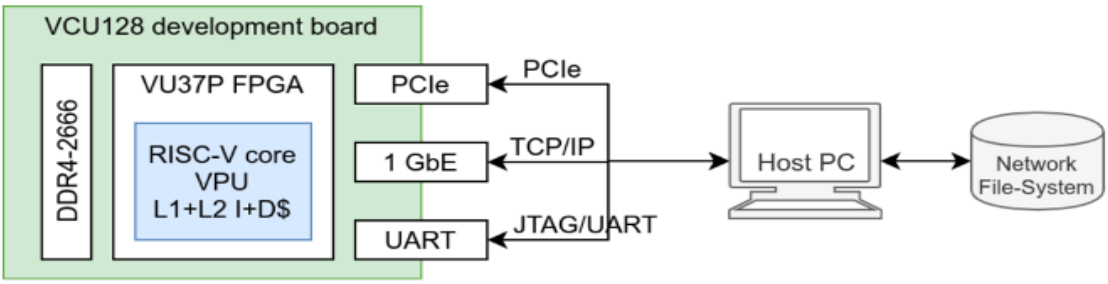

Figure 5. Schematic view of the connection between host server and VCU128 board. Figure reproduced from [Man23].

Figure 5 offers a schematic view of the connection between the host server and the VCU128 board [Man23], via three interfaces: ART, 1Gb Ethernet and a PCI Express bus.

4.1.4 BLAS for the VPU

Concurrently with the eFlows4HPC project, over the past months BSC and Universidad Complutense de Madrid (UCM) collaborated on a separate initiative to develop basic libraries specifically accelerated for the VPU integrated in the EPAC-VEC. This endeavour resulted in a set of fundamental routines for dense linear algebra, commonly known as the BLAS (Basic Linear Algebra Subprograms) [Don90]. Drawing inspiration from GotoBLAS2 [Got08], the package is designed to implement Level-3 BLAS, with a keen focus on optimising the performance of the VPU's lanes and maximising the data streaming capacity of the overall design.

The outcome from this separate effort is a migration of the Basic Linear Algebra Instantiation Software (BLIS),[3] an instance of the BLAS, to run efficiently on the VPU integrated into the EPAC. This software package provides the starting point for the development of efficient implementations of the eFlows4HPC project kernels for DL inference and the SVD on the EPI. Unfortunately, during the experimentation with the eFlows4HPC kernels, we discovered that some of the routines in BLIS are not fully stable and they produced incorrect results under certain circumstances.

In the following subsections, we describe our work to migrate some of the computational kernels in the eFlows4HPC workflows to the EPI.

4.2 SVD on the EPI

The SVD stands out as a crucial numerical tool for low-rank matrix approximation, widely recognized in the field of scientific computing [GolV96]. For the eFlows4HPC project, the SVD holds particular relevance: firstly, the second stage of the reduced order model (ROM) workflow for Pillar I heavily depends on it to compres the original system model into a reduced representation. Secondly, in the realm of data science, dimensionality reduction is gaining prominence. For instance, it proves valuable in preprocessing substantial volumes of information before applying ML techniques for data synthesis. Notably, Principal Component Analysis (PCA) emerges as a swift and versatile unsupervised method for dimensionality

[3] https://github.com/artecs-group/BLIS_EPI_EPAC

reduction. Intriguingly, from a numerical standpoint, PCA essentially boils down to performing an SVD on the input data.

In deliverable D3.4, we provided a comprehensive definition of the SVD, underscoring its significance in DL and the computation of ROMs. Additionally, within the same deliverable, we described several algorithms for the SVD of tall-and-skinny matrices. This type of problem is particularly relevant for the ROM workflow, where the number of rows of the matrix to factorised (about 1 million) exceeds by far its columns (in the order of thousands only). In particular, deliverable D3.4 detailed four SVD-related algorithms:

- Full SVD via Householder reflectors.
- BGS (Block Gram-Schmidt)+CholeskyQR for the QR factorisation.
- Truncated SVD via randomisation (RandSVD).
- Truncated SVD via Lanczos (LancSVD).

For simplicity, we will not reproduce the algorithms in this deliverable, but refer to D3.4 for details. Nonetheless, it is worth emphasising that the analysis of the algorithms in deliverable D3.4 exposed the following:

- The conventional algorithm for computing the full Singular Value Decomposition (SVD) is known for its high cost in terms of flops. Despite its expense, this algorithm is advantageous due to its use of numerically stable building blocks that operate with orthogonal transforms, ensuring full accuracy of the results up to machine precision.
- In contrast, the two truncated SVD algorithms, RandSVD and LancSVD, present a potentially less expensive alternative to obtain a truncated, low-rank matrix approximation. These methods offer the additional flexibility for users to adjust the accuracy of the approximation. It is important to note that, compared to the full SVD algorithm, RandSVD and LancSVD only compute approximations to the largest singular vectors of the matrix, while the full SVD obtains all singular values with full precision. Consequently, the truncated SVD algorithms trade some numerical accuracy for a potentially reduced computational cost. The choice between these methods depends on the specific application and the importance placed on accuracy versus computational efficiency.

4.2.1 Prototype C version

In deliverable D3.4, we assessed the performance of several baseline implementations in C of the SVD-related algorithms, using the building blocks from high performance linear algebra libraries (LAPACK, MAGMA, cuBLAS), for a single-node heterogeneous platform, with one or more multicore processors and accelerated with a single GPU. The work performed in Task 3.6 for the EPI explores the same algorithms for the SVD, commencing with the same baseline codes developed in C as part of Task 3.4.

Figure 6 reports the execution time of two of the SVD algorithms: the BGS+CholeskyQR and the LancSVD, decomposing the cost into their components: matrix multiplications (gemmT, gemmN), triangular system solve (trsm), Cholesky factorization (potrf), square-small SVD (gesdd), and other. (For brevity, we do not report results for the randomised SVD as the results are closely similar.) The implementation of the two types of matrix multiplications and trsm

were specifically developed in the eFlows4HPC project, to deal with the class of tall-and-skinny matrices that appear in the workflow in Pillar I. From the figure, it is clear the reduction of execution time from using the VPU (vector version of the routines) is remarkable when compared with an execution using only the RISC-V core (scalar version).

Figure 7 displays the results from the same experiment in terms of speed-up, to show more clearly the gains obtained from vectorising the code on the VPU. The global acceleration is 6.5x and 5.8x for BGS+CholeskyQR and LancSVD, respectively, which are on par with the speed-ups observed for the most expensive component: matrix multiplication.

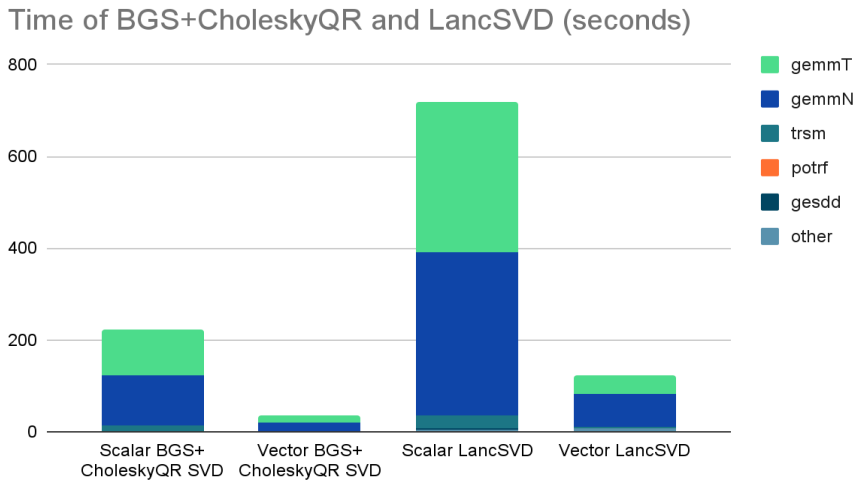

Figure 6. Execution time of BGS+CholeskyQR and LancSVD, comparing the scalar and vector versions of the routines and their components.

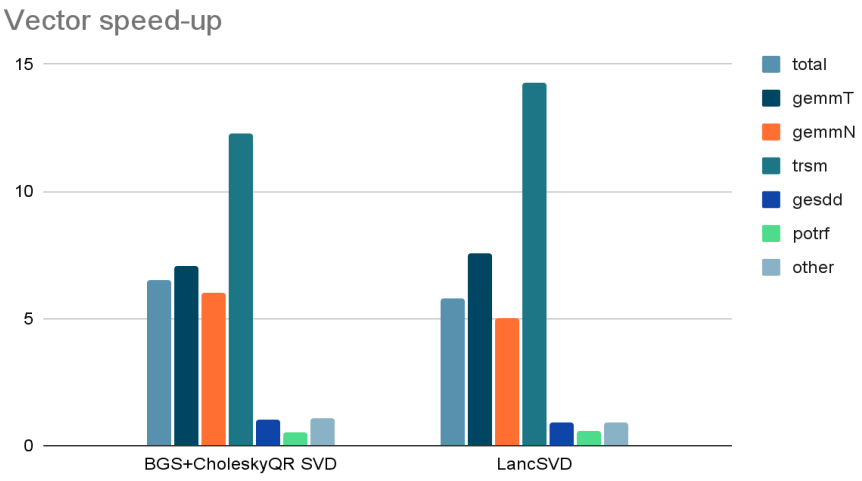

Figure 7. Speed-up obtained with the vectorization for the BGS+CholeskyQR and LancSVD routines and their components.

Using the Extrae+Paraver profiling tools, we could determine that, for example, in the case of LancSVD, the vectorCPI (cycles per vector instruction) is 48.7 for gemmT and 47.3 for gemmN. For reference, the optimal is 32 cycles per vector instruction. Furthermore, the utilisation of the VPU is between 0.85 and 0.81 for the two matrix multiplications. Overall, these are considered as good rates.

4.2.2 Distributed Python version

As we did in the previous section, we have evaluated the distributed SVD-related algorithms presented in D3.4 in the EPI SDV prototype. These algorithms represent one of the kernels of the Pillar I workflow but they are also using PyCOMPSs which is a critical component to run the eFlows4HPC workflow. As introduced in subsection 4.1, the EPI SDV platform emulates the EPI's RISC-V CPU together with the VPU in an FPGA development board which operates at a frequency of 50MHz. Due to the frequency and memory limitations of the platform, we were not able to run large executions, because execution times are considerably long. However this platform is very useful for testing and validating the whole software stack and ensuring it is correctly ported for the RISC-V architecture combined with the VPU.

With this goal, we have ported PyCOMPSs and dislib to the RISC-V ISA. To enable the execution of these tools we have performed the following operations:

1. **Enabling PyCOMPSs:** To successfully run PyCOMPSs workflows in RISC-V, we need a working version of the Python interpreter for executing the workflows and a Java Virtual Machine (JVM) to run the COMPSs runtime**.**
    1.1. **Python support**: In the time that we performed the experiments, a RISC-V ported Ubuntu distribution was available which provided a RISC-V version of the CPython interpreter. We have used this experimental version together with the Python interpreter available in the Ubuntu distribution.
    1.2. **Java support:** The available ubuntu distribution also provided a ZeroVM version of the JVM). The ZeroVM version is a limited JVM which does not include the Just-In-time compiler and other features of state-of-the-art JVMs. However, the OpenJDK had an experimental project to support the porting of the full OpenJDK JVM to RISC-V.[4] We have used this experimental version for our experiments.
2. **Enabling dislib**: It uses numpy and scipy and scikit learn Python modules to perform the kernel computations required by the ML algorithms. These modules rely on BLAS and LAPACK Linear algebra libraries. If we use the standard libraries distributed in the Ubuntu distribution, they are not compiled to use the VPU. To enable the use of the VPU, we have performed the following operations:
    2.1. **BLIS support:** First, we have used a BLIS with support for the vector extensions of the EPI's RISC-V VPU.
    2.2. **LAPACK support:** Second, we have compiled the LAPACK library linked to this modified BLIS
    2.3. **Numpy/Scipy support**: Finally, we have compiled numpy, scipy and scikit-learn python modules indicating the location of these LAPACK and BLIS

[4] https://openjdk.org/projects/riscv-port/

libraries in the compilation configuration. This software stack enables the execution of the dislib's SVD algorithms required by Pillar I using the EPI VPU.

As mentioned above, due to platform limitations, we have executed the algorithm with a smaller data set. For all cases we have used as input a distributed array of 16,000x256 elements distributed in blocks of 4,000x256 elements, and we have performed a single iteration. As we only have a single emulated core, we have used it to run the whole PyCOMPSs workflow. This implies that the master and worker parts both run in the same node. In all the cases, we have measured the total execution time and using a paraver trace, we have also measured the "user code" which excludes the runtime overhead (scheduling, data serialisation, deserialization, etc.). Due to the platform limitations this is larger than the overhead that we expected in a real platform. To validate the correct porting and to measure the effect of using the VPU, we have run the algorithms using the standard libraries with no vector instructions as well as with the stack to support the VPU extensions. We have then compared the execution times of both types of execution to measure the speed-up.

Figure 8 shows the comparison of total execution time of executing the TSQR, Lanczos (*laz* in the figure), and Randomized-SVD (*rand_svd* in the figure) when we run just with the CPU (Scalar) and when we run with CPU and VPU (Vector). We can see that all algorithms are around 50% faster with VPU. If we just compare the user code depicted in Figure 9 we can see that the VPU is about 5 times faster.

Figures 10, 11 and 12 show the execution traces of the three algorithms. Each figure contains the scalar execution in the top part and the vector execution in the bottom part. An execution trace is a timeline representation of what the PyCOMPS process executes at each instant of time. The light-green events represent the time when the application is executing user code of a task. The rest of events represent other task management operations such as the processing of the tasks, importing modules, or serialisation/deserialisation of the task parameters.

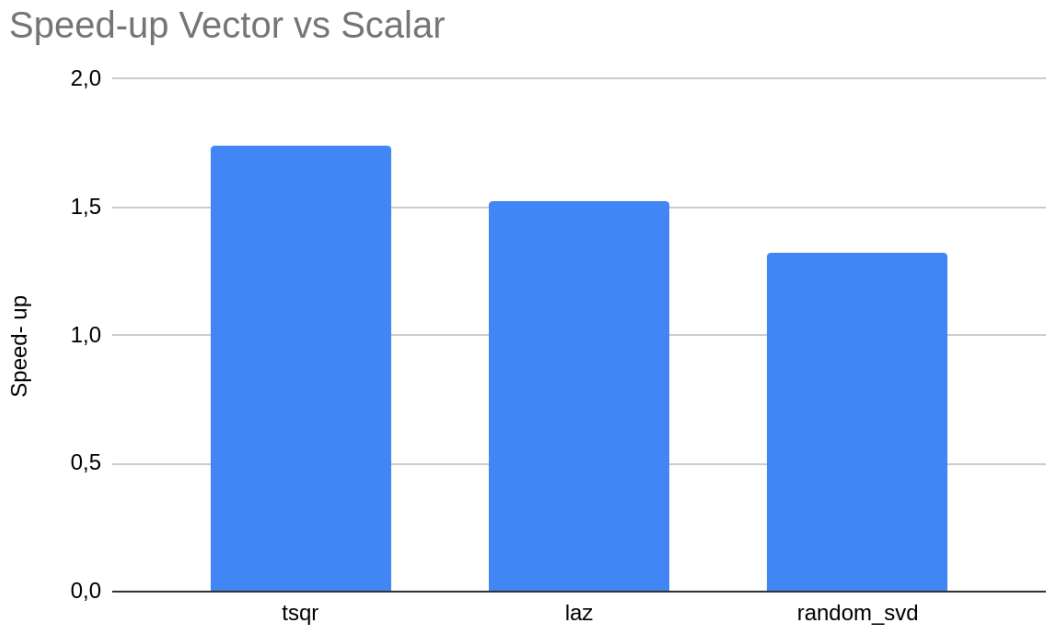

Figure 8. Comparison of total execution time for the different dislib's SVD algorithms with CPU and CPU+VPU.

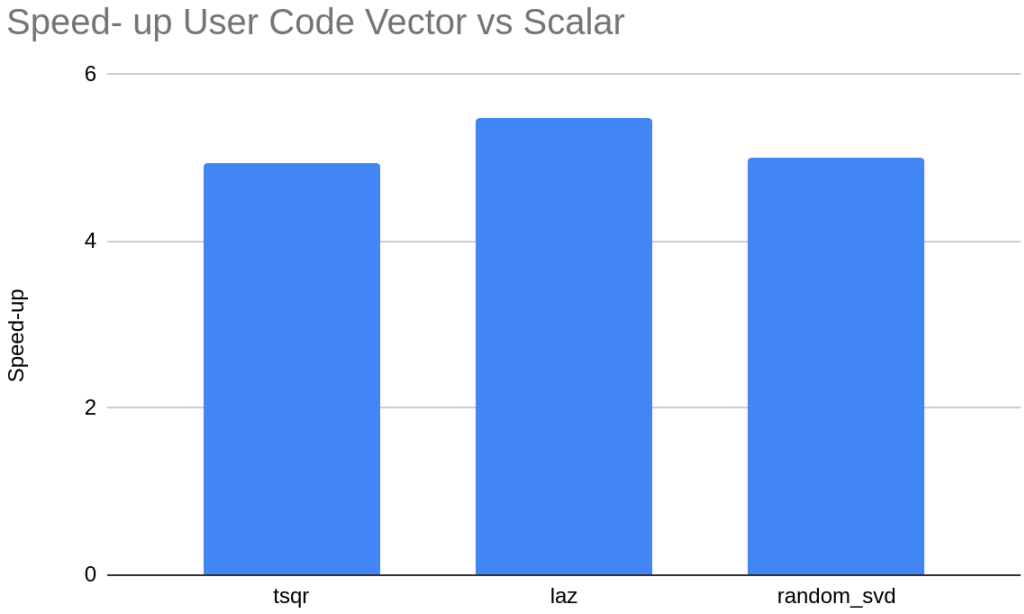

Figure 9. Comparison of user code time for the different dislib's SVD algorithms with CPU and CPU+VPU.

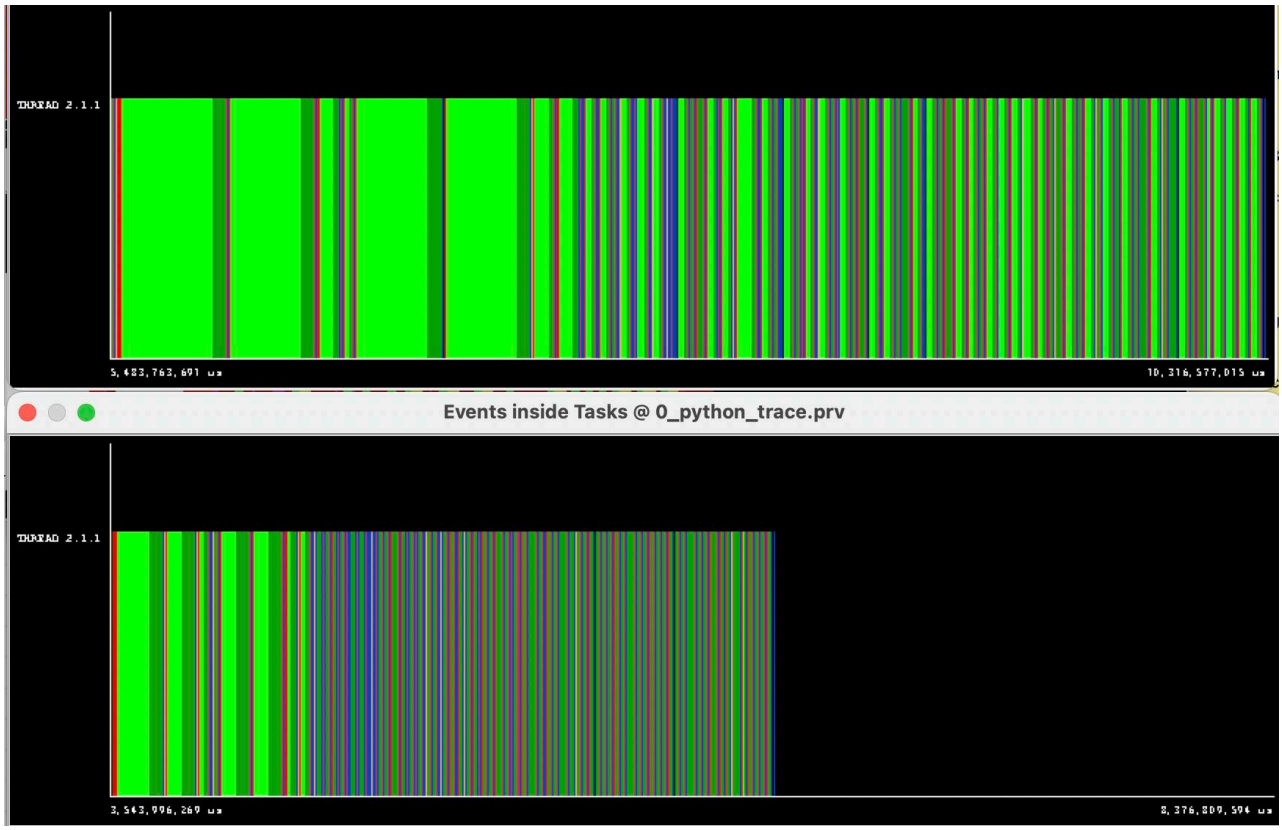

Figure 10. Comparison of execution traces for the TSQR algorithm.

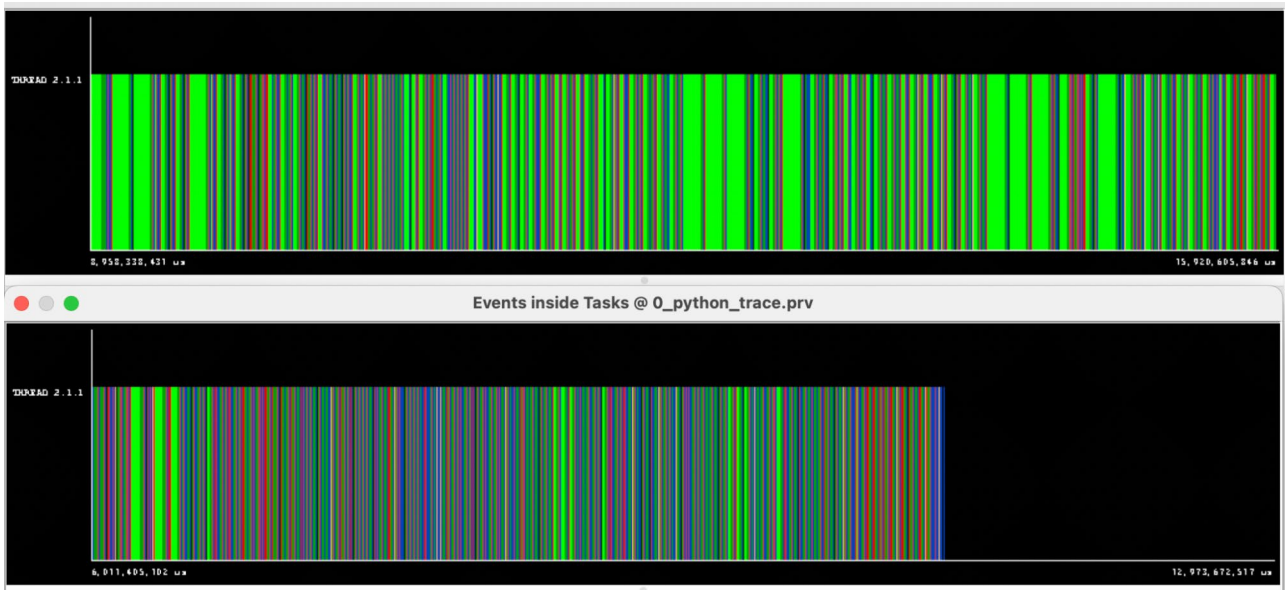

Figure 11. Comparison of Execution traces for the Lanczos algorithm.

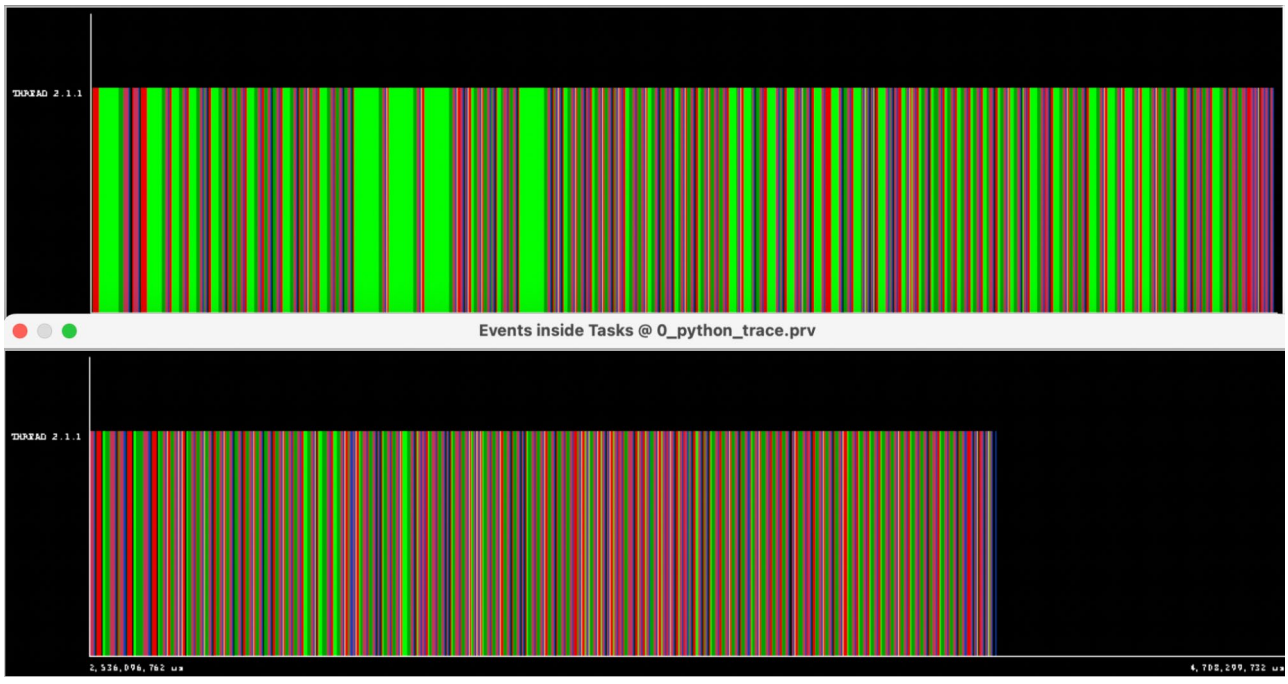

Figure 12. Comparison of execution traces for the Randomized SVD.

These traces visually confirm what we observed in the speed-up figures. In all the traces, we can appreciate a considerable reduction of user code time. In the TSQR, this reduction has more impact on the total execution time because the algorithm has fewer tasks with bigger computations. The other algorithms have a larger number of tasks with smaller computations, so the reduction achieved in the user code has less impact on the total execution code.

4.3 DL kernels on the EPI

For the EPAC-VEC, we decided to migrate ConvLIB to exploit the VPU accelerator in this architecture. In doing so, we had to adopt a compromise between the development effort and expected benefits. Concretely, among the methods in ConVLIB for the GEMM-based convolution, ConvDirect, ConvGEMM and ConvLow, we decided to implement the last one only. The reason is that the former two require specific micro-kernels for the target architecture. Extending our automatic generation tool to produce this type of routines for the VPU is doable but requires major modifications and is expected to attain low performance, given the significant differences between the VPU in the EPAC and a narrow SIMD unit like that in conventional ARM/RISC-V cores.

A critical component that determines the performance of ConvLow is the underlying implementation of GEMM. We initially developed our own routine, based on the ideas underlying GotoBLAS2/BLIS yet taking into account the very long SIMD units and the lack of cache for the VPU. This implementation served us as a reference to determine the level of performance that we could expect from an automatic generator, and was the aspect which made us abandon that option for the EPI. In some detail, attaining high performance from the EPI requires very specific optimisation techniques, which can be only integrated via careful manual coding. As a result, we decided to link the BLIS implementation of GEMM for the VPU as the backend for the ConvLow method in ConvLIB. Although the version of BLIS available at the time of this experimentation was not totally stable, the instance does not produce any errors for the type of matrix multiplications involved in the convolution.

The following results were obtained using single precision (FP32) arithmetic for the ResNet-50 v.15 and VGG16 CNNs, combined in both cases with the ImageNet dataset with single input (batch size equal 1). The execution of each case is repeated for at least 5 seconds and the results are calculated using the average execution times per run.

Figure 13 reports the percentage of the peak performance attained by the ConvLow method for the different layers of the ResNet-50 v1.5 and VGG16 models. (For privacy reasons, it is not possible to report the actual performance using the GFLOPS metric.) The top of the y-axis corresponds to 100% of the peak performance. For reference, we also include the percentage of the peak attained by the implementation of GEMM in BLIS (blue bar). The difference between that and the percentage of the lowering method (red bar) is due to the overhead introduced by the IM2COL/IM2ROW transform. In general, we can appreciate that this overhead is considerable in several of the layers, and requires a special action to accelerate the copies. For some other layers, the overhead is very small. However, this is the case when the implementation of GEMM attains a performance rate that is below 40%. These experiments show that the EPAC-VEC can significantly accelerate the performance of the convolution, but it does require a very careful implementation of the codes, especially one that eliminates non-vectorised codes such as the data copies for the IM2COL/IM2ROW transform.

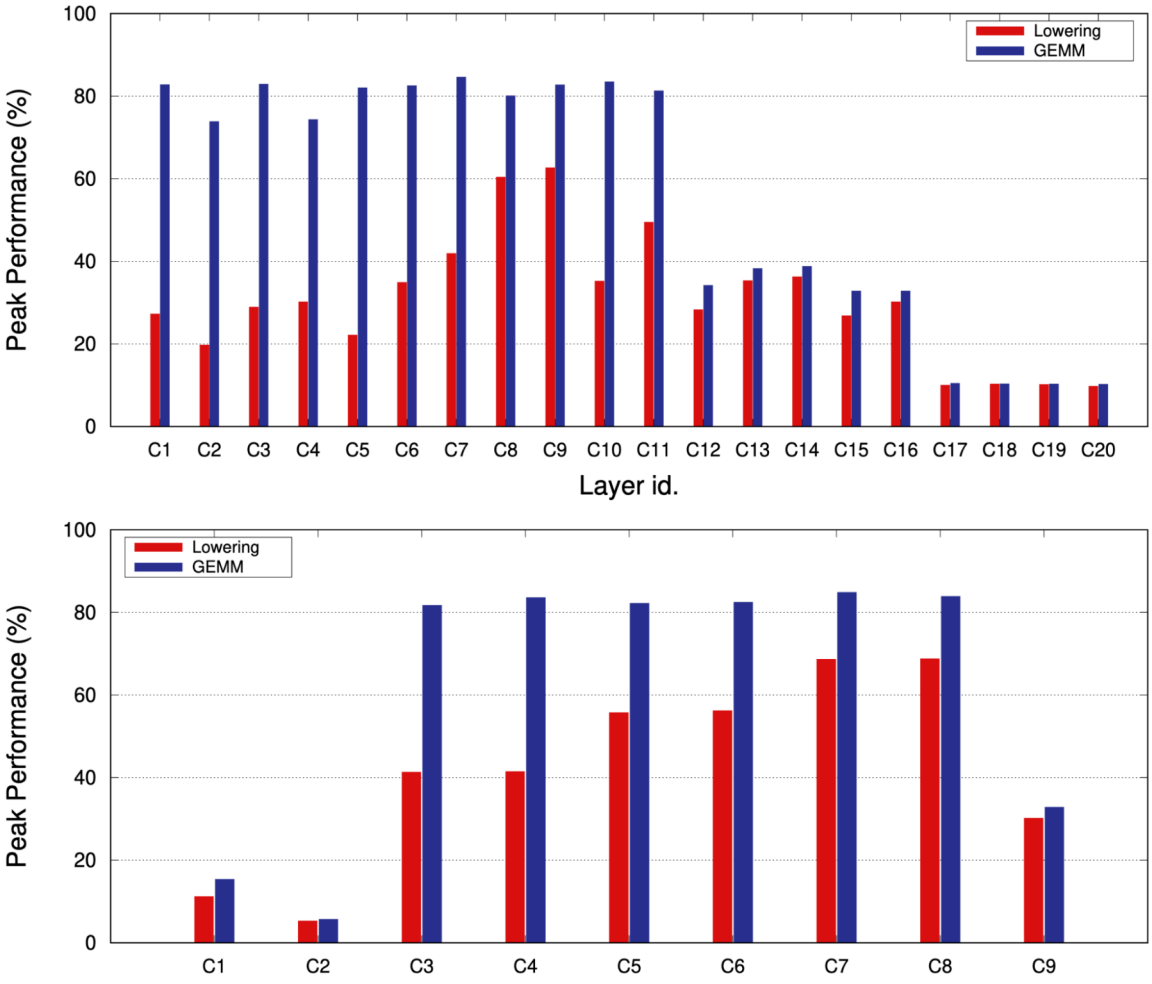

Figure 13. Evaluation of the convolution algorithms on the EPAC. Top: ResNet-50 v1.5; Bottom: VGG16.

5 Acronyms and Abbreviations

BSC: Barcelona Supercomputing Center

BLAS: Basic Linear Algebra Subprograms

BLIS: Basic Linear Algebra Instantiation Software

CCP: Cache Configuration Parameter

CNN: Convolutional Neural Network

DL: Deep Learning

EPAC: European Processor ACcelerator

EPI: European Processor Initiative

flops: floating point operations

FPGA: File Programmable Gate Array

FPU: Floating Point Unit

GEMM: General Matrix-matrix Multiplication

GFLOPS: Gigaflops per second

GPU: Graphics Processing Unit

HBM: High Bandwidth Memory

ISA: Instruction Set Architecture

JVM: Java Virtual Machine

ML: Machine Learning

PCA: Principal Component Analysis

RVV: RISC-V Vector

RAM: Random Access Memory

ROM: Reduced Order Model

SAXPY: scalar α times x plus y

SDV: Software Development Vehicle

SIMD: Single-instruction multiple-data

SoC: System-on-Chip

SVD: Singular Value Decomposition

TPU: Tensor Processing Unit

UCM: Universidad Complutense de Madrid

VL: Vector Length

VPU: Vector Processing Unit

6 Tables and figures

7 References

[Don90] J. J. Dongarra, J. Du Croz, S. Hammarling, I. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. ACM Trans. Mathematical Software, Vol. 16(1), pp. 1-17, 1990.

[GolV96] Golub, G. H. and Van Loan, C (1996). Matrix Computations. The John Hopkins University Press.

[Got08] K. Goto, R. A. van de Geijn. Anatomy of High-performance Matrix Multiplication. ACM Trans. Mathematical Software, Vol. 34(3), pp. 12:1-12:34, 2008.

[Man23] F. Mantovani, P. Vizcaino, F. Banchelli, M. Garcia-Gasulla, R. Ferrer, G. Ieronymakis, N. Dimou, V. Papaefstathiou, J. Labarta. Software Development Vehicles to Enable Extended and Early Co-design: a RISC-V and HPC Case of Study. arXiv:2306.01797v1, June 2023.

[Viz23] P. Vizcaino, G. Ieronymakis, N. Dimou, V. Papaefstathiou, J. Labarta, F. Mantovani. Short Reasons for Long Vectors in HPC CPUs: a study based on RISC-V. arXiv:2309.06865v2, November 2023.