The Journal of Supercomputing (2024) 80:14666–14693
https://doi.org/10.1007/s11227-024-06008-w

POAS: a framework for exploiting accelerator level parallelism in heterogeneous environments

Pablo Antonio Martínez[1] · Gregorio Bernabé[2] · José Manuel García[2]

Accepted: 18 February 2024 / Published online: 25 March 2024
© The Author(s) 2024

Abstract

In the era of heterogeneous computing, a new paradigm called accelerator level parallelism (ALP) has emerged. In ALP, accelerators are used concurrently to provide unprecedented levels of performance and energy efficiency. To reach that there are many problems to be solved, one of the most challenging being co-execution. In this paper, we present a new scheduling framework called POAS, a general method for providing co-execution to applications. Our proposal consists of four steps: predict, optimize, adapt and schedule. With POAS, an unseen application can be executed concurrently in ALP with little effort. We evaluate POAS on a heterogeneous environment consisting of CPUs, GPUs (CUDA cores), and XPUs (Tensor cores) on two different fields, namely linear algebra (matrix multiplication benchmark) and deep learning (convolution benchmark). Our experiments prove that POAS provides excellent performance and completes the tasks within a time very close to the optimal time for the hardware and applications used, with a negligible execution time overhead. Moreover, the POAS predictor performed exceptionally well, achieving very low RMSE values for both use cases. Therefore, POAS can be a valuable tool for fully exploiting ALP and improving overall performance over offloading in heterogeneous settings.

Keywords High-performance computing · Heterogeneous computing · Accelerator level parallelism · Scheduling · Co-execution

✉ Pablo Antonio Martínez
pablo.antonio.martinez@huawei.com

Gregorio Bernabé
gbernabe@um.es

José Manuel García
jmgarcia@um.es

[1] Huawei Technologies Research and Development, Huawei, Cambridge, United Kingdom

[2] Computer Engineering Department, University of Murcia, Murcia, Spain

1 Introduction

In recent years, it has been demonstrated that CPUs are less efficient regarding power consumption and performance compared to accelerators [10]. After the end of Moore's law [11], computer architecture is evolving into the heterogeneous era, where accelerators are used to accomplish different tasks, instead of relying on the CPU for all of them [46]. Accelerators are typically used in fields like machine learning, where many accelerators exist, like the tensor processing unit (TPU), the neural processing unit (NPU), etc. Other notable accelerators include image signal processor (ISP), digital signal processor (DSP), or video encoders/decoders [37].

In fact, some authors say that the next computer architecture paradigm is the accelerator level parallelism (ALP) [15]. This new kind of parallelism seeks to execute workloads in multiple accelerators concurrently, thus exploiting parallelism at the accelerator level. However, co-execution in heterogeneous environments is challenging since the software needs to divide the work into parts and schedule them among radically different devices. In this context, the scheduling may pursue different objectives, like minimizing the execution time, the energy consumption, or both [41]. In either case, achieving it depends heavily on the target hardware platform.

This article presents POAS (Predict, Optimize, Adapt and Schedule), a framework for scheduling an application to run concurrently on multiple accelerators, which can potentially minimize the execution time of a given workload.

To demonstrate how POAS works, we apply our method to two relevant case studies. First, matrix multiplication, one of the centric linear algebra operations, which is present in uncountable HPC applications. Second, convolution, the heart of convolutional neural networks (CNNs), which is one of the most representative metrics for inference performance in low-energy SoCs and training performance in HPC servers. We implement POAS as a framework that runs matrix multiplication and convolution workloads in ALP, supporting multi-core CPUs, GPUs and XPUs (tensor cores), an accelerator for matrix multiplication and DNN workloads.

Unlike previous works that offload workloads to one device at a time, POAS aims to execute one single task in many accelerators concurrently. POAS can be applied to any application as long as it is possible to predict its execution time as a function of the input size. In its current state, the framework cannot be applied to applications where predicting the execution time is not possible. However, we are exploring alternatives for extending the framework to support these application types.

Compared to related works, POAS novelty comes from the fact that it is:

- Application independent: Previous works have already studied scheduling in heterogeneous scenarios, but most of them are application-dependent (like [22]). However, POAS can be applied to any application as long as it is possible to predict its execution time as a function of the input size.

- Designed for ALP: Unlike previous works that offload workloads to one device at a time, POAS aims to execute one single task in many accelerators concurrently.
- Accelerator agnostic: Previous works typically are tied to a specific number and type of device (e.g., CPU/GPU environments), while POAS can be extended to any number and kind of accelerators.
- Flexible: Unlike previous works that focus only on execution time, POAS can minimize execution time and/or energy consumption.
- Middleware: Ideally, POAS could be implemented like a middleware at the OS level, similarly to how Intel Thread Director [17] works.

Furthermore, experimental results highlight that POAS can exploit ALP with negligible overhead, reaching near-optimal results. Combined, this makes POAS an excellent candidate to reach ALP in current and future generation computing systems.

The main contributions of this paper are:

- Defines a novel framework for exploiting Accelerator Level Parallelism (ALP) in heterogeneous environments. The framework is based on a new scheduling model that uses a performance predictor together with the definition and optimization of a mathematical model.
- Details how the proposed framework works in two real-world applications (matrix multiplication and convolution).
- Presents an experimental evaluation of the proposed framework in an ALP environment (CPU, GPU and XPU).

The rest of the paper is organized as follows. Section 2 presents the background in scheduling and co-execution state-of-the-art techniques, as well as related work in heterogeneous matrix multiplication and convolution approaches. In Sect. 3, we present POAS, our framework for allowing co-execution in heterogeneous environments. We detail how POAS works in real-world applications like matrix multiplication and convolution in Sect. 4. A performance evaluation of POAS is shown in Sect. 5. Finally, Sect. 6 concludes the paper and gives some hints for future work.

2 Background and related work

2.1 Accelerators and tensor cores

Accelerators are hardware devices that execute a given workload in less time and/or with higher energy efficiency than conventional CPUs [10]. Nowadays, GPUs are the mainstream, easily accessible accelerators for the masses. While they accelerate many relevant workloads (like machine learning) [9], they are still generic enough for many domains. However, there is a trade-off between efficiency and generality, so GPUs are usually less efficient than more specific accelerators. FPGAs, for example, can be adapted to different domains thanks to their re-programmable hardware, but

they are particularly difficult to use. Lastly, application-specific integrated circuits (ASICs) are designed and built for specific applications, so they achieve the highest levels of performance and efficiency [37]. Accelerators are common in popular domains like machine learning. The well-known tensor processing unit (TPU) [21] accelerates both inference and training workloads. In the area of matrix multiplication, accelerators supporting dense and sparse products [2], as well as sparse-only matrix multiplication [36], exist.

Tensor cores [8, 20] are domain-specific cores designed to enhance matrix multiplication performance, which ultimately boosts deep learning applications. They were included for the first time in the Nvidia GPU Volta microarchitecture. In Volta, tensor cores implement a 4x4x4 FP16 matrix multiply and accumulate instruction, HMMA (half precision matrix multiplication and accumulate) [20]. The Tensor cores in the Turing microarchitecture add support for `int8`, `int4` and `int1` data types [19] through a new IMMA instruction. Finally, in the Ampere microarchitecture, the matrix multiplication size changes from 4x4x4 to 8x4x8, doubling its FP16 throughput [9]. It also adds new instructions for sparse matrix multiplication, which in turn doubles the throughput of dense matrix multiplications.

Tensor cores boost specific applications' performance in an unprecedented way, providing a 4x boost in peak performance compared to CUDA cores, and 8x for the case of sparse matrices [9].

2.2 Scheduling

Task scheduling algorithms have been applied successfully to exploit scenarios where multiple tasks have to be scheduled to different processing elements [24, 47, 48]. Within the same node, scheduling can be divided into two different approaches: offloading and co-execution. In offloading, the idea is to enhance application performance by offloading the compute-intensive part to specialized hardware devices [1, 27]. To decide on which device the workload should be offloaded, previous works studied the performance of each device and selected the best fitting for this task. Unlike task scheduling and offloading, co-execution aims to distribute a single application among different devices and run all of them concurrently.

Task scheduling techniques have been proposed for OpenCL kernels in [48], where authors use both code's features as well as runtime ones to predict the speedup of applications in CPU or GPU. Also in OpenCL, non-analytical methods like decision-based trees are used in [47] to schedule OpenCL kernels on CPU/GPU platforms. Co-execution opportunities are studied in [50] on integrated CPU/GPU architectures. They also studied how to determine which compute elements are suitable or not for a given task (in other words, when co-execution is beneficial or not). List scheduling has been applied in static [51] and dynamic runtime scenarios, where new workloads arrive over time [24]. Profiling and machine learning were combined in [14] to provide scheduling in heterogeneous environments. Integer linear programming (ILP) and linear regression were combined with stream graphs in [28] to efficiently distribute workloads on multi-GPU platforms.

Performance modeling has been widely applied in many works [13, 34, 38, 44]. In a DynamIQ heterogeneous multi-core environment, a performance model to estimate the efficient distribution of critical sections was designed [34]. Task scheduling has been often applied to CPU/GPU environments, but there are also other approaches for more heterogeneous environments, like CPU/FPGA [42]. In [49], authors proposed a scheduling strategy for distributed accelerator-rich environments centered in real-time applications. The predictable execution model (PREM) [38] was proposed to enable time prediction on non-predictable hardware. The approach separates programs into memory and computing phases, which can be independently scheduled. It was proposed for CPU only, but a recent work extended it for CPU/GPU architectures [13]. Many of these works focused primarily on minimizing execution time, while others studied energy consumption. Although the latest is often harder to predict, there are some promising works in this field [12]. Given the heterogeneous nature of today's computing systems, other studies considered both execution time as well as consumption in their scheduling decisions [39, 41].

Recently, several works have focused on designing frameworks and systems to co-execute applications without domain-specific information. Many are often targeted to specific frameworks or languages that enable single-source coding on heterogeneous platforms. A language that is gaining influence lately is oneAPI [18]. oneAPI, as well as other heterogeneous languages, typically achieves good performance in relevant applications like DNNs [25, 26]. However, oneAPI does not officially provide a mechanism for scheduling or co-execution. In a recent research [30], authors proposed a new co-execution runtime in oneAPI based on load-balancing algorithms. Another relevant framework in this context is OpenCL, coupled with a co-execution engine in [29]. In [40], authors extend the OmpSs framework to allow co-execution of OpenCL kernels. Lastly, a Python-based heterogeneous scheduler was proposed in [23], with similar objectives to what POAS pursues. It uses task parallelism and a queue-based approach to schedule programs in multi-GPU environments.

Finally, there are also domain-specific scheduler proposals. Among them, scheduling proposals for general matrix multiplication have been deeply studied over time, mainly due to their high relevance in many computer science applications. Recent works have studied the performance of matrix multiplication in heterogeneous environments [43]. Furthermore, several papers have considered the use of different hardware devices to compute matrix multiplications to exploit heterogeneous systems. One of the first studies [4] already approached the problem from an analytical point of view. The authors analyzed the computational power of each processor in the heterogeneous system and later expressed the workload distribution as an optimization problem. In [6], authors designed a hierarchical approach to distribute parts of the matrix multiplication to different devices. When considering multiple accelerators and a range of n columns to be assigned to each accelerator, the search space becomes too big. Therefore, they proposed a hierarchical way of considering all the possibilities, significantly reducing the search space. A new algorithm based on Strassen's method was presented in [22] for heterogeneous environments. To schedule the work between

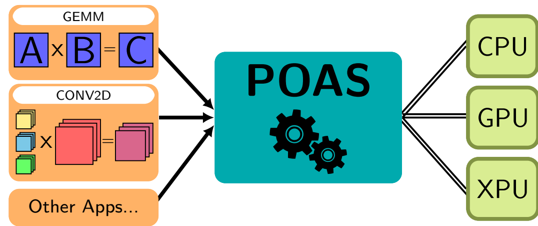

Fig. 1 POAS operation overview. The framework takes different applications and executes them in co-execution, providing ALP

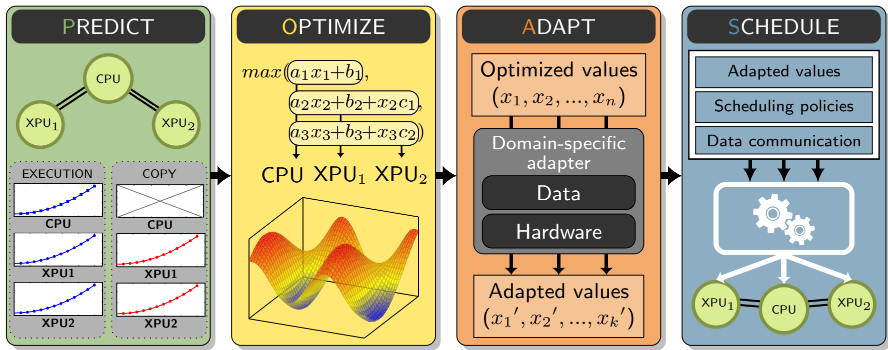

Fig. 2 Detailed overview of POAS (Predict, Optimize, Adapt and Schedule) framework

accelerators, a queue-based system was used, which gives blocks of the matrices to be computed whenever a device is free. Matrix multiplication workload distribution has also been studied in the context of energy efficiency [7], where authors proposed an approach for ARM big.LITTLE processors.

3 Predict, optimize, adapt and schedule (POAS)

In this section, we present POAS (Predict, Optimize, Adapt and Schedule), a framework that can schedule any application to be executed in ALP environments.

Figure 1 depicts a general view of our framework, which takes one application and executes it in ALP, improving the application performance. The framework is divided into four phases (*Predict*, *Optimize*, *Adapt*, and *Schedule*), which must be performed in order. The first one, predict, consists of developing a prediction model of the execution time of the CPU and the accelerators, as well as the memory cost to copy the data between the CPU and the accelerators. In the optimization step, the performance prediction model is used to build a constraint satisfaction problem (CSP). The problem is then optimized to find the values so that the objective function is minimal. Lastly, the results given by the solver may need to be adapted so the scheduler can use them in the last step of POAS.

Figure 2 shows a detailed view of POAS. All phases are mandatory except for the *Adapt* phase, which is optional. Likewise, the output of each phase is the input of the next one, as Fig. 2 shows.

Fig. 3 Analysis of the four POAS phases

	PROBLEM AWARE?	PLATFORM AWARE?
PREDICT	N	Y
OPTIMIZE	Y	Y
ADAPT	Y	N
SCHEDULE	N	Y

The *Predict* phase must be tuned to pursue one of those goals. One of the main strengths of POAS is its ability to provide ALP in a generic way. Rather than achieving ALP through a set of hardcoded, domain-specific constructs, POAS is built of different steps that are flexible enough to be used for diverse applications. At the core of this generality is the division of the prediction phase in two: the prediction itself and the optimization. As Fig. 3 shows, the *Predict* phase is problem agnostic because it does not consider domain-specific information to be built.

Likewise, the *Optimize* phase is both platform and problem-aware because the problem formulation must reflect the behavior of the problem but also consider hardware peculiarities. This decoupling scheme allows for flexible scheduling that does not depend on the problem.

Lastly, the *Adapt* phase depends on the problem, whereas the scheduling is only aware of the hardware platform.

It is worth mentioning that, like other scheduling approaches, POAS is designed for scenarios where there is a significant amount of work to do. If not, ALP would not provide substantial gains over the execution on a single device.

3.1 Predict

In the predict phase, a performance predictor is designed, and the profiling of the hardware platform is performed.

3.1.1 Predictor

The goal of the prediction is to give a precise estimation of the execution time of the application. This prediction is software and hardware-dependent, so the prediction must consider both application and hardware characteristics. POAS is a modular framework, so any performance prediction method can be chosen in this phase. There are many performance prediction approaches, and depending on the domain, one predictor would be more suitable than the others. The POAS framework could implement different predictors that would be used depending on the application. Furthermore, in the case of performance prediction, the performance model must predict both the execution time and the time spent in memory transfers between the CPU and the accelerators over the bus. The only requirement for the performance predictor is to provide a function that, given the input size, predicts the execution time of the application. While the resultant function has no restriction regarding its complexity, it is desirable to have a linear or quadratic function, as discussed in

Sect. 3.2. Regression or similar methods can be used for computing the function from the measured values in the profiling. To achieve competitive performance, the accuracy of the predictor is vital. If the prediction fails to precisely reproduce the experimental results, the scheduling would be poor. At this point, it is worth noting that with POAS it is not mandatory to have the source code, which makes the framework more flexible since it does not depend on the programming language, which is a limitation in many language-centered models.

3.1.2 Hardware profiling

As part of the prediction phase, a profiling of the hardware platform is also necessary. With profiling, the hardware is sampled with different input sizes, and time is measured to build the function that maps the input size into execution time. One key aspect before profiling is to study the behavior of the hardware executing the application because sometimes the hardware provides different performance results depending on data sizes, alignment, and other factors. For example, in matrix multiplication, tensor cores only provide optimal performance if $m\%\ 8 == 0$ and $k\%\ 8 == 0$ [31] (where m and k are matrix dimensions).

3.2 Optimize

The optimization phase takes the prediction model generated in the previous step as input. This phase has two objectives: to define a formulation of the application's behavior and to optimize it. The output of this phase is a set of optimized values, which typically represent the input size of each device, such that the desired objective function is optimized.

3.2.1 Formulation of the problem

The formulation is expressed as a constraint satisfaction problem (CSP), which can be enunciated to achieve different goals. In many cases, however, the problem can be further specialized into a constrained-optimization problem (COP), which is a generalization of the CSP. It is crucial that the mathematical formulation models all the details of how the application works in the real world (i.e., when the compute and communication phases occur and how). The formulation of the problem is the only manual part of the whole framework since it is application-dependent. Depending on the application, communication schemes, and other factors, different applications may need different formulations. Likewise, one formulation might be reused for many applications if they behave similarly.

3.2.2 Optimizing the problem

Regarding methods for optimizing the model, linear or quadratic programming can be used, providing the optimal solution in very little time. However, these methods can only be used if the function that models the behavior of the application is linear

or quadratic. Considering that there might be cases where the performance model is too complex to be represented in these terms (e.g., the function is cubic), the problem should be formulated as a CSP. In this case, alternative methods (like backtracking, local search, etc.) can be used to optimize the performance model. The POAS framework implementation can provide different solvers, which would be used by the appropriate application.

3.3 Adapt

This is the only optional phase of the POAS framework. Depending on the application, the variables that come from the optimized model designed in the previous phase might need some transformations to be used by the scheduler. Therefore, an intermediate phase called adapt might be needed to make the scheduler work correctly. The output of this phase is always a set of valid values to the scheduler. If the input of the adapt phase is a valid input to the scheduler, data are left unmodified, and the adapt phase is a no-op. Otherwise, the adapt phase performs an adjustment of the data. We differentiate between two types of adjustments: data and hardware adjustments.

3.3.1 Data adjustments

This kind of adjustment is needed when the output of the optimized model contains different variables than the one needed to determine how to schedule the application. In these cases, the adapt phase must adjust the values given by the optimization phase to some values that can actually be used in the scheduler. For example, let's say that the scheduler needs the number of elements to be computed by each device within a vector, but the output of the optimize phase is the start and the end of the portion of the vector to be computed. Data adjustments depend on the application so this procedure is essentially application-dependent.

3.3.2 Hardware adjustments

Generally speaking, hardware is very sensitive to data sizes and other factors, so performance might vary depending on the input size. This is very harmful to prediction accuracy and is something that must be solved in this phase. The goal of hardware adjustments is to ensure that the input of the next phase (scheduling) matches the same performance conditions as the previous phase (profiling).

For example, let's consider the case of tensor cores. As we discussed, tensor cores typically perform differently depending on the size of m and k. Let's say that we perform the profiling phase assuming that input sizes will always be multiple of 8, the best-case scenario. However, the optimized values m and k given by the solver do not have to be a multiple of 8. This phase takes care of these low-level details, which are key for high-quality prediction accuracy.

It is worth noting that the goal is not to capture all the aspects of both hardware and software that have an impact on performance, but to capture only the hardware

characteristics that are not captured by the prediction phase in POAS. For example, the impact of the memory hierarchy or the use of shared memory in GPUs is directly captured by the predict phase and therefore does not have to be considered here.

3.4 Scheduler

Within the POAS framework, different scheduling policies can be implemented. In this work, we only consider a static scheduling approach. The static scheduler uses the performance model and optimizes the problem formulation to get the optimal inputs for each device. Other scheduling policies (for example, dynamic scheduling) are left for future work. The scheduler policy must also include how to manage the communications between the CPU and the accelerators, which might have a significant impact on performance. The framework might implement different schedulers that work better or worse for different applications, allowing users to select the best scheduler for each case.

3.4.1 Data communication scheme

In work distribution, the effective use of the memory bus is a performance crucial aspect. In ALP environments (like SoCs), accelerators are usually connected to a shared bus, where all of them can communicate with the CPU. Hence, optimizing applications for exploiting ALP is challenging since the bus (thus, the throughput) must be shared among the accelerators.

As a first approach, we propose a scheduler based on priority scheduling. The idea is to assign a priority to each device connected to the shared bus. Then, data are copied to/from the CPU in the order dictated by the priority ordering. There are many approaches to designing this scheme with different goals, like minimizing the idle time of accelerators. We leave for future work to further investigate more efficient approaches.

4 Scheduling GEMM and convolution with POAS

This section details how POAS can schedule real-world applications. Hence, this section should not be considered as an attempt to find the optimal prediction or scheduling methodology for GEMM or convolution. The goal of this section is to show how the framework works, so its main focus is not on the particular performance prediction approaches or schedulers used. First, we detail all the phases for matrix multiplication. Later, in Sect. 4.5, we highlight only the differences between GEMM and convolution, since most of the workflow in POAS for GEMM and convolution remains the same.

We designed a POAS implementation focused on minimizing the execution time, targeting CPUs, GPUs and tensor cores (from now on, XPUs). The implementation relies on optimized libraries to perform the matrix multiplications: MKL (in Intel

CPUs), BLIS (in AMD CPUs) and cuBLAS (for both CUDA and tensor cores). For convolution workloads, our implementation relies on oneDNN (in CPUs) and cuDNN (in GPUs).

4.1 Predict (GEMM)

4.1.1 Linear regression

To design the performance predictor for GEMM, we used a regression analysis approach. It is well known that GEMM general algorithm has a complexity of $O(n^3)$. But to use linear regression, we must find a way to represent the time with linear complexity. Thus, we model the execution time with the number of operations (from now on, *ops*), such that $ops = m * n * k$, where m, n and k are the matrix dimensions. In other words, the execution time grows with a cubic complexity if we consider the input size, but it grows linearly considering the number of operations.

While this linear function can generally predict the performance of GEMM, there are certain hardware peculiarities which might cause the prediction to fail. For example, the XPU will provide radically different results depending on the input size of the matrix, as the tensor cores can only be optimally used when the input meets some criteria. To eliminate ambiguity, the performance predictor always assumes the best case (e.g., in the case of tensor cores, it assumes that input size meets the criteria that give the best performance). Therefore, one additional task in the adapt phase is ensuring that real workloads can be computed in the same way that the predictor was trained for. We further contemplate these details in Sect. 4.3. In addition to the compute times, we also predict copy times between CPU and GPU.

4.1.2 Profiling

We perform a profiling step of the hardware platform, which is done only once at installation time and takes less than five minutes to complete. The profiling phase measures the computing power of all the hardware devices available in the system and the memory bandwidth between the CPU and the accelerators. Then, the results are stored in a text file that is read when real matrix multiplication workloads arrive. To improve prediction accuracy, we profile the performance of squared matrix multiplication only, rather than profiling many different matrix shapes. Restricting the profiling space can improve prediction significantly since the range of predicted inputs is smaller. Then, when a big, non-square matrix computation arrives, POAS divides the matrix into a list of squared matrices, which are equivalent to computing the whole matrix at once. (We detail the slicing algorithm in Sect. 4.3.1.) Using this approach, we predict the performance of all matrix shapes precisely. Therefore, the profiling phase consists of two steps:

- Computing power profiling: The program runs a set of squared matrix multiplications (using appropriate libraries like MKL, BLIS or cuBLAS). The sizes of the squared matrices are variable and adjustable depending on the device (see

Sect. 5.1.3 for more details). When all the experiments have finished, linear regression is performed to obtain the linear function that models the execution time of the device.
- Memory bandwidth profiling: The program runs a microbenchmark that measures the bandwidth between the CPU and each accelerator.

4.2 Optimize (GEMM)

In the optimization phase, we formulate a constraint satisfaction problem (CSP) that minimizes the execution time. Therefore, the goal of the solver is to find a distribution of *ops* among the hardware devices such that the total execution time is minimal.

4.2.1 Problem formulation

We express the execution and copy times as a mixed-integer linear programming (MILP) problem. We define c_x as the independent variables, which represents the number of operations (*ops*) to be computed by device x. We also define y_x as the function that gives the time to copy A, B and C matrices. The goal of the solver is to minimize the following objective function (which models the total execution time of the GEMM in n devices):

$$max(t_{c_1} + t_{y_1}, t_{c_2} + t_{y_2}, \dots , t_{c_n} + t_{y_n}) \tag{1}$$

where

- n is the number of devices in the system.
- t_{c_x} is a linear function in the form $ac_x + b$ that models the execution time of the device x when it computes c_x operations.
- t_{y_x} is a linear function that models the copy time of the device x when it computes c_x operations (if x is a CPU, then $t_{y_x} = 0$).

with constraints:

$$c_1, c_2, \dots , c_n \geq 0 \tag{2}$$

$$\sum_{i=0}^{n} c_i = N \tag{3}$$

where N is the total number of operations to be computed (i.e., $m * n * k$). To calculate the copy time function (y_x), we first start by computing bytes to be transferred (B) as:

$$B = dt_x * (mk + kn + mn) \tag{4}$$

where dt_x is the data type size in bytes and m, n, k are the matrix dimensions. When distributing the matrices across devices, we only vary m (see Sect. 4.3.1). Then, we

can find the relationship of bytes copied with the number of operations (c_x) by substituting m in the previous equation:

$$B = dt_x * \left(\frac{c_x}{nk} k + kn + \frac{c_x}{nk} n \right) \tag{5}$$

if we simplify and account for the memory bandwidth (bw_x), we get:

$$y_x = \frac{dt_x * \left(c_x \left(\frac{1}{k} + \frac{1}{n} \right) + kn \right)}{bw_x} \tag{6}$$

Equation 6 gives the time to copy A, B and C matrices, assuming that the communications happen in a bus exclusively used by device x. We implement the MILP problem using CPLEX 12.10 [16]. The CPLEX solver is embedded in the framework using the CPLEX API, and the MILP formulation is defined dynamically, depending on the devices being used. When the model has been optimized, the output variables of the MILP solver are $c_1, c_2, ... c_n$, which represent the number of operations to compute by each device.

4.3 Adapt (GEMM)

The optimized values given by the MILP solver in the previous phase are the number of operations, while the scheduler needs values for m, n and k. Therefore, in this phase, the number of operations is converted into matrix shape values, so they can be used by the scheduler. For this task, we designed an algorithm called `ops_to_mnk` that works on both data and hardware adjustments.

4.3.1 Data adjustments

Regarding data adjustments, the `ops_to_mnk` algorithm must accomplish two tasks:

1. Find m, n and k such that the number of operations matches the operations given by the MILP solver. This gives the m, n and k dimensions for each device.
2. Express the global matrix product as a list of squared sub-matrices products (in a best-effort manner). This divides the m, n and k dimensions for each device into sub-matrices for precise performance prediction.

For the first task, we start setting n and k to their original values. Partitioning a matrix with a different value of n would provide partial results in the output C matrix, so we fix n for convenience. Setting k to the original value makes the `ops_to_mnk` algorithm easier since just the rows of A (m) must be distributed. Then, to map ops to mnk, only m has to be determined, which is computed as $m = \frac{ops}{n*k}$.

For the second task, the algorithm must ensure that resultant matrices are as squared as possible (best-effort). Having squared matrices is the optimal scenario, as we would be performing the matrix multiplications in the same way as the profiling

phase. But it can only be accomplished if the input size is divisible by the submatrix sizes, which is not always possible. However, matrices that are very close to being squared (e.g., $m = 1.1k$) can also be predicted with very high precision. Let us denote with an apostrophe the dimensions of the submatrix (e.g., k') and without it, the dimensions of the original matrix (e.g., k). The algorithm tries to make m' and k' as similar as possible while keeping n' equal to n. Our algorithm always ensures that the number of horizontal dimensions in A fits perfectly (i.e., $k \% k' == 0$). Without such restriction, “gaps" may appear in the last column of A. Therefore, the search space in k' is restricted to the divisors of k, which happens to be big enough when the input matrix is also big. For determining m' size, the algorithm iterates over all the possibilities, analyzing how “squared” the resultant matrices using a simple heuristic would be. For a given list of squared matrices with $\{m'_1, m'_2, ... m'_n\}$ and $\{k'_1, k'_2, ... k'_n\}$, the squareness (sq) is computed as:

$$sq = \sum_{i=0}^{N} \left(\frac{min(m'_i, k'_i)}{max(m'_i, k'_i)} * m'_i k'_i n \right) \tag{7}$$

This value represents how squared the global set of sub-matrices is. Thus, to find the best sub-matrix distribution, the algorithm chooses the one that maximizes the value of the heuristic.

4.3.2 Hardware adjustments

The `ops_to_mnk` algorithm asserts that the matrix sizes satisfy the requirements imposed by the hardware to achieve optimal performance. In our case study, we consider CPUs, GPUs and tensor cores, so the `ops_to_mnk` algorithm must meet two additional requirements:

- Tensor Cores: To reach optimal performance, the input sizes must meet the following conditions: $m \% 8 == 0$ and $k \% 8 == 0$ [31]. To do so, the algorithm reduces the input size until it meets the desired requirements. In the end, this means that the tensor cores get fewer operations than the MILP solver specified, but this is barely noticeable since the size reduction is tiny compared to the global size.
- CPU cores: When profiling the CPU, inputs are designed to fit into cache memory. Therefore, when a real workload arrives, the algorithm must ensure that the generated submatrices also fit into cache.

4.4 Scheduler (GEMM)

For the scheduler, we use a static scheduling approach, as we found that gives excellent results for our case study. In other words, the scheduler receives the matrix sizes for each device and does not change them over time. We explore some of the possible issues of this approach in Sect. 5.3.

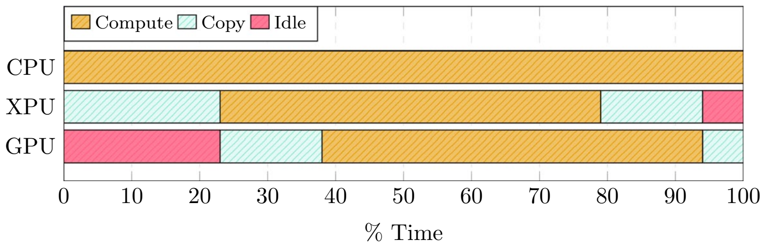

Fig. 4 Proposed scheduling communication scheme in a shared bus with CPU+GPU+XPU

Regarding the shared PCIe bus, we use a priority scheduling approach. When the program reads the configuration file, it assigns a priority for each device: the faster the device, the higher priority. Then, A and B matrices are copied in the order established by the priority. Thus, lower-priority accelerators remain idle, while the higher-priority devices are copying the data. After the computation, the first device (meaning the faster one) copies C to the host, and the same order is used to copy the remaining parts of C. In this case, higher idle times are experienced from high-priority devices, which have to wait for the rest of the devices to complete. Figure 4 shows the proposed communication scheme.

4.5 Convolution

4.5.1 Prediction

Similarly to how we divided matrix multiplication by the number of rows in matrix A, we look for a way of dividing a convolution workload to distribute it among the compute elements. We decide to divide convolutions by the minibatch size, which is a common technique in distributed and parallel approaches [5, 35]. In the profiling phase, convolution is measured by varying all parameters (image sizes, number of filters, filter sizes) except for the minibatch size, which we restrict to a reduced set. (We detail values for this set in Sect. 5.1.3.) We perform the profiling phase complying with the convolution tensor core restrictions. First, C and N must be multiple of 8 [32]. Second, the 4D tensors layout must be NHWC [33]. For simplicity, we use no padding and a stride of 1.

4.5.2 Optimize

We follow a similar formulation to the one shown in matrix multiplication, where we express the time with respect to the number of operations. Naturally, we have to compute the number of operations for convolution, which is [3]:

$$ops = K_h * K_w * C * H_{out} * W_{out} * K \tag{8}$$

where K_h and K_w are the height and width of the filters, C is the number of channels, H_{out} and W_{out} are the height and width of the output image, and K is the number of filters. In our problem formulation, we change this formula to fit our particular

needs. First, we observed that DNN implementations are typically parallelized over the number of filters (K), meaning that the execution time is invariant to K (when the filter sizes are small enough). Second, we must account for the number of minibatches in the formula. Therefore, we use the following expression:

$$ops = K_h * K_w * C * H_{out} * W_{out} * N \tag{9}$$

where N is the number of minibatches. Likewise, we compute the memory copy function following the same approach as in matrix multiplication. For example, the bytes to be copied (B) for the input image is:

$$B = dt * N * C * H * W \tag{10}$$

$$= dt * \frac{c_x}{K_h * K_w * C * O_{out} * W_{out}} * C * H * W \tag{11}$$

where, again, c_x and dt are the number of operations and the size in bytes of the data type, respectively.

4.5.3 Adapt

In convolution, we also need to adapt the optimized values (e.g., transform operations into convolution shapes). We implemented a straightforward algorithm called `ops_to_batches` that simply computes the number of minibatches (N) of each device as:

$$N = \frac{ops}{K_h * K_w * C * H_{out} * W_{out}}$$

The algorithm also ensures that the XPU input sizes passed to the scheduler have N and C multiple of 8.

4.6 Implementation details

In our POAS implementation, we copy the data between the CPU and GPU asynchronously. However, the GPU does not start computing until the whole data stream is copied. This simple approach could be improved using CUDA streams and overlapping the computation with memory copies. In either case, the performance predictor can be adapted to predict the memory copies with or without overlap. Therefore, for our study, it is not particularly relevant whether the implementation copies the data with or without overlap.

5 Evaluation

We evaluate POAS using matrix multiplication and convolution applications. Section 5.1 details our hardware and software configuration. In Sect. 5.2, we analyze the prediction accuracy of POAS. Lastly, we evaluate the performance of POAS in Sect. 5.3.

Table 1 Hardware and libraries summary for the testbed environment

		CPU	GPU	XPU
mach1	Hardware	Xeon v3	RTX 2080 Ti	RTX 2080 Ti
	Software	MKL 2020.0.2	cuBLAS 11.2.0	Same as GPU
		oneDNN 1.96.0	cuDNN 8.0.5	
mach2	Hardware	AMD EPYC	RTX 3090	RTX 2080 Ti
	Software	AOCL BLIS 3.1	cuBLAS 11.8.0	Same as GPU
		oneDNN 1.96.0	cuDNN 8.4.1	

5.1 Test bed

5.1.1 Hardware and software configuration

The evaluation platform is equipped with mach1 and mach2, two HPC servers with a CPU+GPU+XPU configuration. During this evaluation, we refer to an XPU as a GPU that uses the tensor cores to perform the matrix multiplication, whereas GPU uses traditional CUDA cores. The hardware configuration, as well as the summary of the libraries that POAS relies on, is summarized in Table 1. The specifications for each device are detailed in Table 2.

Both systems run Centos 8.2 (4.18.0-193 kernel in mach1 and 4.18.0-348 in mach2). We build POAS using g++ 8.4.1. Regarding the communication between CPU and GPUs, the RTX 2080Ti's in mach1 are connected to a PCIe 3.0 x16 bus, which peak memory bandwidth is 15.75 GB/s. In mach2, both cards are connected to a PCIe 4.0 x16 bus, providing a peak memory bandwidth of 31.75 GB/s. Since the RTX 2080Ti supports up to PCIe 3.0, the card in mach2 works in 3.0 mode, even though it is connected to a 4.0 slot. In both mach1 and mach2, both GPUs are connected to the same PCI channel, and thus, the PCI bus usage is similar to what Fig. 4 shows. For the convolution, we use the `CUDNN_TENSOR_NHWC` tensor format, as it is the optimal format for tensor cores [33]. For the experiments, we reserve one physical CPU core for managing the GPU and XPU. Henceforth, mach1 has 5 physical cores and mach2 has 23 cores to run the CPU workloads.

5.1.2 Input sizes

For matrix multiplication, we conceive six different matrix sizes (shown in Table 3) sorted in descending order by the number of operations (TOps). We are interested in evaluating relatively small matrices, like the first two inputs, as well as squared and non-squared matrices. We also want to study very skinny matrices like input 3, where the m dimension is much larger than the others. The same idea is explored for n and k dimensions in inputs 4 and 5. Those inputs are useful to understand how solid the predictor is because they allow us to see if the predictor performs well on non-square and skinny matrices.

Table 2 Hardware specifications for the testbed environment

Model	CPUs		GPUs / XPUs	
	Intel Xeon E5-2603 v3	AMD EPYC 7413	NVIDIA RTX 2080 Ti	NVIDIA RTX 3090
Architecture	Haswell	Zen 3	Turing	Ampere
Technology	22 nm	7 nm	12 nm	8 nm
CPU cores	6	24	–	–
CUDA cores	–	–	4352	10496
Tensor cores	–	–	544	328
Max. frequency	1.6 GHz	3.6 GHz	1.5 GHz	1.6 GHz
TFLOP/s (FP32)	0.307	2.76	13.45	35.58
TFLOP/s (FP16)	–	–	107.5	284.65
LLC	15 MB	128 MB	6 MB	6 MB
Memory size	64 GB	512 GB	11 GB	24 GB

For convolution, we design four inputs (shown in Table 4) based on real CNN workloads [45]. Inputs 1 and 3 are representative of the ResNet 50 architecture, while inputs 2 and 4 are based on AlexNet.[1] Due to memory size limitations, inputs 1 and 2 are executed only in mach1, and inputs 3 and 4 are run in mach2, which has a bigger GPU memory size. For each input, we repeat the computations 50 times, therefore executing 50 matrix multiplication and convolutions over the accumulated data. We run each input ten times, and the values shown are the average over these three independent runs.

5.1.3 Profiling configuration

In matrix multiplication, the profiling phase performs 30 squared matrix products with matrix sizes ranging between 1000 and 2000 for the CPU and between 3000 and 6000 for GPU/XPU. For the generation of the list of squared sub-matrices, they are restricted to be of a size such that the number of operations are between the same number of operations that were performed during profiling. In other words, in the CPU, the sub-matrices are restricted to $1000 \times 1000 \times 1000$ (10^9) and $2000 \times 2000 \times 2000$ (8×10^9) operations, and in GPU between $3000 \times 3000 \times 3000$ ($27 * 10^8$) and $6000 \times 6000 \times 6000$ ($216 * 10^8$) operations. Thus, sizes are computed on the fly depending on the size of n in the original matrix.

In convolution, the profiling phase performs a set of convolutions with a minibatch size of 8, 128 and 256 for CPU, GPU and XPU, respectively. Similarly to matrix multiplication, those sizes are the same as the minibatch size used in real workloads.

[1] We reduced the number of filters due to GPU memory limits.

Table 3 Input sizes (GEMM)

Input	m (K)	n (K)	k (K)	TOps
1	30	30	30	27.0
2	60	20	35	42.0
3	130	20	20	52.0
4	40	80	20	64.0
5	40	30	60	72.0
6	56	40	40	89.6

Table 4 Input sizes (convolution)

Input	n	c	h	w	k	kh	kw
1	3500	16	224	224	16	7	7
2	4000	16	227	227	16	11	11
3	3000	32	224	224	16	7	7
4	2500	32	227	227	16	11	11

5.2 Prediction accuracy

To evaluate the performance predictor used in POAS, we study the prediction accuracy. We measure and compare the execution and memory copy times with the predicted values. Then, we calculate the prediction error e as an expression of the relative error: $e = 100 * \frac{v - v_{pred}}{v}$, where v is the measured time in our experiments and v_{pred} is the value given by the predictor. We also compute the root mean square error (RMSE), which gives a perspective of the prediction robustness across different inputs.

Tables 5 and 6 show the prediction error and root mean square error (RMSE) for GPU and XPU, where we show the global prediction error (and RMSE) in the first instance, followed by the computing and memory copy prediction error (and RMSE), respectively. Overall, we observe that the prediction error is low (typically, under 5%). This is a key factor to provide high-quality co-execution because otherwise, the load imbalance would be very high, leading to substantial performance degradation. Except for a few cases, the memory prediction error is very low, especially for mach2, whose prediction is close to being perfect. Some inputs are predicted with slightly higher prediction error ratios than the mean (e.g., the latest ones in the GPU and XPU in mach1). In fact, these "outliers" are the main fact that increases the RMSE of the whole evaluation. We believe that these observations are caused by high temperatures, which cause overheating. During the profiling phase, we leave all the device's frequencies unlocked. Because the profiling phase is relatively short, the device does not get significantly hotter than the idle temperature. However, in real workloads, the temperature can increase much more, downscaling the clock frequency to avoid overheating. In other words, the measured frequency in the profiling phase may not match the frequency used in real workloads. This is

Table 5 Root mean square error (RMSE) and prediction error for GEMM

Input	mach1			mach2		
	CPU	GPU	XPU	CPU	GPU	XPU
	Comp	Global (Comp., Copy)	Global (Comp., Copy)	Comp	Global (Comp., Copy)	Global (Comp., Copy)
1	3.5%	1.2% (7.8%,5.3%)	1.1% (3.7%,1.0%)	2.3%	4.0% (8.1%,0.3%)	4.8% (10.1%,1.0%)
2	1.5%	3.1% (5.5%,0.4%)	3.1% (3.1%,0.1%)	0.9%	1.3% (2.5%,0.0%)	6.2% (11.7%,1.0%)
3	3.3%	1.7% (1.5%,2.0%)	3.8% (6.9%,0.5%)	0.7%	0.7 %(1.9%,0.6%)	7.6% (12.4%,0.7%)
4	4.6%	9.5% (5.5%,13.6%)	5.4% (6.4%,4.2%)	2.0%	2.0% (4.0%,0.3%)	4.8% (8.0%,1.2%)
5	2.3%	5.5% (9.0%,0.0%)	2.9% (6.0%,0.1%)	1.3%	6.1% (10.1%,0.3%)	5.3% (10.4%,1.2%)
6	0.6%	6.7% (7.0%,5.8%)	2.8% (3.9%,1.3%)	3.2%	3.9% (6.5%,0.3%)	6.8% (11.0%,0.7%)
RMSE	2.37	5.50 (3.38,5.50)	3.11 (2.64,0.74)	1.58	2.84 (2.83,1.61)	4.47 (4.73,0.29)

The compute (Comp.) error and RMSE are shown for CPU, whereas the error and RMSE are divided into computing and memory copy for GPU and XPU (in parentheses), along with the global error and RMSE

Table 6 Root mean square error (RMSE) and prediction error for convolution

Input	mach1			Input	mach2		
	CPU	GPU	XPU		CPU	GPU	XPU
	Comp	Global (Comp., Copy)	Global (Comp., Copy)		Comp	Global (Comp., Copy)	Global (Comp., Copy)
1	0.1 %	1.8 % (0.0%,2.4%)	1.2 % (1.4%,1.1%)	3	3.7 %	0.0 % (0.5%,0.0%)	0.5 % (0.2%,0.9%)
2	6.4 %	0.3 % (2.9%,3.1%)	0.3 % (0.4%,0.1%)	4	1.6 %	0.6 % (1.9%,0.1%)	1.7 % (1.8%,1.6%)
RMSE	2.23	0.37 (0.47,0.67)	0.28 (0.16,0.12)	RMSE	0.88	0.16 (0.17,0.02)	0.44 (0.24,0.23)

The compute (Comp.) error and RMSE are shown for CPU, whereas the error and RMSE are divided into computing and memory copy for GPU and XPU (in parentheses), along with the global error and RMSE

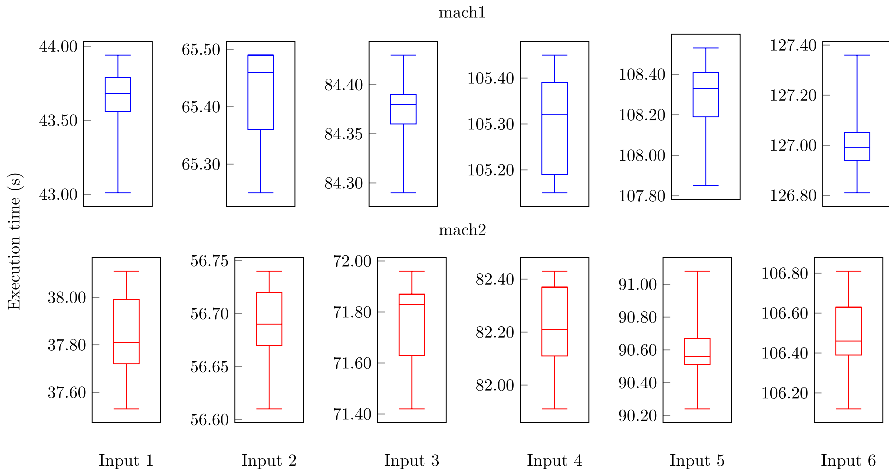

Fig. 5 Box plots of the execution time of GEMM in mach1 (blue) and mach2 (red)

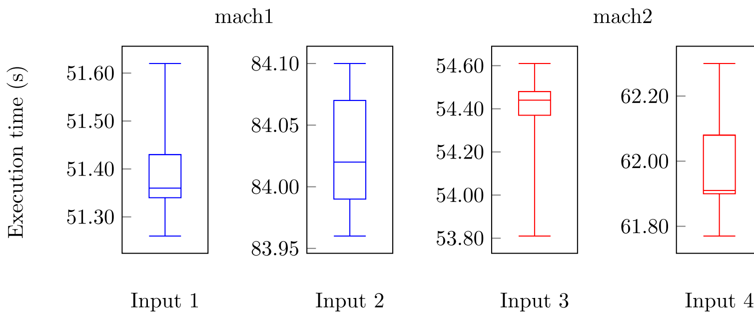

Fig. 6 Box plots of the execution time of convolution in mach1 (blue) and mach2 (red)

especially true for mach1 since it has substantially worse heat dissipation capabilities than mach2.

Regarding RMSE, POAS achieves very low values for both use cases, which confirms the great robustness of the predictor, despite the use of static scheduling. However, a more sophisticated solution could employ a dynamic scheduler that considers the frequency in real-time of every device and dynamically balance the workload to further improve accuracy. In either case, POAS fully adapts to the underlying hardware, properly exploiting its computing power.

Figures 5 and 6 show the boxplots of the execution time for both applications. The variance between runs is very low, yielding a mean difference between the longest and shortest of 0.73%. Based on our results, we can confirm that POAS is able to efficiently exploit ALP in CPU+GPU+XPU environments for GEMMs and convolutions.

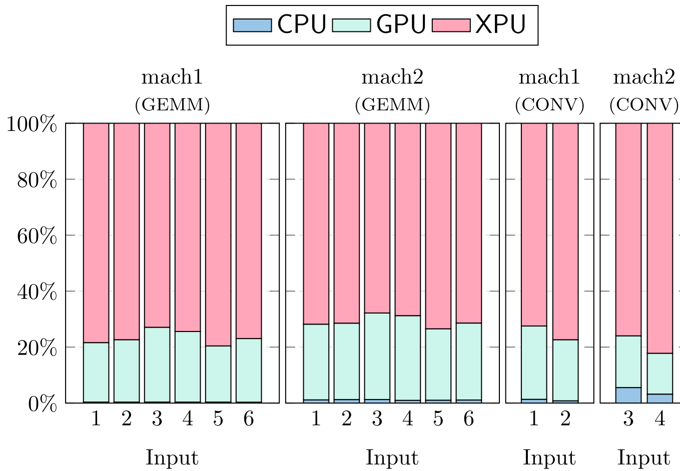

Fig. 7 Percentage of work distribution among devices in mach1 and mach2 for GEMM and convolution

5.3 Performance

5.3.1 POAS overhead

To evaluate the overhead of the framework we measure the execution time of the main components of POAS. First, we measure how much is the cost of using CPLEX to solve the MILP problem at runtime, used in the Optimize phase. In all our experiments, the execution time of CPLEX was between 0.1 and 0.2 s, which becomes negligible compared to the execution of the actual computation. Another candidate for inducing unwanted overhead is the `ops_to_mnk` and `ops_to_batches` algorithms, used in the Adapt phase. Those algorithms are very fast, with typical execution times in the order of a few milliseconds, so their time does not add any overhead to the framework. In fact, the majority of the time spent before computing the actual application comes from the initialization of the GPU/XPU and the memory allocation, both unrelated to the framework.

5.3.2 Work distribution

We show the workload distribution used by POAS in Fig. 7. As we can see, the CPU provides little help in computing the matrix multiplication (especially in mach1, where it gets less than 1% of the work), while the GPU takes between 20% and 30% of the work. Because matrix multiplication is a very compute-intensive workload, the communication penalty between the CPU and the accelerators is smaller than the higher computational power of the accelerators. On the other hand, convolution has a lower arithmetic intensity, so it is easier for the CPU to contribute more work than in matrix multiplication since memory copy overhead is bigger in the accelerators. Hence, we can observe that the CPU participates more in both machines and

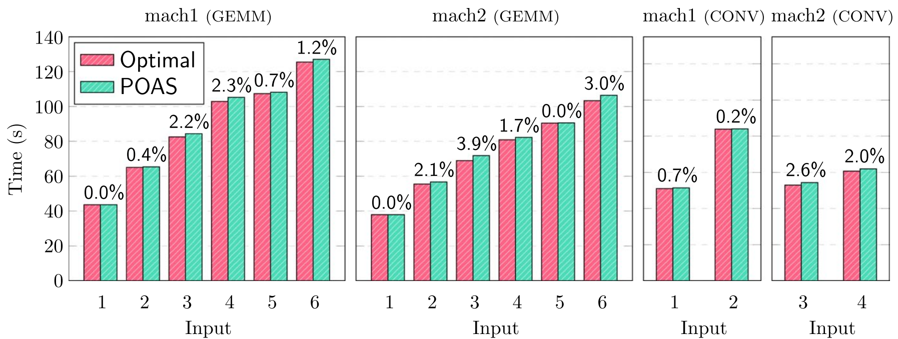

Fig. 8 Runtime comparison of POAS implementation for GEMM and convolution against optimal distribution

the GPU participates less since the shared bus is occupied more often by the XPU, which has a higher priority in accessing the PCIe bus. This figure is also useful to compare when ALP is a better choice over offloading. As mentioned, the CPU in our setup is very weak compared to the XPU. In a setup with only the weak CPU and the XPU, ALP would yield negligible improvement compared to offloading. The same would happen if having a slow GPU. In essence, when the accelerator is significantly more powerful than the rest of the devices, ALP provides little speedup over offloading. However, in our setup, the CUDA cores are a valuable resource when used in conjunction with the XPU compared to offloading only to the accelerator, so ALP is indeed beneficial.

Now, we compare POAS execution time against the optimal work distribution. To find the optimal distribution, we develop a small program that explores all possible work distributions and finds the one that achieves the minimum execution time. Figure 8 shows that POAS distribution was very close to the optimal in both machines and applications. The difference between the POAS and optimal distributions comes from two factors. The first one is prediction errors, which we already studied in depth. The second one is load unbalance in the POAS work distribution. Even though POAS aims to distribute evenly the work among devices, this is not always possible. Sometimes, inputs must be divided in non-even distributions to make sure that accelerators receive the optimal input size. (We discussed this in Sect. 4.3.2.) This division unbalances the distribution because other devices must take the remainder work from the accelerator, or do less work since it is now done by the accelerator. In any case, this second factor has less influence than the prediction error. However, we observe that POAS tends to give excessive work to the CPU in mach2. A small work excess in the CPU has a bigger impact because it is more sensible to execution time variations. This explains why POAS is closer to optimal in mach1.

6 Conclusions and future work

Energy-constrained systems benefit from accelerators thanks to their lower consumption while high-performance systems also take advantage of massive performance improvements in compute-intensive workloads. To exploit heterogeneity, accelerator level parallelism (ALP) is a promising approach. As the number of applications in which accelerators are used is growing quickly, we need solutions that allow this process to be performed easily and efficiently.

This work has presented POAS, a framework for scheduling workloads among the heterogeneous compute elements available within a node. POAS adapts to the software libraries and hardware, maximizing resource usage. We tested our framework on two different fields: linear algebra (matrix multiplication benchmark) and deep learning (convolution benchmark) using a heterogeneous environment consisting of CPUs, GPUs (CUDA cores), and XPUs (Tensor cores). Our framework, POAS, showed excellent performance, completing the tasks in a time very close to the optimal time for the hardware and applications used, with a negligible execution time overhead. Additionally, the POAS predictor performed very well, achieving very low RMSE values for both use cases.

Therefore, POAS can be a valuable tool to fully exploit ALP to improve overall performance over offloading in heterogeneous settings.

For future work, we plan to extend POAS with more sophisticated scheduling policies. We also plan to study how the framework adapts to other kinds of problems where predicting the execution time upfront is harder or not possible, like in sparse matrix applications. Another open topic is how to efficiently schedule the communications between the CPU and accelerators, which can also have a notable impact on overall performance, especially in shared bus scenarios.

Acknowledgements We are grateful to Francisco de Asis Guil Asensio for his insightful feedback and discussions with the MILP model used in Sect. 4.2. This work was conducted while Pablo Antonio Martínez was with the University of Murcia, Murcia, Spain.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. Grant No. (TED2021-129221B-I00) funded by MCIN/AEI/10.13039/501100011033 and by the "European Union NextGenerationEU/PRTR," and Grant No. (PID2022-136315OB-I00) funded by MCIN/AEI/10.13039/501100011033/ and by "ERDF A way of making Europe," EU.

Data availability The data can be provided upon request.

Declarations

Ethical approval Not applicable.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line

to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Ahmed U, Lin JC, Srivastava G, Aleem M (2021) A load balance multi-scheduling model for OpenCL kernel tasks in an integrated cluster. Soft Comput 25(1):407–420
2. Anders M, Kaul H, Mathew S, et al. (2018) 2.9TOPS/W Reconfigurable Dense/Sparse Matrix-Multiply Accelerator with Unified INT8/INTI6/FP16 Datapath in 14NM Tri-Gate CMOS. In: 2018 IEEE Symposium on VLSI Circuits, pp. 39–40
3. Basha SS, Farazuddin M, Pulabaigari V, Dubey SR, Mukherjee S (2024) Deep model compression based on the training history. Neurocomputing 573:127257
4. Beaumont O, Boudet V, Rastello F, Robert Y (2001) Matrix multiplication on heterogeneous platforms. IEEE Trans Parallel Distrib Syst 12(10):1033–1051
5. Ben-Nun T, Hoefler T (2019) Demystifying parallel and distributed deep learning: an in-depth concurrency analysis. ACM Comput Surv 52(4):1–43
6. Cámara J, Cuenca J, Giménez D (2020) Integrating software and hardware hierarchies in an autotuning method for parallel routines in heterogeneous clusters. J Supercomput 76(12):9922–9941
7. Catalán S, Igual FD, Mayo R, et al. (2015) Performance and energy optimization of matrix multiplication on asymmetric big. LITTLE Processors
8. Choquette J, Giroux O, Foley D (2018) Volta: performance and programmability. IEEE Micro 38(2):42–52
9. Dally WJ, Keckler SW, Kirk DB (2021) Evolution of the graphics processing unit (GPU). IEEE Micro 41(6):42–51
10. Dally WJ, Turakhia Y, Han S (2020) Domain-specific hardware accelerators. Commun ACM 63(7):48–57
11. Esmaeilzadeh H, Blem E, St. Amant R, Sankaralingam K, Burger D (2011) Dark silicon and the end of multicore scaling. ISCA '11, New York, NY, USA. Association for Computing Machinery, pp. 365–376
12. Ford BW, Zong Z (2022) A cost effective framework for analyzing cross-platform software energy efficiency. Sustain Comput: Inform Syst 35:100661
13. Forsberg B, Benini L, Marongiu A (2021) HePREM: a predictable execution model for GPU-based heterogeneous SoCs. IEEE Trans Comput 70(1):17–29
14. Geng T, Amaris M, Zuckerman S et al (2022) A profile-based AI-assisted dynamic scheduling approach for heterogeneous architectures. Int J Parallel Prog 50(1):115–151
15. Hill MD, Reddi VJ (2021) Accelerator-level parallelism. Commun ACM 64(12):36–38
16. IBM. IBM ILOG CPLEX Optimizer, 2022. https://www.ibm.com/analytics/cplex-optimizer
17. Intel. Optimizing software for x86 Hybrid archiecture. Intel White Paper, 2021
18. Intel. Intel oneAPI Programming Guide, 2022. https://www.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top.html
19. Jia Z, Maggioni M, Smith J, Scarpazza DP(2019) Dissecting the NVidia turing T4 GPU via Microbenchmarking
20. Jia Z, Maggioni M, Staiger B, Scarpazza DP (2018) Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking
21. Jouppi NP, Young C, Patil N, et al. (2017) In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA '17, New York, NY, USA, Association for Computing Machinery, pp. 1–12
22. Kang H, Kwon HC, Kim D (2020) HPMaX: heterogeneous parallel matrix multiplication using CPUs and GPUs. Computing 102(12):2607–2631
23. Lee H, Ruys W, Henriksen I, et al. (2022) Parla: a python orchestration system for heterogeneous architectures. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '22

24. Mack J, Arda SE, Ogras UY, Akoglu A (2022) Performant, multi-objective scheduling of highly interleaved task graphs on heterogeneous system on chip devices. IEEE Trans Parall Distrib Syst 33(9):2148–2162
25. Martínez PA, Peccerillo B, Bartolini S et al (2022) Applying Intel's oneAPI to a machine learning case study. Concurr Comput Pract Exper 34(13):6917
26. Martínez PA, Peccerillo B, Bartolini S et al (2022) Performance portability in a real world application: PHAST applied to Caffe. Int J High Perform Comput Appl 36(3):419–439
27. Michel Brown W, Carrillo JM, Gavhane N et al (2015) Optimizing legacy molecular dynamics software with directive-based offload. Comput Phys Commun 195:95–101
28. Nguyen D, Lee J (2016) Communication-aware mapping of stream graphs for Multi-GPU Platforms. In: Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO '16, pp. 94–104, New York, NY, Association for Computing Machinery
29. Nozal R, Bosque JL, Beivide R (2020) EngineCL: usability and performance in heterogeneous computing. Futur Gener Comput Syst 107:522–537
30. Nozal R, Bosque JL (2021) Straightforward heterogeneous computing with the oneapi coexecutor runtime. Electronics 10(19):2386
31. NVIDIA. CUDA Toolkit Documentation (cuBLAS): tensor core usage, 2022. https://docs.nvidia.com/cuda/cublas/index.html#tensorop-restrictions
32. NVIDIA. Guidelines for good performance on tensor cores, 2022. https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html#tensor-ops-guidelines-for-dl-compiler
33. NVIDIA. Tensor Layouts In Memory: NCHW vs NHWC, 2022. https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html#tensor-layout
34. Ouyang X, Zhu Y (2022) Core-aware combining: accelerating critical section execution on heterogeneous multi-core systems via combining synchronization. J Parall Distrib Comput 162:27–43
35. Oyama, Y, Ben-Nun, T, Hoefler T, Matsuoka S (2018) Accelerating deep learning frameworks with micro-batches. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER), pp. 402–412
36. Park DH, Pal S, Feng S et al (2020) A 7.3 M output non-zeros/J, 11.7 M output non-zeros/gb reconfigurable sparse matrix-matrix multiplication accelerator. IEEE J Solid-State Circ 55(4):933–944
37. Peccerillo B, Mannino M, Mondelli A, Bartolini S (2022) A survey on hardware accelerators: taxonomy, trends, challenges, and perspectives. J Syst Architect 129:102561
38. Pellizzoni R, Betti E, Bak S, et al. 2011 A predictable execution model for COTS-based embedded systems. In: 2011 17th IEEE Real-time and Embedded Technology and Applications Symposium, pp. 269–279
39. Peng J, Li K, Chen J, Li K (2022) HEA-PAS: a hybrid energy allocation strategy for parallel applications scheduling on heterogeneous computing systems. J Syst Architect 122:102329
40. Pérez B, Stafford E, Bosque JL, Beivide R, Mateo S, Teruel X, Martorell X, Ayguadé E (2019) Auto-tuned OpenCL kernel co-execution in OmpSs for heterogeneous systems. J Parall Distrib Comput 125:45–57
41. Raca V, Umboh SW, Mehofer E, Scholz B (2022) Runtime and energy constrained work scheduling for heterogeneous systems. J Supercomp 78(15):17150–17177
42. Rodríguez A, Navarro A, Nikov K et al (2022) Lightweight asynchronous scheduling in heterogeneous reconfigurable systems. J Syst Archit 124:102398
43. Sorokin A, Malkovsky S, Tsoy G (2022) Comparing the performance of general matrix multiplication routine on heterogeneous computing systems. J Parall Distrib Comput 160:39–48
44. Stevens JD, Klöckner A (2020) A mechanism for balancing accuracy and scope in cross-machine black-box GPU performance modeling. Int J High Perform Comput Appl 34(6):589–614
45. Sze V, Chen Y-H, Yang T-J, Emer JS (2017) Efficient processing of deep neural networks: a tutorial and survey. Proceedings IEEE 105(12):2295–2329
46. Thompson NC, Spanuth S (2021) The decline of computers as a general purpose technology. Commun ACM 64(3):64–72
47. Wen Y, O'Boyle MFP (2017) Merge or Separate? Multi-Job Scheduling for OpenCL Kernels on CPU/GPU Platforms. In: Proceedings of the General Purpose GPUs, GPGPU-10, New York, NY, USA. Association for Computing Machinery, pp 22–31
48. Wen Y,Wang Z, O'Boyle MFP (2014) Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In: 2014 21st International Conference on High Performance Computing (HiPC), pp. 1–10

49. Yesil S, Ozturk O (2022) Scheduling for heterogeneous systems in accelerator-rich environments. J Supercomput 78(1):200–221
50. Zhang F, Zhai J, He B, Zhang S, Chen W (2017) Understanding Co-Running Behaviors on Integrated CPU/GPU Architectures. IEEE Trans Parallel Distrib Syst 28(3):905–918
51. Zhou N, Liao X, Li F et al (2021) List scheduling algorithm based on virtual scheduling length table in heterogeneous computing system. Wirel Commun Mob Comput 2021:9529022

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.