Journal of Systems Architecture 124 (2022) 102398

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Lightweight asynchronous scheduling in heterogeneous reconfigurable systems

Andrés Rodríguez [a], Angeles Navarro [a], Kris Nikov [b], Jose Nunez-Yanez [b], Rubén Gran [c], Darío Suárez Gracia [c], Rafael Asenjo [a,*]

[a] Department of Computer Architecture, Universidad de Málaga, Spain
[b] Department of Electrical & Electronic Engineering. University of Bristol, UK
[c] Computer Architecture Group. Universidad de Zaragoza, Spain

ARTICLE INFO

Keywords:
Heterogeneous architecture
FPGA
Heterogeneous scheduling
Throughput model
Energy efficiency

ABSTRACT

The trend for heterogeneous embedded systems is the integration of accelerators and general-purpose CPU cores on the same die. In these integrated architectures, like the Zynq UltraScale+ board (CPU+FPGA) that we target in this work, hardware support for shared memory and low-overhead synchronization between the accelerator and the CPU cores make the case for exploring strategies that exploit a tight collaboration between the CPUs and the accelerator. In this paper we propose a novel lightweight scheduling strategy, FastFit, targeted to FPGA accelerators, and a new scheduler based on it, named MultiFastFit, which asynchronously tackles heterogeneous systems comprised of a variety of CPU cores and FPGA IPs. Our strategy significantly reduces the overhead to automatically compute the near-optimal chunksizes when compared to a previous state-of-the-art auto-tuned approach, which makes our approach more suitable for fine-grained applications. Additionally, our scheduler MultiFastFit has been designed to enable the efficient co-execution of work among compute devices in such a way that all the devices are busy while minimizing the load unbalance.

Our approaches have been evaluated using four benchmarks carefully tuned for the low-power UltraScale+ platform. Our experiments demonstrate that the FastFit strategy always finds the near-optimal FPGA chunksize for any device configuration at a reasonable cost, even for fine-grained and irregular applications, and that heterogeneous CPU+FPGA co-executions that exploit all the compute devices are usually faster and more energy efficient than the CPU-only and FPGA-only executions. We have also compared MultiFastFit with other state-of-the-art scheduling strategies, finding that it outperforms other auto-tuned approach up to 2x and it achieves similar results to manually-tuned schedulers without requiring an offline search of the ideal CPU-FPGA partition or FPGA chunk granularity.

1. Introduction

The demise of Dennard's scaling has boosted the interest on heterogeneous platforms, which are now seen as a path forward to deliver the energy and the performance improvements needed over the next decade. A plethora of heterogeneous devices and platforms, featuring CPU cores, GPUs and FPGAs, among other accelerators and ASICs, have arisen. In this arena, some consensus exists over the "No transistor left behind" idea, meaning that all devices should help in optimizing different kind of applications or part of them. In this regard, new heterogeneous programming models, as SYCL [1], DPC++ and oneAPI [2], are gaining momentum in order to ease the development of heterogeneous applications without compromising performance.

In this paper, we aim at contributing towards this goal, by proposing heterogeneous schedulers capable of distributing the workload among CPU cores and FPGA compute units (usually known as FPGA IPs), minimizing load unbalance and energy consumption. More precisely, we target a Xilinx Zynq UltraScale+ board featuring 4 Cortex-A53 cores along with an on-chip FPGA. The FPGA is large enough to accommodate up to four instances (4 IPs) of the kernels we evaluate in this work. In Fig. 1 we show four possible heterogeneous computing scenarios of increasing complexity considered in this work. Scenario A is a very common one in which only the accelerator is used, and out of the four CPU cores, only one is running the so-called "host-thread" that is just taking care of feeding the FPGA IPs (busy-waiting

* Corresponding author.
E-mail addresses: andres@ac.uma.es (A. Rodríguez), angeles@ac.uma.es (A. Navarro), kris.nikov@bristol.ac.uk (K. Nikov), j.l.nunez-nanez@bristol.ac.uk (J. Nunez-Yanez), rgran@unizar.es (R. Gran), dario@unizar.es (D. Suárez Gracia), asenjo@ac.uma.es (R. Asenjo).

https://doi.org/10.1016/j.sysarc.2022.102398
Received 12 May 2021; Received in revised form 1 December 2021; Accepted 2 January 2022
Available online 15 January 2022

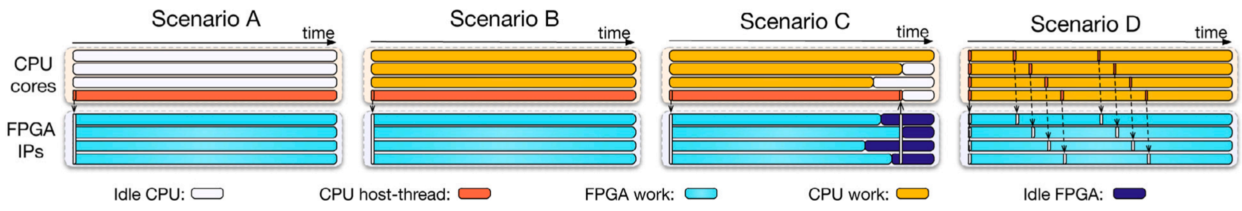


Fig. 1. Different scenarios when using CPU+FPGA platforms.

until FPGA completion). Scenario B, shows a situation in which one CPU cores runs the "host-thread" while the other three CPU cores can collaborate in the computation. If the application is regular and there is no noise in the platform (other processes) a perfect balance can be achieved. These conditions are not always met, so in Scenario C we see a common situation for an irregular code in which the workload cannot be statically estimated beforehand. In this case, seven chunks of work (of an estimated size) are submitted to the 3 CPU cores and the 4 FPGA IPs, while the host-thread consumes another core busy-waiting for the FPGA to finish. Due to either the irregularities of the code or to other noise in the platform, the estimated size does not result in evenly balanced workload, which results in resources underutilization (shown in the figure as idle CPU and idle FPGA times).

Our goal is to achieve Scenario D, in which: (i) there is no host-tread wasting a CPU core; and (ii) the workload is partitioned into variable size chunks of work that are dynamically offloaded to the FPGA IPs and CPU cores, so that near-perfect load balanced is achieved.

This last scenario (Scenario D) may come as a surprise to developers that are used to targeting discrete accelerators connected through a PCIe bus. With these discrete accelerators, it usually makes sense to minimize the number of data offloads and kernel invocations due the high latency of the PCIe bus. However, the trend for heterogeneous embedded systems, integrating accelerators and general-purpose CPU cores on the same die, is to enable unified virtual address spaces for both devices, which, in turn, seamlessly allows for data sharing and low-overhead synchronization between the accelerator and the CPU cores. In these integrated architectures, like our UltraScale+ board (CPU+FPGA), it can be profitable to explore strategies that exploit a tight collaboration and communication between the CPUs and the accelerator.

In addition to the scheduling problem, we are also concerned about the programmer's productivity and about the platform energy consumption. For some years running, we have been contributing to solve these issues by proposing a high level API called Heterogeneous Building Blocks (HBB). This library offers an easy-to-use parallel_for template built on top of TBB (Threading Building Blocks) [3] and encapsulates different heterogeneous schedulers that we have validated on CPU+GPU [4] and CPU+FPGA [5,6] platforms. Some of the included scheduling strategies are Static, Dynamic, Logfit (originally devised for CPU+GPU) and HAP (a Logfit specialization for FPGAs).

Logfit and HAP are capable of dynamically distributing different chunks[1] of the parallel iteration space among CPU cores and the accelerator. To this end, the scheduler monitors the throughput of each compute unit during the execution of the iterations and uses this metric to adaptively resize the CPU and accelerator chunks in order to optimize overall throughput and to prevent underutilization and load unbalance. One disadvantage of these scheduling strategies is the overhead in which incur due to the logarithmic fitting needed to recompute the accelerator chunks. In this work, we take a step forward and extend the previous schedulers in different aspects: (i) now they do not rely on busy-waiting host threads to feed the FPGA IP. Instead we incorporate an interruption mechanism that blocks a host thread while the FPGA IP is working and later it is woken-up once the IP has finished; (ii) now they are able to consider several FPGA IPs working asynchronously instead of just one; and (iii) now a new scheduling strategy, called FastFit, designed to simplify the computation of accelerator chunks and further reduce the scheduling overheads on CPU+FPGA systems, and a new version of our scheduler based on it, named MultiFastFit, which is able to tackle multiple FPGA IPs, are added to the library.

With all this, this paper proposes the following novel contributions:

1. A proposal of an FPGA-specialized scheduling strategy, called FastFit (from Fast Fitting), which simplifies the training phase of our scheduler by using just two runtime samples to estimate the near-optimal FPGA chunksize of iterations. This lightweight strategy significantly reduces the number of samples and the cost of re-computing FPGA chunksizes when compared to the previous state-of-the-art HAP scheduling strategy. HAP requires four (or more, particularly in fine-grained applications) runtime samples to perform the first logarithmic fitting to compute the FPGA chunksize, and later each time that a new FPGA chunk is required, it has to recompute the new size applying logarithmic fitting again.
2. An extension of our schedulers that now asynchronously cope with several FPGA IPs and several CPU cores (MultiStatic, MultiDynamic, MultiHAP and MultiFastFit), striving to keep all the compute devices busy and minimize the load unbalance. FPGA IPs are configured to generate interruptions as soon as they are ready to process more data.
3. A thorough exploration of the optimality, scalability, performance and energy efficiency of our MultiFastFit proposal for different device configurations using four applications in which the main kernels have been carefully tuned for the low-power UltraScale+ CPU+FPGA platform. We compare this scheduler against MultiStatic, MultiDynamic and MultiHAP schedulers to find that the FastFit strategy always discovers the near-optimal FPGA chunksize for all device configurations without requiring manually tuning, and at a reasonable cost even for fine-grained and irregular applications.

The rest of the paper is organized as follows. The next section introduces the problem as well as related works that tackle a similar problem. Section 3 briefly describes our heterogeneous library, HBB, and a summary of implementation details for the different scheduling strategies that have been studied. Section 4 outlines the platform configuration and benchmark characteristics. In Section 5 we delve into a thorough and extensive experimental evaluation that covers the FastFit model validation, scalability and energy efficiency of the scheduler proposed. Finally, we wrap up with conclusions in Section 6.

2. Background and related work

The traditional approach of using heterogeneous platforms consists of offloading complex kernels to the accelerator by selecting the best execution resource either at compile time or runtime. New programming models and frameworks such as OmpSs [7], oneAPI [2] or SYCL [1] are being proposed following this paradigm. However, for the parallel_for pattern, which is of interest to our research work, they do

[1] A chunk is a block of consecutive iterations that are independent of other iterations or chunks of a parallel loop.

not solve the automatic workload partition and scheduling problems. In particular, those approaches do not consider any compile time or runtime mechanism to find the most suitable work granularity for each device. Moreover, there are cases in which it makes sense to exploit all the available compute devices at once. This means that there is the need to develop a strategy to divide the work among all these compute resources so they are used effectively. This is often referred as workload balancing or collaborative co-execution, which has been explored extensively in the literature in systems that combine GPUs and CPUs. For example, Fluidic [8] allows for CPU+GPU collaborative execution of parallel loops. The GPU starts computing a data buffer in ascending order in parallel with the CPU that process sub-ranges (chunks) of iterations in descending order. The GPU is responsible of aborting its execution when it tries to execute an iteration that has been already processed by the CPU. Another related work focusing on parallel_for [4] has also been implemented for heterogeneous chips composed of CPU plus an integrated GPU.

The possibility of adding an FPGA to a system using CPU and GPU and use them simultaneously and collaboratively has also been explored in diverse application areas such as medical research [9]. In that work, the hardware uses multiple devices connected through a common PCIe backbone, and the designers optimized how different parts of the application are mapped to each computing resource. This type of heterogeneous computing can be considered as connecting devices vertically (i.e. exploit temporal parallelism) since the idea is to build a streaming pipeline with results from one part of the algorithm moving to the next. Data is captured and initially processed in the FPGA and moved with DMA engines to the CPU and GPU components. A study of the potential of FPGAs and GPUs to accelerate data center applications is done in [10]. The paper confirms that FPGA and GPU platforms can provide compelling energy efficiency gains over general purpose processors, but it also indicates that the possible advantages of FPGAs over GPUs are unclear due to the similar performance per watt and the significant programming effort of FPGAs. In any case, it is important to note that the paper does not use high level languages to increase FPGA productivity as done in our work, and the power measurements for the FPGA are based on worst case tool estimations and not direct measurements as done here.

In our research, we explore a horizontal (i.e. exploit data parallelism) collaborative solution more closely related to the work done in [11] that also targets Xilinx FPGAs. This previous research focuses on a multiple device solution and demonstrates how the N-Body algorithm can be implemented in a heterogeneous platform in which both FPGA and GPU work together to compute the same algorithm kernel on different sets of particles. While the research in this paper uses a dynamic scheduling approach to compute the optimal split, in [11] the split is calculated manually with 2/3 of the workload to FPGA and the remaining 1/3 to GPU. The concept of distributing the workload at runtime has also been studied in other heterogeneous CPU+FPGA systems such as the Intel HARP that combines high-end Xeon-class tightly coupled to an FPGA device located in the same package [6] or the low-power low-cost Terasic DE1 board that comprises two ARM Cortex-A9 and an embedded FPGA on the same chip [12]. In these two previous works a scheduling strategy called HAP (Heterogeneous Adaptive Partitioner), which adaptively and continuously adjusts the size of the chunk of iterations offloaded to both CPU cores and the FPGA device is evaluated. This strategy uses a three-phase strategy consisting of training, stable and final where it tries to determine the iteration chunksize that maximizes the performance of the FPGA. In this paper we also evaluate HAP as a comparison point, but while in the two previous works only one FPGA IP was considered, in this work we extend HAP to asynchronously serve several FPGA IPs. A different approach presented in [13] is based on work-stealing. In this case, threads that have completed their assigned workloads can obtain work from busy hardware threads. It is applied to graph processing benchmarks and it shows levels of performance comparable to HAP. Both techniques exploit cache coherence, possible between hardware threads mapped to the FPGA and software threads mapped to the CPU cores. We do not consider this strategy in this paper because the described work stealing mechanism can incur in duplicate processing of items at the beginning of the thief's new chunk. While this is not an issue for the graphs evaluated in [13], it may affect correctness in our benchmarks.

In this paper we base part of our approach on the system done in [5], where Xilinx SDSoC is used to generate the FPGA compute units for regular applications. We extend here this previous work by analyzing irregular parallel loops[2] that can be found in applications such as SPMM and propose a new scheduling strategy called FastFit better suited to make the most out of the FPGA features by minimizing the scheduling overheads. Moreover, the work in [5] targeted smaller Zynq devices consisting on a dual 32-bit Cortex A9 processor coupled to a small FPGA fabric. In here we target Zynq Ultrascale+ devices that offer a much higher performance platform based on four 64-bit ARM processors and a much larger FPGA fabric that enables the deployment of higher performance benchmarks.

2.1. Motivation for efficient asynchronous scheduling

In order to highlight the relevance of key aspects exploited by our approach, in particular the use of interruptions instead of a busy-waiting mechanism to feed the FPGA IPs, as well as the fact that we consider several FPGAs IPs, we refer to Fig. 2 were we plot the throughput achieved by the Static scheduling strategy for two of the benchmarks evaluated in this paper, HOTSPOT and GEMM (see Section 4) on our UltraScale+ platform with 4 CPU cores and 4 FPGA IPs.

The Static scheduling strategy requires a user defined argument, offload_ratio (from 0 to 1), that indicates the percentage of parallel iterations (from 0% to 100%) that should be offloaded to the FPGA. The FPGA includes 4 IPs running at 200 MHz, which can generate interruptions (IntOn, blue line) or not (IntOff, green dashed line) after finishing the computation. Without interruptions, 4 host-threads have to be busy-waiting until FPGA IP completion. This hampers the execution of the 4 CPU threads that are also processing part of the iterations on the CPU cores. With interruptions, the 4 host-threads are blocked while the FPGA IP is working and later woken-up by the interruption that the IP triggers once it has finished. From the figure we can see the positive impact of enabling interruptions and blocking host-threads: the whole platform can deliver up to 1,5x more throughput and the CPU cores can process a larger percentage of the iteration space (e.g. in HOTSPOT IntOn the CPU cores should process 40% of the iterations wheres with InOff, it is better if they only process 30% of the iterations). Since this feature has been already introduced in our framework and discussed elsewhere [5,14], from now on we consider that IntOn is the default mode for the rest of this work.

3. Heterogeneous parallel_for and schedulers

The Heterogeneous Building Blocks (**HBB**) library API is a C++ template library that facilitates exploiting heterogeneous platforms by automatically partitioning and scheduling the workload among the cores and the accelerator. The current version offers a `parallel_for()` function template, originally devised to exploit CPU+GPU platforms [4], which we extended to run on heterogeneous systems comprising CPU cores and OpenCL/SDSoC capable FPGAs [6].

HBB offers an abstraction layer that hides the initialization and management details of TBB and OpenCL constructs (contexts, command

[2] By irregular parallel loops we mean parallel loops that exhibit different computational load per iteration and that may access non-coalesced data of an irregular data structure.

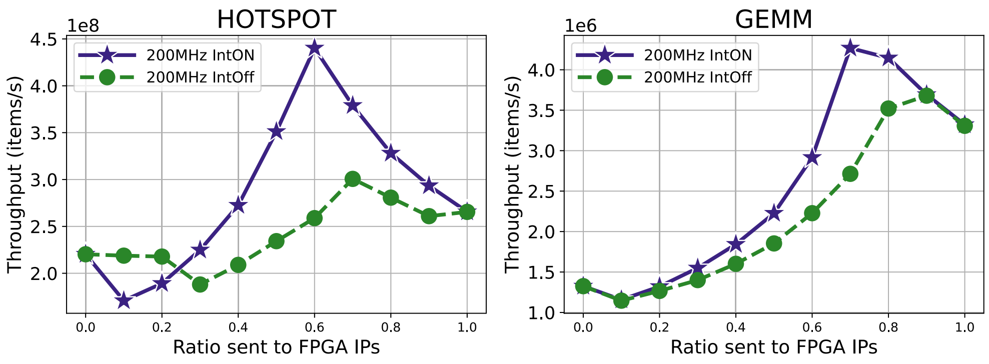


Fig. 2. Throughput of Static for different offload_ratio values. The higher the better.

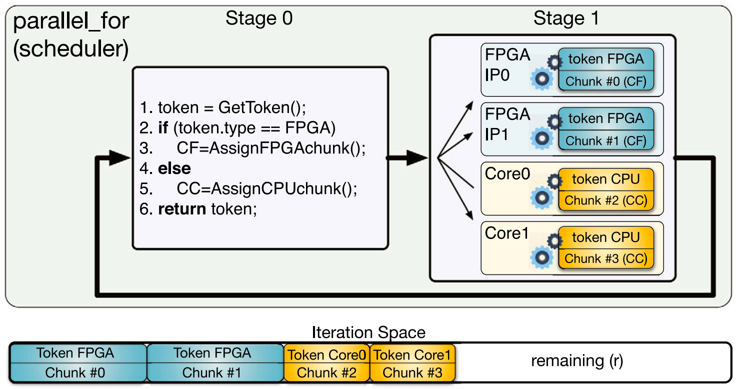


Fig. 3. Scheme of the scheduler engine implementation. Stage 0 performs the partitioning and scheduling, while Stage 1 computes the assigned chunks. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

queues, device_ids, etc.), thus users can focus on their own applications instead of dealing with thread management and synchronization. The engine managing the *parallel_for* pattern is implemented with a two-stage pipeline TBB template, which we explain next.

3.1. Scheduler engine implementation

We cover here how the scheduler engine is implemented in HBB and how the chunks that will be executed by the CPU cores and the FPGA IPs are assigned, depending on the selected scheduling (partitioning) strategy: Static, Dynamic, HAP and FastFit.

Fig. 3 depicts a simplified scheme of how the scheduler engine for our `parallel_for()` has been built. The engine is implemented with a two-stage pipeline TBB template with Stage 0 and Stage 1. Each token corresponds to a compute unit class, FPGA IP or CPU core in our system, and the sum of tokens represents the number of chunks of the iteration space that will be processed simultaneously. In the figure, we have two CPU tokens and two FPGA tokens being processed concurrently, two on the CPU cores and two more on the FPGA IPs. The first stage, Stage 0, gets an available token (line 1) and selects the chunksize for FPGA and CPU tokens, and extracts the corresponding chunk of iterations (CF and CC) from the set of remaining iterations, r (lines 2–6). The procedure to compute the FPGA and CPU chunksizes is covered in Section 3.2. Then, when a token reaches the second stage, Stage 1, the chunk gets processed on the corresponding device. The time required for the computation of the chunk on the FPGA and on a CPU core is recorded. This time[3] is used to update the relative speed of the FPGA w.r.t. a CPU core, that we call φ. Factor φ will be required to adaptively adjust the size of the next chunk assigned to a CPU core, because as we will explain next, our schedulers always try to balance the time consumed by accelerator and CPU chunks. Tokens are recycled until there are no remaining iterations. On behalf of understanding, in Table 1 we show a summary of the parameters included in the following model description.

[3] For the FPGA, the registered computation time includes the data transfer, when the platform does not support shared memory among devices. It could incorporate the host-to-device and device-to-host data communications times for the case of discrete accelerators.

3.2. FastFit Analytical model

The FPGA-specialized scheduling strategy that we propose in this paper, called *FastFit* (Fast Fitting), is based on an analytical model that can quickly estimate the throughput (e.g. iterations per sec.) for each FPGA IP when executing a chunk of parallel iterations. Our proposal uses an abstract pipeline model to obtain a near optimal FPGA chunksize, and with that, the corresponding CPU chunksize optimizing the throughput in both devices while ensuring load balancing.

Our hypothesis is that the effective FPGA throughput can be guessed by a model that assumes that the FPGA IP is implemented as a pipeline. This assumption leads to a good balance between accuracy and simplicity. More complex models could be more accurate, but their extra overhead may not payoff. Two latencies characterize the pipeline model: issue and completion. The *Issue Latency* (denoted by IL) is the number of cycles required between issuing two consecutive independent iterations. On the other hand, the *Completion Latency* (denoted by CL) is the number of cycles until the result of a parallel iteration is available. In other words, the issue latency is the time before the next independent iteration can be started, while the completion latency depends on the depth of the pipeline and is the time required to fill it up. In most cases, both latencies suffice to estimate the execution time of the FPGA kernel.

The scheduling strategy, as sketched in Algorithm 1, comprises two phases: the *Training Phase*, which finds near-optimal chunksizes for the

Table 1
Summary of the model parameters.

Acronym	Name	Description
CF	*FPGA Chunk*	**size of the FPGA chunk** (min(CF_ρ, r))
CC	*CPU Chunk*	**size of the CPU chunk**. Eq. (12)
IL	*Issue Latency*	Cycles between 2 consecutive parallel iterations
CL	*Completion Latency*	Cycles to complete 1 iteration
DL	*Depth Latency*	$DL = CL - IL$
F	*Frequency*	Clock frequency of FPGA kernel
δ	*second chunk size*	Number of iterations for second chunk just FPGA (Training Phase)
$t_{F_m}(\Omega)$	*Measured FPGA time*	Measured execution time of Ω iterations on the FPGA
$t_{C_m}(\Omega)$	*Measured CPU time*	Measured execution time of Ω iterations on the CPU
$t_{F_e}(\Omega)$	*Estimated FPGA time*	Estimated execution time of Ω iterations on the FPGA. Eq. (1)
$\lambda_{F_e}(\Omega)$	*Estimated throughput*	Estimated throughput of Ω iterations on the FPGA. Eq. (6)
$\lambda_{F_{peak}}$	*Peak throughput*	Estimated peak throughput on the FPGA with full pipeline. Eq. (7)
ρ	*Optimal threshold*	Ratio of the $\lambda_{F_{peak}}$ to be reached (0.9-0.99) at Exploitation Phase
CF_ρ	*Optimal chunksize*	FPGA chunksize for optimal throughput threshold. Eqs. (4), (5), and (9)
N_C	*CPU elements*	Number of elements processed in a single iteration on the CPU
N_F	*FPGA elements*	Number of elements processed in δ iterations on the FPGA
λ_C	*CPU throughput*	Estimated/Measured CPU throughput at Training/Exploitation Phase
λ_F	*FPGA throughput*	Estimated/Measured FPGA throughput at Training/Exploitation Phase
φ	*Relative speed*	Rrelative speed of the FPGA w.r.t a CPU core $\varphi = \frac{\lambda_F}{\lambda_C}$
r	*remaining iterations*	Pending iterations of the parallel_for loop

FPGA and CPU; and the *Exploitation Phase*, which uses the previously computed FPGA chunksize and adaptively modified the CPU chunksize, until the end of the iteration space.

Algorithm 1: FastFit scheduling strategy

```
   // First step: Training Phase
   Input: Frequency (F), second chunk size ($\delta$), optimal
          throughput threshold ($\rho$)
1  $t_{C_m}(1)$ = time_single_iteration_chunk_cpu();
2  $t_{F_m}(1)$ = time_single_iteration_chunk_fpga(); // Eq. (2)
3  $t_{F_m}(\delta)$ = time_multiple_iterations_chunk_fpga($\delta$); // Eq. (3)
4  $CF_\rho$ = compute_fpga_chunk_throughput_threshold($\rho$);
       // Eqs. (4), (5), and (9)
5  $CC$ = compute_cpu_chunk($CF_\rho$); // Eqs. (10), (11),
       and (12)
   // Second step: Exploitation Phase
   Input: $CF$, r, $\varphi$
6  while $r > 0$ do
       // Concurrent Execution while pending work
7      $CF = min(CF_\rho, r)$; // Compute_next_fpga_chunk
8      $CC = CF/\varphi$; // Compute_next_cpu_chunk:
           Eq. (12)
9  end
```

In the *Training Phase*, Lines 1–5, the scheduler offloads two chunks to the FPGA and one chunk to the CPU, recording the execution times. In lines 1 and 2, the first chunk for the FPGA and the CPU is made of 1 iteration each one. In Line 3, the second chunk, only for the FPGA, contains a representative number of parallel iterations δ (in our study we find that around 1%–5% of the iteration space is enough to characterize the FPGA throughput of our applications).

Let us suppose that we know the clock frequency of the FPGA (denoted by F and provided by the SDSoC compiler in a report file). When we offload a chunk of parallel iterations of size CF to the FPGA, then the time to complete them can be estimated as,

$$t_{F_e}(CF) = (CF \cdot IL + DL) \cdot \frac{1}{F} \tag{1}$$

where DL represents the number of cycles required to flush the pipeline, that can be computed as the difference between CL and IL, $DL = CL - IL$. By applying Eq. (1) to the two FPGA chunks of 1 and δ iterations, respectively, we obtain a system of two equations and two unknowns:

$$t_{F_m}(1) = t_{F_e}(1) = (IL + DL) \cdot \frac{1}{F} \tag{2}$$

$$t_{F_m}(\delta) = t_{F_e}(\delta) = (\delta \cdot IL + DL) \cdot \frac{1}{F} \tag{3}$$

As we know F, δ, $t_{F_m}(1)$ and $t_{F_m}(\delta)$, we can figure out IL and DL as,

$$IL = \frac{t_{F_m}(\delta) - t_{F_m}(1)}{\delta - 1} \cdot F \tag{4}$$

$$DL = t_{F_m}(1) \cdot F - IL \tag{5}$$

From previous equations, we can model the FPGA *estimated throughput* for a chunk CF of parallel iterations as,

$$\lambda_{F_e}(CF) = \frac{F}{IL + DL/CF} \tag{6}$$

Peak performance is attained with full pipelines, for which there are enough parallel iterations to effectively hide the completion latency. Therefore, latency hiding is achieved by executing a large enough chunk of independent iterations. Ideally, when the DL is completely hidden ($DL/CF \to 0$), then the issue latency determines the run time and we achieve peak performance. From Eq. (6) we compute the *peak performance* or *optimal throughput*, that we denote $\lambda_{F_{peak}}$ as,

$$\lambda_{F_{peak}} = \frac{F}{IL} \tag{7}$$

The goal of the *Training Phase* in our scheduler is to find a sufficiently large block of parallel iterations that guarantees that the estimated FPGA throughput is above a certain threshold of the peak performance, $\rho \cdot \lambda_{F_{peak}}$. Typically we will seek ρ values in the range $[0.9, 0.99]$, meaning that we aim for chunksizes that achieve throughputs that are within 90% and 99% of the peak performance. Larger FPGA chunksizes do not pay off, increase the probability of load unbalance and hinder CPU collaboration, as we corroborate in the Experimental Section. From Eqs. (6) and (7), and for a user-defined ρ, we know,

$$\frac{F}{IL + DL/CF_\rho} \geq \rho \cdot \frac{F}{IL} \tag{8}$$

In other words, the chunk of parallel iterations that guarantee a throughput above a ρ threshold of the peak, CF_ρ, can be computed as,

$$CF_\rho \geq \frac{DL}{IL} \cdot \frac{\rho}{1-\rho} \tag{9}$$

This step was summarized in Line 4 of Algorithm 1 where, using the execution times computed in Lines 2–3, Eqs. (4) and (5) can be computed. These results allow to solve Eq. (9) to get the near-optimal FPGA chunksize, CF_ρ.

Likewise, we can discover the optimal chunk for each CPU core from the FPGA chunk computed above, as can be seen in Line 5 of Algorithm 1. As input we take the execution time of one iteration in CPU $t_{C_m}(1)$ (measured in Line 1), and the number of elements, N_C, processed in that iteration. We also take as input N_F (the number of elements processed on the FPGA for the chunk of δ iterations). With that, we compute the throughput (elements per unit of time) of the CPU and the FPGA for both chunks as can be seen in Eqs. (10) and (11).

$$\lambda_C = \frac{N_C}{t_{C_m}(1)} \tag{10}$$

$$\lambda_F = \frac{N_F}{(CF_\rho \cdot IL + DL)/F} \tag{11}$$

As we said, the relative speed of the FPGA over the CPU is $\varphi = \frac{\lambda_F}{\lambda_C}$. It is advisable that the FPGA and CPU cores take the same time to compute their corresponding chunks, which results in the optimal CPU chunk size, CC, which can be computed as:

$$CC = \frac{CF_\rho}{\varphi} \tag{12}$$

After the first assignment of chunks CF_ρ and CC, the scheduler transitions to the *Exploitation Phase* in Lines 6–9 of Algorithm 1. During this phase, each thread entering Stage 0 (see Fig. 3), will invoke either Eq. (9) (FPGA token) or Eq. (12) (CPU token), in order to assign the corresponding FPGA and CPU chunks, CF, CC, respectively. After each chunk execution, the throughputs, λ_F, λ_C, and the corresponding relative speed, φ, are updated. When the number of remaining iterations, r, is not enough to feed the FPGA with CF_ρ iterations, then $CF = r$ (see Line 7) and CC is also recomputed so that the FPGA cores and the CPU cores finish computing their chunks at the same time.

As we validate in Section 5, FastFit is superior to our previous CPU+FPGA scheduling strategy, HAP [6]. The main difference is due to the reduction of the partitioning/scheduling overhead that FastFit achieves thanks to the new model. Firstly, FastFit only uses two FPGA samples at the beginning of the iteration space, which significantly reduces the time of the *Training Phase*. On the other hand, HAP, requires several FPGA samples (for monotonically increasing CFs) until the throughput stabilizes. With these samples, HAP computes a logarithmic fitting that is needed to model the FPGA throughput as a function of CF. Additionally, during the *Exploitation Phase*, HAP has to recompute each new FPGA chunksize using again a logarithmic fitting function, while FastFit only re-uses the near-optimal chunk already computed in its *Training Phase*.

3.3. Adapting the scheduler for several FPGA IPs

In previous works we have validated the three scheduling (partitioning) strategies that we have implemented in our HBB library to target heterogeneous systems [4,6]:

- *Static*: it splits the iteration space in two chunks at once: one for the CPU cores and the other for the accelerator. The size of these two chunks is user-defined and provided via the `offload_ratio` input argument. If `offload_ratio=0` (0%) the CPU process the whole iteration space, and so does the FPGA if it is equal to 1 (100%). The CPU chunk is divided in equally sized sub-chunks for each CPU core, i.e. `chunkCore=chunkCPU/NumCPUCores`.
- *Dynamic*: it lazily splits the iteration space dynamically. Each time the FPGA is idle, it takes a fixed-size chunk from the iteration space. The size of this chunk is user-provided using the `CF` input argument. The CPU cores also take chunks of the iteration space, but now they are adaptively computed as `CC=CF/`φ. A guided self-scheduling [15] is used when there are not enough remaining iterations to enforce the previous equations.
- *HAP*: it also dynamically splits the iteration space, but the user does not provide the `CF` size. On the contrary, this `CF` size is now an adaptive variable that is automatically computed by the scheduler following a logarithmic fitting strategy that has been proved beneficial for irregular codes on CPU+GPU [4] or CPU+FPGA [6] systems.

The previous evaluated implementations are able to deal with several CPU cores and a single accelerator (GPU or FPGA). However, the platform that we describe in the next section implements 4 FPGA IPs for each of the benchmarks that we use in the experimental validation. Thanks to the IP interruption (IntOn) mechanism that we have implemented on the device, each IP can process a chunk of iterations asynchronously at its own pace, which requires that we feed each IP as soon as it has completed the previous chunk.

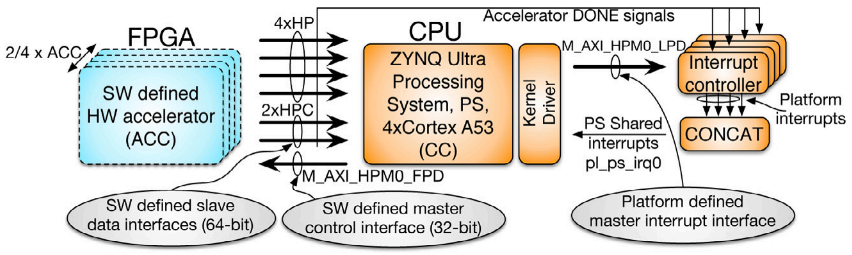
This new requirement is easily accomplished in our scheduler engine by considering the FPGA IPs as individual compute devices as we do with the CPU cores. This is, there is a TBB working thread for each device (CPU core or FPGA IP) that process in Stage 1 (see Fig. 3) the corresponding chunk (of size CC or CF, respectively). Thanks to the IntOn feature, the FPGA working threads (a.k.a. host-threads) are blocked most of the time (they awake just to offload a new chunk to the FPGA IPs), that way avoiding the oversubscription of the CPU cores. Thanks to that, we now have what we call "Multi" versions of the heterogeneous schedulers: MultiStatic, MultiDynamic, MultiHAP and MultiFastFit, which we evaluate in Section 5.

4. Experimental settings

4.1. Platform

As previously mentioned, the platform features the Xilinx Zynq Ultrascale+ SoC [16], which has a quad core ARM Cortex-A53, as well as an on-chip FPGA. Data transfer between the CPU and FPGA logic is done via AXI interface and the chip supports access via 4 High-Performance (HP) and 2 High-Performance-Cacheable (HPC) ports into CPU memory. The HPC memory buses are used in our implementations to support cache coherent shared memory between the CPU cores and FPGA IPs. Programming the FPGA is done via the SDSoC environment with optimized hardware accelerator implementations for the benchmarks that are used in the evaluation. A key component of the platform is the custom interrupt generation mechanism (IntOn) consisting of (i) hardware interrupt generators, which connect to the CPU IRQ lines and indicate when each hardware accelerator is finished; and (ii) software kernel-level drivers, which catch the interrupts and wake the host thread (the thread in charge of offloading work to the FPGA). The procedure to put the issuing threads to sleep and wake them up when the accelerator finishes work is as follows:

1. Load the corresponding accelerator bitstream onto the FPGA, which includes the (up to 4) interrupt controllers already connected to the `ap_done` output line of each accelerator (up to 4). The accelerators are also connected via the PS IRQ lines to the CPU.
2. Use `insmod` command to insert the interrupt driver kernel module (.ko) files to the system kernel.
3. The host (CPU) part of the workload/program opens the interrupt drivers, each interrupt driver listens for interrupt requests coming from the dedicated IRQ lines to the CPU.



FPGA
2/4 x ACC
SW defined HW accelerator (ACC)
4xHP
2xHPC
CPU
ZYNQ Ultra Processing System, PS, 4xCortex A53 (CC)
Kernel Driver
Accelerator DONE signals
M_AXI_HPM0_LPD
Interrupt controller
Platform interrupts
CONCAT
PS Shared interrupts pl_ps_irq0
M_AXI_HPM0_FPD
SW defined slave data interfaces (64-bit)
SW defined master control interface (32-bit)
Platform defined master interrupt interface

Fig. 4. SDSoC multiprocessing platform.

4. When the host program issues some work onto an accelerator, the issuing thread executes an `ioctl` call to the interrupt driver, which puts the current thread to sleep, until an interrupt event occurs on the monitored IRQ line.
5. When an interrupt occurs, i.e. the accelerator has finished computing the allocated workload, the current issuing thread gets woken up and returns to the scheduling thread, so we can keep issuing the next piece of work to the accelerator until the entire problem is solved.

Fig. 4 shows the hardware platform configurations, including the data access ports and the interrupt controllers. A key feature is that every FPGA IP has its own dedicated interrupt controller, interrupt driver and host thread so that each FPGA IP can perform independently in an asynchronous way. Moreover, the host thread does not waste CPU cycles waiting for the accelerator. This is needed because the standard mechanism deployed by the vendor tools (e.g. SDx, Vitis) with IPs obtained via high-level synthesis is to implement a busy-waiting thread that locks a CPU thread monitoring if the IP has finished. This is incompatible with our heterogeneous computing approach since, in essence, a core is wasted and becomes unavailable during this busy-waiting time. The platform set-up is open-source under the BSD-3 license and all the relevant files can be found online [17].

In order to measure the energy consumption in this platform, we rely on Ubuntu and the server part of the pmlib library [18]. This pmlib server is a thread that samples the INA266 power monitors at 200 Hz so that power readouts are available from the benchmark. The underlying linux kernel driver is configured to sample the power monitors at the maximum programmable rate (every 2 ms). The library monitors a total of 12 power rails, 7 for the PL-side (Programmable Logic/FPGA) and 5 for the PS-side (Processing Subsystem/ARM Cortex-A53).

4.2. Benchmarks

This subsection describes our benchmark collection. This collection comprises both integer and floating point applications from different domains: linear algebra (Sparse Matrix Matrix Multiplication-SPMM, Dense Matrix Matrix Multiplication-GEMM), cryptography (AES), physics simulation (HOTSPOT-HS). More information on the implementation of the benchmarks can be obtained online [17].

HS HOTSPOT [19] is a well known application from the Rodinia benchmark collection used to estimate processor temperature based on an architectural floorplan and simulated power measurements. The thermal simulation iteratively solves a series of differential equations. Each output cell in the computational grid represents the average temperature value of the corresponding area of the chip. In these experiments an input of 1024×32768 points was considered. HOTSPOT main computation consists in applying a 2D filter of size 3×3 to all temperature values organized in a 2D array representing the chip surface. The Vivado HLS implementation stores a 2D patch of temperatures values in internal FPGA memory organized as a logical shift register that receives a new temperature value as the filter moves along the patch. Data is organized so that temperature values disregarded from the shift register are not needed again and a temperature value is computed per clock cycle.

GEMM General Matrix Multiplication is the dense matrix matrix multiplication, well suited for parallel computing since all the multiplications of the row and column elements can be done in parallel. A naïve implementation in hardware will simply buffer matrix A and B in the FPGA, and it will then compute the multiplication. This is effective, but it is very resource-intensive and can quickly overwhelm the BRAM and DSP resources available in the considered device. An alternative is to buffer only one row of A and a tile of B formed by a number of complete columns BW and then work in the multiplication in parallel. The amount of parallelism is defined by the width of BW, with higher values meaning that more DSP blocks work in parallel. The main constrain is that memory allocated in the FPGA needs to be sufficient to store the area BW. There is a trade-off between the largest matrix supported, and the amount of parallelism exploited.

AES This benchmark comes from the Hetero-Mark suite [20]. It implements the Advanced Encryption Standard (AES) algorithm. The program takes plain text as input and encrypts it using a given encryption key. In our experiments we took an input text of 250 MB and a key of 256 bits. The AES considered in this work consists of an initial addroundkey function and then a total of 14 rounds that apply functions shiftrow, subbytes and mixcolumn to input data blocks of 16 bytes, finally producing 16 bytes of cypher text. The proposed Vivado HLS description consists of a full dataflow implementation with a fully unroll main loop and independent functions than write and read from 16-byte buffers. This approach results in very high performance since data streams with minimum latency through dedicated hardware for each unrolled function. We have used the ARMv8 Cryptographic Extension for AES [21] trying to get the best possible performance on the CPU side.

SPMM Sparse Matrix Matrix Multiplication is a sparse matrix-dense vector multiplication from Bell and Garland work [22], also representing an irregular application since each row has a different number of non-zeros. The *mix-tank_new* sparse matrix (dimension of 29957×29957) from the University of Florida Sparse Matrix Collection that exhibits a diagonal-like profile was selected as input. The SPMM hardware is based on our previous work of an SPMV (sparse matrix dense vector) kernel presented in [23]. The SPMM accelerator has been designed to support the popular CSR (compressed store row) format to store sparse matrices, and it avoids

Table 2
Details and characterization of the benchmarks.

Benchmark	Iteration size	Iteration space	Granularity (s. per iter.)	Throughput unit
HOTSPOT	1024 doubles	32 K iter.	1.75 x 10^{-5}	elements/s
GEMM	1024 doubles	16 K iter.	3.05 x 10^{-3}	elements/s
AES	16 bytes	16 000 K iter.	8.16 x 10^{-8}	bytes/s
SPMM	66.5 doubles on average	29 957 iter.	6.8 x 10^{-5}	non-zeros/s

having to buffer full chunks of the input matrices using a streaming dataflow architecture. It consists of an input stage, streaming mapping layers, compute stage, streaming mapping layers and output stage. The streaming mapping layers reformat and distribute the data received from its input buffers among its output buffers that then feed the compute stage. To accelerate the multiplications on the CPU, the CPU cores call the highly tuned ARM Performance Libraries sparse routines [24].

For HOTSPOT, GEMM and SPMM, we parallelize the traversal of the rows of the matrices (each row is a parallel iteration). For AES, each parallel iteration encrypts a 16-byte block. Table 2 contains some relevant details of the benchmarks that will be discussed and analyzed in the next section.

5. Experimental results

In this section, we evaluate different aspects of MultiFastFit: from the accuracy of the model, FastFit, on which is based and its ability to estimate the near optimal chunksize (and its impact in the throughput), to the scalability and energy efficiency of the heterogeneous co-executions delivered by our scheduler. We also include a performance comparison of MultiFastFit vs. state-of-the-art heterogeneous schedulers.

5.1. Analysis of MultiFastFit

In our framework, MultiFastFit incorporates two parameters that can be tuned according to the particularities of the applications. One of them is δ, the ratio to set the number of iterations used for recording the time of the second chunk in the *Training Phase* (Eq. (3)). If this parameter is not provided by the user, by default our scheduler assigns $\delta = 0.1$ (1% of the iteration space). The other parameter is ρ, the threshold to compute the near optimal FPGA chunksize (Eq. (9)). We conducted a sensitive study consisting of an exhaustive exploration of $\rho = \{0.9, 0.95, 0.99\}$ and $\delta = \{0.1, 0.2, 0.5\}$ analyzing their impact on the performance of our benchmarks. Higher values of ρ report bigger FPGA chunksizes (see Eq. (9)), which can improve slightly FPGA performance during the Exploitation Phase but at the cost of generating a smaller number of chunks to feed the FPGA IPs, as well as leaving less workload to the CPU cores, which in some cases (GEMM and SPMM) produced some load imbalance at the end of the Exploitation Phase when all FPGA IPs and CPU cores were working. For instance, on average 15% of throughput degradation was observed for $\rho = 0.99$ in GEMM when compared to $\rho = 0.95$. For HOTSPOT and AES, both $\rho = 0.95$ and $\rho = 0.99$ reported similar performance. We also observed slightly smaller performance for $\rho = 0.9$ in all benchmarks, although degradation was below 3% w.r.t the maximum measured throughput. On the other hand, once ρ had been selected, δ values did not have noticeable impact on performance, although $\delta = 0.05$ (5% of the iteration space) consistently provided stable times to robustly compute IL (Eq. (4)) and DL (Eq. (5)) for all our benchmarks. As a summary we found that $\rho = 0.95$ and $\delta = 0.05$ provided a good trade-off, thus, these are the values that are used by default in our suite of experiments. Also, FPGA frequency was set up at the maximum supported by the IPs, 200 Mhz, because we wanted to study the efficiency of our approach at maximum performance.

5.1.1. Accuracy and optimal scalability of the model

Fig. 5 provides two set of results for each benchmark. At the bottom, we illustrate the near optimal chunksize that our model reports at the end of the *Training Phase* (yellow dashed line) vs. the average chunksize measured (green dashed line) for different configurations. The chunksize represents a block of parallel iterations. Note that each parallel iteration of a chunk contains several items, which depends on the benchmark (row size on the temperature matrix in HOTSPOT, row size for GEMM, average row size for SPMM, or the size of an encrypted block of 16 bytes in AES; please see Table 2 for more details). A configuration given by $y|x$, indicates y CPU cores and x FPGA IPs. In particular, for each group of lines we study the accuracy of our model and the optimal scalability of MultiFastFit for 5 different experiments. On each experiment, the number of CPU cores is fixed (from 0 to 4), but we incrementally incorporate additional FPGA IPs (from 1 to 4). So, the leftmost group of lines $(0|x)$ represents the FPGA-only homogeneous executions on x FPGA IPs, while the remaining groups represent heterogeneous co-executions. In fact, the starting point for any of these remaining 4 groups $(y|0)$ represents CPU-only homogeneous executions on y CPU cores.

At the top of Fig. 5 we depict the throughput that MultiFastFit estimates (yellow solid line) vs. the actual average throughput measured (green solid lines) for the corresponding configurations. The throughput is represented as items/s for HOSTSPOT, GEMM, SPMM, and bytes/s in AES. The modeled throughput (yellow solid line) is computed from Eqs. (10) (λ_C) and (11) (λ_F) once MultiFastFit has found the near optimal chunksize (CF_ρ) in the *Training Phase* for each configuration. As these equations just compute the throughput for one CPU or one FPGA IP respectively, the throughput for any configuration $y|x$ can be finally modeled as: $y \cdot \lambda_C + x \cdot \lambda_F$, and it represents the aggregated near-optimal throughput for each configuration.

From Fig. 5 we see that MultiFastFit is able of improving performance when the number of compute devices increases, and heterogeneous CPU+FPGA co-executions that exploit all the compute devices are usually faster. But let us study each benchmark in detail. For both GEMM and HOTSPOT, FastFit predicts a stable chunksize on any configuration, as we see in Fig. 5. In these codes, the average measured chunksize is above the predicted one. In any case, the difference between the modeled and the average measured chunksize is lower than 1% for HOTSPOT or 10% for GEMM. This difference is caused by the size of the second chunk explored in the *Training Phase* (5% of the iteration space, which is much bigger than the finally modeled chunksize). In particular, in GEMM the maximum size of the matrix that can be allocated to perform the experiments is 16Kx1024 doubles, due to board coherent memory constraints. As the near-optimal chunksize is around 430 iterations (rows in this case), increasing the number of IPs reduces the number of chunks that can be assigned to each one, so incorporating the training chunk (around 819 iterations) to the average tends to result in higher average chunksizes. This effect is more noticeable when the number of CPU cores increases, because then fewer iterations and therefore fewer chunks will be left to the FPGA IPs.

In any case, chunksizes near or above the modeled values should guarantee near optimal throughput. In fact, for both GEMM and HOTSPOT the measured throughput and the estimated one are close on any configuration. Also note that for each experiment once the number of CPU cores is fixed, increasing the number of IPs does not affect the size of the estimated near optimal chunk. These results hint that for each experiment, executions should scale well when increasing

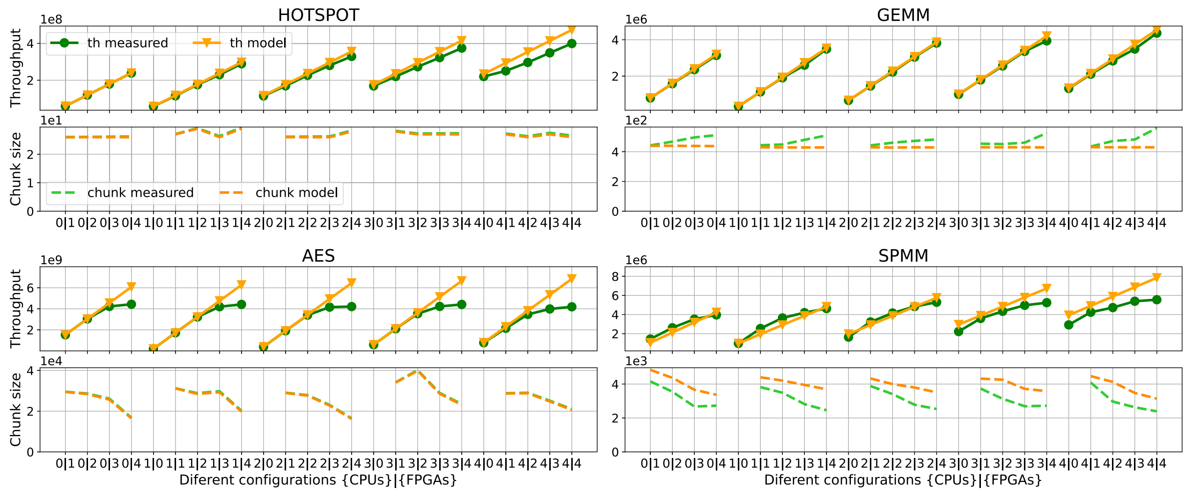


Fig. 5. Modeled throughput vs. average measured throughput on different configurations (solid lines). Near optimal modeled chunksize vs. average measured chunksize (dashed lines). For HOTSPOT, GEMM, SPMM the throughput is in items/s, and the chunksize in rows. For AES the throughput is in bytes/s and the chunksize in 16-byte blocks. The higher the throughput the better. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the number of IPs. In fact, the measured throughput scales similarly to the modeled one, and it degrades slightly within the range [1%, 4%] w.r.t the estimated one for GEMM, and within [2%, 15%] for HOTSPOT. The degradation is more apparent in HOTSPOT due to the finer granularity of the FPGA chunks (below 100 μs) what exposes the overhead of MultiFastFit particularly when the number of compute resources increases. On the contrary, the granularity of a chunk in GEMM is around 100 ms, enough to conceal the scheduler overhead.

AES is another very fine-grained application (chunks take less than 60 μs), but contrary to HOTSPOT, AES is memory bound. In particular, the two HPC memory buses of the board saturate when the number of IPs working concurrently is higher than 3, which causes FPGA starvation. This is noticeable in Fig. 5 when comparing the estimated throughput and the measured one on any experiment. Interestingly, the size of the chunk predicted for the model and the average measured one are close (differences are lower than 1%), indicating that the *Training Phase* does not have an impact. However, for any experiment we notice that the size of the modeled chunksize drops when the number of compute IPs is precisely higher than 3. Lower chunk values mean sub-optimal sizes. These lower values are a consequence of sub-optimal throughput measured in the *Training Phase*, when all the devices are competing for the saturated HPC bus bandwidth. In any case, this saturation is the main responsible for the loss of scalability of the measured throughput when compared to the estimated one, which assumes optimal linear scalability. The second culprit is the oversubscription in the CPU cores since the 4 FPGA host-threads awake every 60 μs. to feed the FPGA and collide with the computation of chunks on the CPU cores. For instance, the degradation of the measured throughput drops to 37% in the 0|4 FPGA-only configuration (4 IPs), and even to 63% in the 4|4 CPU+FPGA heterogeneous configuration.

SPMM is also a memory bound application, but with a higher granularity than AES or HOTSPOT (a chunk takes around 56 millisec.). Additionally is an irregular benchmark. As a memory bound application, we notice that increasing the number of compute IPs causes the reduction of the size of the near optimal chunk reported by the model. This has an effect in the measured throughput when compared to the estimated one. For instance, for the homogeneous 0|4 FPGA-only configuration, the measured throughput degrades 7% w.r.t. the estimated one. Also the degradation is clearly visible in Fig. 5, where the average measured throughput tends to flatten out and not scale up when the number of IPs increases on the heterogeneous CPU+FPGA experiments. There is also another issue related again to the board memory constraints. The maximum size of the sparse matrix that can be allocated to perform the experiments has 29957 rows. As the near optimal chunksize reported is around 3000–4000 iterations (rows in this case), increasing the number of compute devices reduces the number of chunks that can be assigned to each IP. So when the final step of the *Exploitation Phase* of our scheduler assigns a smaller (sub-optimal) chunksize with the last remaining iterations, this has the effect of reporting smaller average measured chunksizes, as can be seen in Fig. 5 for the heterogeneous CPU+FPGA experiments (dashed green lines). This problem is exacerbated when the number of CPU cores increases, because then fewer iterations (and chunks) will be available for the FPGA IPs, which coupled with the irregular nature of the few available chunks will result in load unbalance between the devices in that final step. This issue is visible in the figure for the $3|y$ or $4|y$ heterogeneous experiments, where the average measured throughput is always below the estimated one. Summarizing, the reduction of the near-optimal chunksize (sub-optimal size) due to the HPC bus congestion and the load unbalance between devices caused by the few irregular chunks explain the important degradation of throughput in these experiments. For instance, for the 3|4 and 4|4 configurations the measured throughput drops by 28% and 40%, respectively, when compared to the corresponding predicted throughput.

5.1.2. *Exploring ideal scalability*

While the previous section discusses the sensitivity of our model to find the near-optimal chunksize for each configuration, and the impact that different sources of overhead have in the scalability of the performance when compared to an optimal linear model, in this section we discuss in more detail the heterogeneous CPU+FPGA experiment in which our benchmarks achieve the maximum performance: $4|x$. Now, we compare the average measured throughput against an aggregated ideal throughput computed as the sum of the measured throughput in homogeneous x FPGA (FPGA-only, where x goes from 1 to 4) and 4 CPU-only executions. Fig. 6 reports the measured and ideal throughputs (from 5 to 8 compute units). Also, the FPGA-only (x FPGA IPs) and CPU-only (y CPU) throughputs are shown for reference: 1 to 4 are CPU cores in a blue line, or FPGA IPs in an orange line; green lines are for 4 CPU cores plus 1 to 4 FPGA IPs; the ideal throughput is represented with a dark green dotted line (4 (CPU + x FPGA ideal).

In Fig. 6 we see that for HOTSPOT and GEMM, the FPGA-only experiments clearly scale from 1 to 4, while for AES and SPMM there is a loss of scalability beyond 3 IPs due to the saturation of the HPC memory buses, as commented in the previous section. This behavior is projected on the 4 CPU + x FPGA ideal results, factoring out this way

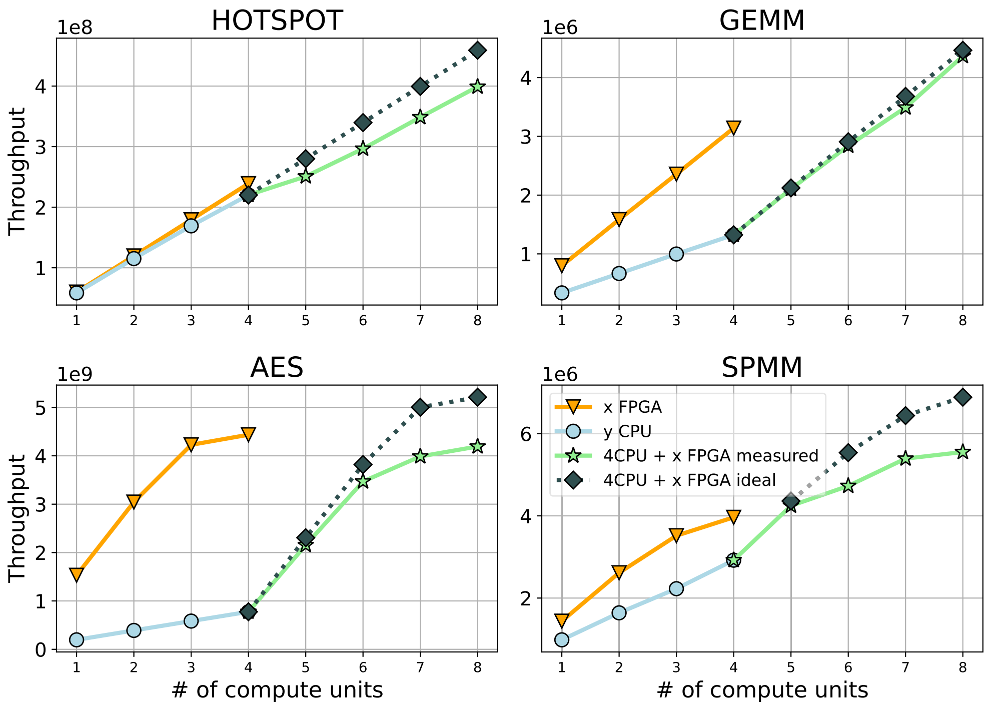


Fig. 6. Average measured throughput for MultiFastFit vs. ideal throughput on heterogeneous 4CPU+xFPGA. As baseline, average measured throughput for homogeneous CPU-only and FPGA-only configurations. For HOTSPOT, GEMM, SPMM the throughput is in items/s, while for AES is in bytes/s. The higher the throughput the better. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the effect of the bus saturation in the performance of the heterogeneous experiment. For fine-grained benchmarks as HOTSPOT and AES, the degradation of throughput is due mainly to the scheduler partitioning overhead (Stage 0 of Fig. 3). This overhead can represent a 15% (20%) of loss of efficiency in HOTSPOT (AES). In the case of SPMM the degradation is due to a very small number of sub-optimal chunks that produce load unbalance between the CPU cores and the FPGA IPs (as commented in the previous section), which has a greater impact as the number of compute units increases. In this benchmark the throughput is degraded by up to 24% w.r.t. the ideal.

In any case Fig. 6 also shows that the heterogeneous 4CPU+4FPGA configuration is the one that achieves the highest performance for HOTSPOT, GEMM and SPMM, being the improvement 1,7x, 1,4x and 1,4x respectively w.r.t. the FPGA-only 4 FPGA, while for AES there is a slight degradation of 0,94x.

5.1.3. Exploring energy efficiency

Once we have studied the performance and quantified the loss of efficiency due to various scheduler overheads in the $4|x$ heterogeneous experiment, our next step is to study its energy efficiency.

Fig. 7 depicts energy efficiency as items/J or bytes/J for the heterogeneous 4CPU+xFPGA experiment. Also, the FPGA-only (x FPGA IPs) and CPU-only (y CPU cores) energy efficiencies are shown for reference: 1 to 4 are CPU cores in a blue line, or FPGA IPs in an orange line; the green line is for 4 CPU cores plus 1 to 4 FPGA IPs. Similarly, as stated in the previous section, the heterogeneous 4CPU+4FPGA configuration is the one that achieves the highest energy efficiency for HOTSPOT, GEMM and SPMM, being the improvement 1,4x, 1,1x and 1,3x respectively w.r.t. the 4 FPGA (FPGA-only), while for AES there is a degradation of 0,92x.

In general, configurations that achieve the highest throughput are also the ones that obtain the best energy efficiency. However, we have also explored the possibility of changing the FPGA operational frequency, from the highest, 200 MHz, to the lowest, 25 MHz, or even clock-gating the FPGA (0 Hz) when it is not used in CPU-only executions. This study can be of interest for memory bound applications for which a power capping might be required. AES benchmark is a interesting case of study for this scenario. Fig. 8 shows the impact in the throughput (bytes/s) and mean power (watts) for AES and different compute units: 1 to 4 are CPU cores in bluish lines, or FPGA IPs in orange lines; green lines are for 4 CPU cores plus 1 to 4 FPGA IPs. Solid lines are for the FPGA running at 25 MHz and dashed lines for the FPGA at 200 MHz.

From Fig. 8 we notice that when the FPGA is not being used is better to clock-gate it, because we can reduce power consumption in more than 1 W (more than 30% of saving). Other important observation is that changing the FPGA operational frequency to 25 MHz significantly reduces the power consumption of the board: by 1/2 on the 4 FPGA configuration and by 1/1,7 on the 4CPU+4FPGA one. Interestingly, although at 200 MHz AES does not scale beyond 3 IPs due to the HPC memory bus contention, at 25 MHz both the x FPGA and the heterogeneous 4CPU+xFPGA experiments scale linearly, being the configuration 4CPU+4FPGA the one that achieves the highest throughput and highest energy efficiency at that operational point. In other words, at that frequency the benchmark is not bound by memory accesses. Assuming a power capping of 4 Watts, we can conclude that for AES, the heterogeneous configuration that exploits all the compute units is the optimal solution, both from the performance and energy efficiency points of view.

5.2. Comparison of MultiFastFit with state-of-the-art heterogeneous schedulers

Fig. 9 shows HOTSPOT, GEMM, AES and SPMM for MultiStatic, MultiDynamic, MultiHAP and MultiFastFit schedulers. Y-axes show the throughput measured as items/s or bytes/s, thus the higher the better. X-axes represent the FPGA chunksize, CF, measured in number of parallel iterations (rows in all benchmarks but in AES for which each parallel iteration process a 16-byte-block). Bluish lines are MultiDynamic executions in which only the CPU is exploited (from one, 1C, to four, 4C, CPU cores). As we can see, these CPU-only executions do not depend on the FPGA chunksize. Greenish lines represent all possible

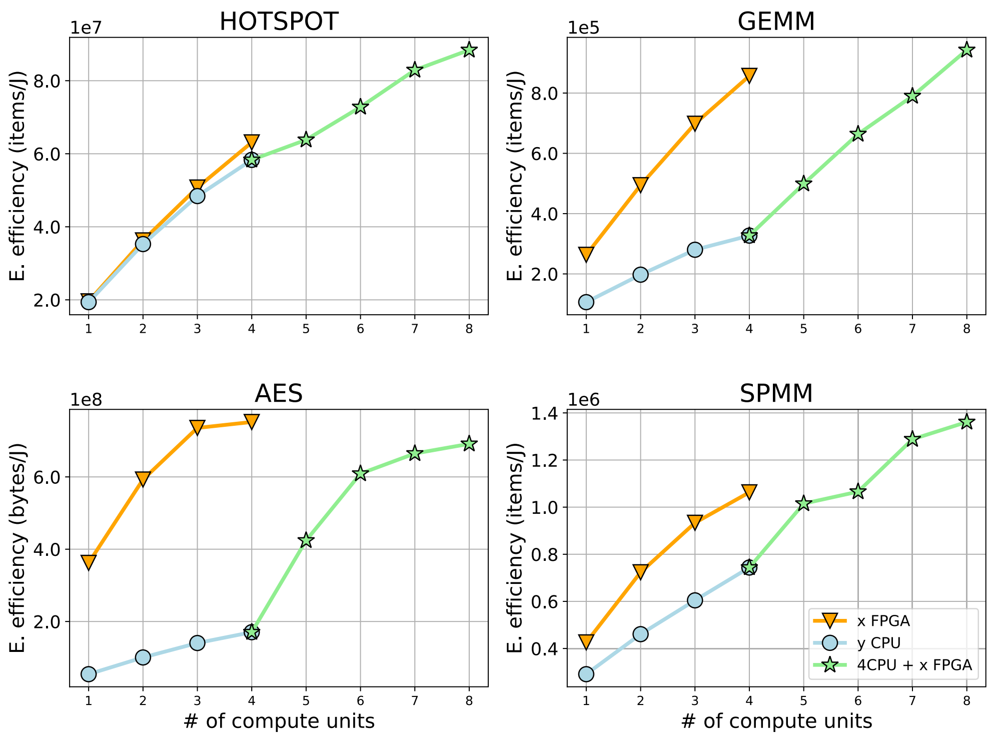


Fig. 7. Energy efficiency for MultiFastFit on heterogeneous 4CPU+*x*FPGA. As baseline, energy efficiency for homogeneous CPU-only and FPGA-only configurations. For HOTSPOT, GEMM, SPMM the efficiency is in items/J, while for AES is in bytes/J. The higher the better. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

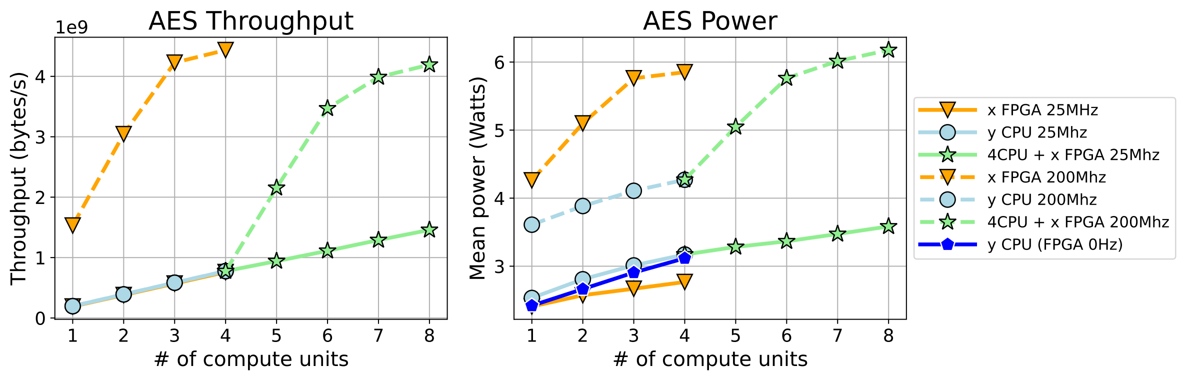


Fig. 8. Throughput and power consumption for AES when setting the FPGA operation frequency at 25 MHz vs. 200 MHz on different configurations. On LHS: throughput, the higher the better, on the RHS: power, the lower the better. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

configurations in which the 4 FPGA IPs are exploited. They include: the FPGA-only execution with 4 FPGA IPs (4F), and the CPU+FPGA heterogeneous configurations with 4 FPGA IPs and one to four CPU cores (1C+4F to 4C+4F). These results for the MultiDynamic scheduler shed light on the relative performance of the CPU and the FPGA and how chunksizes affect the throughput. In heterogeneous executions very large chunks cause load unbalance among the devices, while sizes too small produce sub-optimal performance on the accelerator. From the figure we also discover the FPGA chunksizes at which maximum throughput is achieved. Note however, that MultiDynamic requires an offline chunksize exploration in order to find the optimal chunk. In this study we explore 2^i sizes, but a finer step exploration may be necessary for other applications.

The yellow star, ☆, indicates the throughput that can be achieved with the MultiStatic scheduler. Remember that, as we saw in Fig. 2, for MultiStatic, the user has to manually find the best ratio of the iteration space that is offloaded to the FPGA (while the rest is computed on the CPU). MultiStatic performs a single offload operation: the FPGA fraction of the iteration space is evenly split between the 4 FPGA IPs, much as the CPU fraction is divided between the CPU cores. This minimizes the scheduler overhead (Stage 0 of Fig. 3 is invoked only once) but at the price of the offline exploration for the best partition ratio. The X-axes position of the yellow star indicates the FPGA chunksize that corresponds to the best ratio (found by traversing all the ratios from 0% to 100% in 10% steps). That way, this MultiStatic scheduler can be considered both an oracle-like and a brute-force approach.

The orange star, ★, and red star, ★, indicate the throughput achieved by MultiHAP and MultiFastFit, respectively. Now, the X-coordinate of those stars is the average size of all the FPGA chunks, CF, that have been offloaded to the four IPs. This includes the chunks

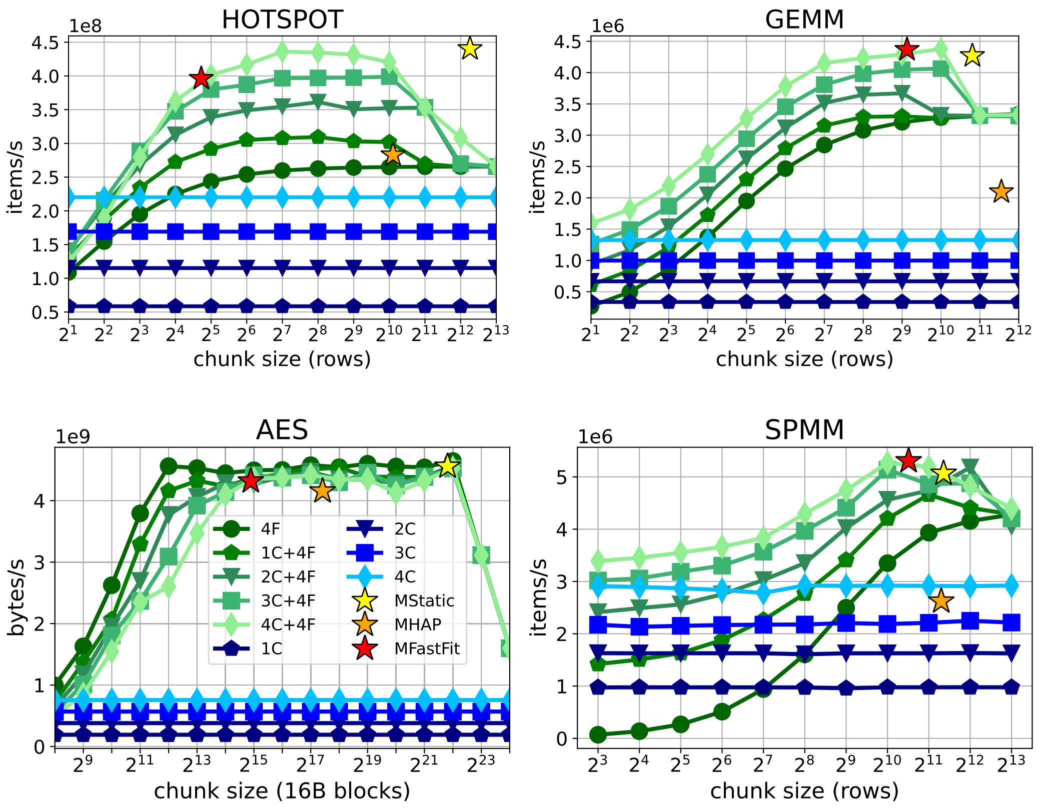


Fig. 9. Throughput comparison for MultiFastFit vs. state-of-the-art schedulers. The higher the better. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

in the *Exploration* and *Exploitation Phases*, as well as the last chunk that, at the end of the iteration space, can be smaller (sub-optimal) than the one obtained by the models (HAP or FastFit).

Summarizing, the heterogeneous CPU+FPGA executions have the potential of being faster than the CPU-only and FPGA-only executions. The exception is the AES benchmark whose dataflow architecture maps optimally onto the FPGA. In that case, adding CPU cores to try to help in the computation slightly reduces the throughput. As explained previously, this heavily memory bound application is working with a congested memory bus when the number of FPGA IPs is higher than 3 and adding CPU cores does not help to mitigate this problem. Moreover, given the very fine granularity of the benchmark any scheduler partitioning stage introduces not negligible overhead.

Although the manually-tuned MultiDynamic and MultiStatic can result in better throughputs, as in HOTSPOT and AES, the auto-tuned MultiHAP and MultiFastFit are not far from the best throughput or can even outperform MultiStatic. MultiFastFit is always better than MultiHAP: 40% in HOTSPOT, 108% in GEMM, 4% in AES and 102% in SPMM. For GEMM and SPMM the main source of overhead of HAP is its *Training Phase* that needs several samples to perform the fitting and find the near-optimal chunk. For HOTSPOT and AES the main source of overhead is now the cost of the logarithmic fitting computation needed each time that a new FPGA chunk is assigned. Our new FPGA-oriented proposal, MultiFastFit, is never more than 9% slower than the best manually-tuned alternative (the worse case is HOTSPOT), it can be up to 5% faster than MultiStatic (the best case is SPMM), and more importantly it does not require an offline search of the ideal CPU-FPGA partition or FPGA chunk granularity.

6. Conclusions

In this paper we propose a novel lightweight scheduling strategy, FastFit, targeted at FPGA accelerators, and a new scheduler based on it, named MultiFastFit, which asynchronously tackles heterogeneous systems comprised of a variety of CPU cores and FPGA IPs. Our strategy significantly reduces the overhead to automatically compute the near-optimal chunksize when compared to a previous state-of-the-art auto-tuned approach, which makes our approach more suitable for fine-grained applications. Additionally, our scheduler MultiFastFit has been designed to enable the efficient co-execution of work among compute devices in such a way that all the devices are busy while minimizing the load unbalance.

Our approaches have been evaluated using four benchmarks carefully tuned for the low-power UltraScale+ CPU+FPGA platform, which is our system of choice. Our experiments demonstrate that the FastFit strategy always finds the near-optimal chunksize for any device configuration at a reasonable cost, even for fine-grained and irregular applications, and that scalability is ensured provided that the HPC memory bus is not saturated or there are enough parallel iterations to supply near-optimal chunks to all the devices. We have found that heterogeneous CPU+FPGA co-executions that exploit all the compute devices are up to 1,7x faster and 1,4x more energy efficient than the FPGA-only executions.

We have also compared MultiFastFit with other state-of-the-art scheduling strategies. Comparing the two auto-tuned schedulers studied, we have found that MultiFastFit outperforms MultiHAP up to 2x. Also, the results show us that MultiFastFit achieves similar results to manually-tuned schedulers such as MultiStatic or MultiDynamic: in the worst case it is only 9% slower than the best manually-tuned alternative.

This work focuses on scheduling the workload of a single benchmark such that it can run in parallel with CPU and FPGA resources on different data inputs. We do not consider optimization techniques for FPGA runtime reconfiguration that fall outside the scope of this work. In the general case that different and independent benchmarks must be run (e.g. GEMM and Hotspot) in parallel then the OS can perform this function using CPUs and FPGA IPs as long as the independent accelerators for each kernel are implemented in the FPGA.

Future work involves applying MultiFastFit to other irregular applications such as Graph processing. Complex graphs contain up to trillions of edges that combined with their potentially irregular nature pose significant challenges for a heterogeneous runtime scheduler. For example, we would like to explore periodically launching the training phase to capture changes in the data input that produce computational irregularities. How often this should be done or when to trigger the activation of the training phase is still an open question. We also plan to apply the current GEMM and SPMM heterogeneous benchmarks to sparse neural network problems. In this scenario the amount of sparsity in the network layers varies and the distribution of work between the dense and sparse CPU-FPGA kernels can be adjusted by the scheduler at runtime

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially supported by the Spanish projects PID2019-105396RB-I00, UMA18-FEDERJA-108, and UK EPSRC projects ENEAC (EP/N002539/1), HOPWARE (EP/V040863/1) and RS MINET (INF\R2\192044). Funding for open access charge: Universidad de Málaga / CBUA.

References

[1] The Khronos SYCL Working Group, SYCL 2020 specification (revision 3), 2021.
[2] Intel, Intel oneAPI programming guide, 2020.
[3] M. Voss, R. Asenjo, J. Reinders, Pro TBB: C++ Parallel Programming with Threading Building Blocks, APress, 2019.
[4] A. Navarro, F. Corbera, A. Rodriguez, A. Vilches, R. Asenjo, Heterogeneous parallel_for template for CPU–GPU chips, Int. J. Parallel Program. (2018).
[5] J. Nunez-Yanez, S. Amiri, M. Hosseinabady, A. Rodríguez, R. Asenjo, A. Navarro, D. Suarez, R. Gran, Simultaneous multiprocessing in a software-defined heterogeneous FPGA, J. Supercomput. (2018).
[6] A. Rodríguez, A.G. Navarro, R. Asenjo, F. Corbera, R. Gran, D.S. Gracia, J.L. Núñez-Yáñez, Parallel multiprocessing and scheduling on the heterogeneous Xeon+FPGA platform, J. Supercomput. 76 (6) (2020) 4645–4665.
[7] J. Bueno, J. Planas, A. Duran, R.M. Badia, X. Martorell, E. Ayguade, J. Labarta, Productive programming of GPU clusters with OmpSs, in: Intl. Parallel and Distributed Processing Symp., IEEE, 2012, pp. 557–568.
[8] P. Pandit, R. Govindarajan, Fluidic kernels: Cooperative execution of OpenCL programs on multiple heterogeneous devices, in: Int. Symp. on Code Generation and Optimization, 2014.
[9] P. Meng, M. Jacobsen, R. Kastner, FPGA-GPU-CPU heterogenous architecture for real-time cardiac physiological optical mapping, booktitle=intl. conf. on field-programmable technology, fpt'12, pages=37–42,, 2012.
[10] S. Prongnuch, T. Wiangtong, Heterogeneous computing platform for data processing, in: Int. Symp. on Intelligent Signal Processing and Communication Systems, 2016, pp. 1–4.
[11] K.H. Tsoi, W. Luk, Axel: A heterogeneous cluster with FPGAs and GPUs, in: Intl. Symp. on Field Programmable Gate Arrays, in: FPGA '10, 2010, pp. 115–124.
[12] A. Rodríguez, A. Navarro, R. Asenjo, F. Corbera, R. Gran, D. Suárez, J. Nunez-Yanez, Exploring heterogeneous scheduling for edge computing with CPU and FPGA MPSoCs, J. Syst. Arch. 98 (2019) 27–40, http://dx.doi.org/10.1016/j.sysarc.2019.06.006, URL https://www.sciencedirect.com/science/article/pii/S1383762119300918.
[13] M. Agostini, F. O'Brien, T. Abdelrahman, Balancing graph processing workloads using work stealing on heterogeneous CPU-FPGA systems, in: 49th Intl. Conf. on Parallel Processing - ICPP, 2020.
[14] K. Nikov, M. Hosseinabady, R. Asenjo, A. Rodriguez, A. Navarro, J. Nunez-Yanez, High-performance simultaneous multiprocessing for heterogeneous System-on-Chip, in: 13th International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG-2020), Bologna, Italy, 2020, URL https://arxiv.org/abs/2008.08883.
[15] D. Rudolph, C. Polychronopoulos, An efficient message-passing scheduler based on guided self scheduling, in: 3rd Intl. Conf. on Supercomputing, in: ICS'89, 1989.
[16] Xilinx zynq UltraScale+ MPSoC ZCU102 evaluation kit, 2021, Accessed: 2021-2-9, https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html.
[17] Energy efficient adaptive computing with multi-grain heterogeneous architectures (ENEAC) code repository, 2021, Accessed: 2021-11-18, https://github.com/kranik/ENEAC.
[18] S. Barrachina, M. Barreda, S. Catalán, M.F. Dolz, G. Fabregat, R. Mayo, E. Quintana-Ortí, An integrated framework for power-performance analysis of parallel scientific workloads, Energy (2013) 114–119.
[19] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, M.R. Stan, Hotspot: A compact thermal modeling methodology for early-stage VLSI design, IEEE Trans. Very Large Scale Integr. Syst. 14 (5) (2006).
[20] Y. Sun, X. Gong, A.K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, D. Kaeli, Hetero-mark, a benchmark suite for CPU-GPU collaborative computing, in: Intl. Symp. on Workload Characterization (IISWC), 2016, pp. 1–10.
[21] ARM, ARM architecture reference manual ARMv8, for ARMv8-a architecture profile (section A2.3), 2021, Accessed: 2021-5-9, https://developer.arm.com/documentation/ddi0487/latest/.
[22] N. Bell, M. Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors, in: Conference on High Performance Computing Networking, Storage and Analysis, in: SC '09, 2009.
[23] M. Hosseinabady, J.L. Nunez-Yanez, A streaming dataflow engine for sparse matrix-vector multiplication using high-level synthesis, IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 39 (6) (2020) 1272–1285.
[24] ARM performance libraries, 2021, https://www.arm.com/products/development-tools/server-and-hpc/allinea-studio/performance-libraries.

Andrés Rodríguez obtained a Ph.D. in Computer Science Engineering from the Universidad de Málaga, Spain, in 2000. From 1996 to 2002, he was an Assistant Professor in the Computer Architecture Department at Universidad de Málaga, being an Associate Professor since 2003. He lectures on operating system design, mobile devices architectures and IoT. His research interests are in parallel programming models, tools for heterogeneous architectures and edge computing.

Angeles Navarro obtained a Ph.D. in Computer Science from the Universidad de Málaga, Spain, in 2000. She is a Full Professor in the Department of Computer Architecture at Universidad de Málaga. She has been a Research Visiting Scholar in the University of Illinois at Urbana-Champaign (UIUC), the Technical University of Munich (TUM), the EPCC at the University of Edinburgh, the University of Bristol, and a Research Visitor in IBM T.J. Watson Research Center at New York and in Cray Inc at Seattle. She has served as a program committee member for several High Performance Computing related conferences as PPoPP, SC, ICS, PACT, IPDPS, ICPP, EuroPar, ISPA and ISC. She is the co-lider of the Parallel Programming Models and Compilers group at the Universidad de Málaga. Her research interests are in programming models for heterogeneous systems, analytical modeling, compiler and runtime optimizations.

Kris Nikov is a Senior Research Associate in SCEEM (School of Computer Science, Electrical and Electronic Engineering, and Engineering Maths) at the University of Bristol and a member of the advanced microelectronics group. They received a Ph.D. in Electrical and Electronic Engineering from the University of Bristol in 2018 and currently lecture on digital design, simulation and implementation. Their research interests include energy modeling and management as well as software and hardware performance optimization with a focus on reconfigurable and embedded heterogeneous systems.

Jose Nunez-Yanez is a Reader (associate professor) in adaptive and energy efficient computing at the University of Bristol and member of the microelectronics group. He holds a Ph.D. in hardware-based parallel data compression from the University of Loughborough. His main area of expertise is in the design of reconfigurable architectures for signal processing with a focus on run-time adaptation, parallelism and energy-efficiency. In 2006–2007 he was a Marie Curie research fellow at ST and in 2011 he was a Royal Society research fellow at ARM Ltd, Cambridge. He has published over 130 journal and conference publications and perform consultancy services in the area of lossless data compression and FPGA technology training. He is currently an industrial research fellow with the Royal Society working on MINET project ‘‘Machine Intelligence at the network edge’’ at Sensata technologies.

Rubén Gran Tejero graduated in Computer Science from the University of Zaragoza, Spain. He received his Ph.D. from the Polytechnic University of Catalonia (UPC), Spain, in 2010. He is currently Associate Professor in the Department of Computer Science and Systems Engineering at the University of Zaragoza. Previously, from 2010 to 2021 he stayed as Assistant Professor in the same institution. He has been Visiting Research Associate in the University of Illinois at Urbana-Campaign (UIUC) in 2011 and 2013. He has been part of the Program Committee of several conferences and workshops in the area: IPDPS, HPCS and PMBS. His research interests include hard real-time systems, hardware for reducing worst-case execution time and energy consumption, efficient processor microarchitecture, and effective programming for parallel and heterogeneous systems.

Darío Suárez Gracia received his Ph.D. degree in Computer Engineering from the University of Zaragoza, Spain, in 2011. From 2012 to 2015, he was at Qualcomm Research Silicon Valley. Currently, he is an Associate Professor at the University of Zaragoza. He has been a visiting scholar at the University of Toronto, the UIUC, and FORTH-Hellas. His research interests include parallel programming, heterogeneous computing, memory hierarchy design, energy-efficient multiprocessors, and fault-tolerance. Dr. Suárez Gracia is a member of the Aragon Institute of Engineering Research (I3A),the Spanish Society of Computer Architecture (SARTECO) the IEEE, the IEEE Computer Society, the ACM, and the HiPEAC European NoE.

Rafael Asenjo is Professor of Computer Architecture at the University of Málaga. He obtained a Ph.D. in Telecommunication Engineering in 1997. He has been using TBB since 2008 and over the last five years, he has focused on productively exploiting heterogeneous chips leveraging TBB as the orchestrating framework. In 2013 and 2014 he visited UIUC to work on CPU+GPU chips. In 2015 and 2016 he also started to research into CPU+FPGA chips while visiting the University of Bristol. He served as General Chair for ACM PPoPP'16 and as an Organization Committee member as well as a Program Committee member for several HPC related conferences (PPoPP, SC, PACT, IPDPS, HPCA, EuroPar, and SBAC-PAD). His research interests include heterogeneous programming models and architectures, parallelization of irregular codes and energy consumption. He co-authored the latest book (open access) on Threading Building Blocks (Pro TBB), is oneAPI Innovator, SYCL Advisory Panel member and ACM member.