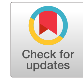
Future Generation Computer Systems 125 (2021) 10–23

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Efficient heterogeneous matrix profile on a CPU + High Performance FPGA with integrated HBM



Jose Carlos Romero [a], Angeles Navarro [a], Antonio Vilches [b], Andrés Rodríguez [a], Francisco Corbera [a], Rafael Asenjo [a,*]

[a] *Universidad de Málaga, Spain*
[b] *Shapelets, Puerta del Mar 18, 2nd Floor, 29005, Málaga, Spain*

ARTICLE INFO

Article history:
Received 17 February 2021
Received in revised form 6 May 2021
Accepted 11 June 2021
Available online 18 June 2021

Keywords:
High Performance FPGA
High Bandwidth Memory
Heterogeneous scheduler
Lightweight partitioner
Analytical model
Time series
Matrix profile

ABSTRACT

In this work, we study the problem of efficiently executing a state-of-the-art time series algorithm class – *SCAMP* – on a heterogeneous platform comprised of CPU + High Performance FPGA with integrated HBM (High Bandwidth Memory). The geometry of the algorithm (a triangular matrix walk) and the FPGA capabilities pose two challenges. First, several replicated IPs can be instantiated in the FPGA fabric, so load balance is an issue not only at system-level (CPU+FPGA), but also at device-level (FPGA IPs). And second, the data that each one of these IPs accesses must be carefully placed among the HBM banks in order to efficiently exploit the memory bandwidth offered by the banks while optimizing power consumption.

To tackle the first challenge we propose a novel hierarchical scheduler named *Fastfit*, to efficiently balance the workload in the heterogeneous system while ensuring near-optimal throughput. Our scheduler consists of a two level scheduling engine: (1) the system-level scheduler, which leverages an analytical model of the FPGA pipeline IPs, to find the near-optimal FPGA chunk size that guarantees optimal FPGA throughput; and (2) a geometry-aware device-level scheduler, which is responsible for the effective partitioning of the FPGA chunk into sub-chunks assigned to each FPGA IP. To deal with the second challenge we propose a methodology based on a model of the HBM bandwidth usage that allows us to set the minimum number of active banks that ensure the maximum aggregated memory bandwidth for a given number of IPs. Through exhaustive evaluation we validate the accuracy of our models, the efficiency of our intra-device partition strategies and the performance and energy efficiency of our *Fastfit* heterogeneous scheduler, finding that it outperforms state-of-the-art previous schedulers by achieving up to 99.4% of ideal performance.

© 2021 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

Time series analysis is becoming a major tool on many different domains, such as cloud computing monitoring [1], climate forecasting [2] or earthquake detection [3], among others. A very valuable outcome of these kind of analysis is the discovery of motifs (similarities) or discords (anomalies). Recently, the matrix profile computation has been proposed to accurately and efficiently find motifs and discords [4]. This algorithm consists in cross comparing all subsequences of the time series and recording a score in a resulting time series named "matrix profile". A simple inspection of the maximum and minimum values of the matrix profile is enough to identify the discords and motifs, respectively. To compute the matrix profile, different classes have been incrementally proposed: *STAMP* [4], *STOMP* [5], *SCRIMP* [6], and the latest and most efficient one *SCAMP* [7]. *SCRIMP* has been implemented for different parallel architectures: (1) distributed-memory computers [8]; (2) Intel Xeon Phi KNL processors that integrate 3D-stacked high-bandwidth memory (HBM) [9,10]; or (3) Heterogeneous CPU + GPU architectures [11]. However, the state-of-the-art algorithm to efficiently compute the matrix profile, *SCAMP*, has been only implemented on a Multi-GPU Cluster [7].

In this work, we propose an efficient implementation of *SCAMP* on a heterogeneous platform featuring a multicore CPU and a High Performance FPGA with integrated High Bandwidth Memory, HBM. This implementation poses some interesting challenges apart from tuning the SCAMP algorithm to efficiently run on the FPGA. For instance, several Matrix Profile kernels can be deployed on the FPGA as replicated IPs (FPGA compute units). In order to feed the CPU cores and the FPGA IPs with the corresponding

* Corresponding author.
E-mail addresses: jromero@ac.uma.es (J.C. Romero), angeles@ac.uma.es (A. Navarro), antonio.vilches@shapelets.io (A. Vilches), andres@ac.uma.es (A. Rodríguez), corbera@ac.uma.es (F. Corbera), asenjo@uma.es (R. Asenjo).

https://doi.org/10.1016/j.future.2021.06.025
0167-739X/© 2021 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

chunks of parallel iterations that guarantee optimal throughput while ensuring load balance, a hierarchical scheduler is proposed. It first partitions the work between the CPU cores and FPGA, and then it proceeds to partition the FPGA work among the different IPs. The system-level (inter-device) scheduler, called *Fastfit*, has been devised to quickly identify the granularity of the work that has to be offloaded to the FPGA in order to achieve both high FPGA utilization and CPU+FPGA load balance. The device-level (intra-device) scheduler is aware of the geometry of the *SCAMP* algorithm (a triangular matrix walk) to also distribute the work evenly among the FPGA IPs. Since the testbed FPGA features 32 HBM banks, we also contribute with a methodology to set the minimum number of active banks that ensure the maximum aggregated memory bandwidth while reducing power consumption. This methodology is based on a model of the HBM bandwidth usage and sharing of banks among IPs. We experimentally validate our scheduler in terms of performance and energy consumption and compare it with previous related and state-of-the-art heterogeneous schedulers.

Summarizing, the main contributions of this paper are:

- We present, to the best of our knowledge, the first FPGA implementation of a Matrix Profile algorithm using High Level Synthesis, HLS. We tune the *SCAMP* algorithm class for FPGA execution, which is the state-of-art algorithm for efficient time series analysis.
- We contribute with an efficient heterogeneous CPU + FPGA implementation that reduces the execution time and energy consumption with respect to the only-CPU approach.
- We propose *Fastfit*, a hierarchical scheduler: (1) at the outer/inter-device/system level, it efficiently balances workload among the FPGA and the CPU cores using a strategy that calculates a near optimal partitioning of the work for each device. For it, our scheduler uses an analytical model that assumes that an FPGA IP is internally implemented as a pipeline from which it estimates the near-optimal FPGA chunk size that maximizes the device throughput; and (2) at inner/intra-device/device level, it computes an even partition of the diagonals of the Matrix Profile so that all FPGA IPs complete their assignment at the same time.
- We develop a methodology based on a model to optimize the memory bandwidth usage of HBM Banks in a High Performance FPGA with integrated High Bandwidth Memory. Our model allows to easily find the minimum number of active HBM banks that reduce power consumption while ensure maximum aggregated bandwidth.

The rest of the paper is organized as follows. Section 2 briefly reviews the related work in the field of time series analysis and scheduling strategies for heterogeneous architectures. Next, in Section 3, we describe the *SCAMP* algorithm and the optimizations that we propose for our heterogeneous CPU+FPGA platform. Next two sections describe the *Fastfit* heterogeneous scheduler and HBM analytical model. The experimental results are presented in Section 6. The paper wraps up with conclusions and future work (Section 7).

2. Related work

2.1. Time series and matrix profile algorithms

Time series analysis covers many fields, such as cloud computing [1,12], forecasting [2], geology [3], or economics [13]. In particular, the discovery of similarities (motifs) or critical points (discords) in a time series is relevant for several of the previous problems. Motifs and discords can be found via probabilistic approaches [14]. However in this research we focus on the matrix profile [4] alternative because it provides an exact solution that cannot be obtained by probabilistic approaches.

Innovative implementations have been proposed for the matrix profile computation since its first appearance as the *STAMP* [4] algorithm, such as *STOMP* [5], *SCRIMP* [6] and *SCAMP* [7]. *STAMP* is supported by an FFT computation of dot products to compute the matrix profile in $O(n^2 log(n))$ complexity. *STOMP* reduces the complexity to $O(n^2)$, however the matrix profile has to be computed sequentially in rows. *SCRIMP* further optimizes the algorithm by exploiting the parallel computation of the diagonals of the matrix. Both *STOMP* and *SCRIMP* have been implemented for multicore and GPUs. *SCRIMP* also have a distributed-memory implementation aimed at multidimensional time-series [15], and some optimizations proposed for the execution on Intel Xeon Phi KNL processors that integrate 3D-stacked high-bandwidth memory (HBM) [10]. The state-of-the-art algorithm to compute the matrix profile, *SCAMP* [7] takes advantage of the Pearson correlation to compare subsequences, instead of euclidean distance. The use of the Pearson correlation improves both performance in the computation and accuracy in the results. Consequently, we choose *SCAMP* as the baseline for our heterogeneous CPU+FPGA implementation.

2.2. Heterogeneous computation of time series

Heterogeneous implementations of time series algorithms are scarce. The research by [16] leverages a heterogeneous platform without taking into account motifs and discords discovery, or scheduling strategies. Recently, our group has proposed the first heterogeneous CPU+GPU implementation of *SCRIMP* with adaptive scheduling [11]. One of the novelties of this paper is that we first update *SCRIMP* to *SCAMP* and also change the heterogeneous platform now comprised of CPU + High performance FPGA with integrated HBM.

FPGA architectures are gaining momentum in HPC and Data Centers as an energy-efficient alternative to CPUs and GPUs for different kind of problems like chaotic time series predictions [17], image processing [18], and cloud servers [19]. However, there is no implementation of any matrix profile algorithm tailored for FPGAs.

2.3. Heterogeneous scheduling using CPU + accelerator

Developing heterogeneous applications that makes the most out of CPU+Accelerator platforms is difficult and error prone due to low-level considerations: data sharing, synchronization, load balancing, scheduling, etc. To make it more approachable, new programming models and frameworks such as OmpSs [20], oneAPI [21] or SYCL [22] are being proposed. However, for the parallel_for paradigm, which is of interest to our application, they do 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. To address these issues, an heterogeneous parallel_for template was proposed in [23]. That template is based on the Threading Building Blocks library, TBB [24], effectively easing the development of heterogeneous applications. In that work, an adaptive scheduler named *Logfit* was also presented.

Logfit partitions the iteration space in chunks with a variable number of iterations. These chunks are processed on the CPU or accelerator on demand. Optimal chunk sizes that maximize the device throughputs while ensuring load balance are recomputed and adapted at the end of each chunk computation. *Logfit* is a robust adaptive scheduler, particularly suited for

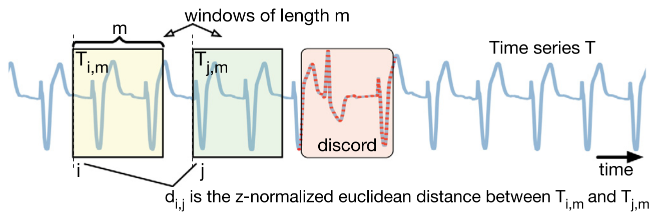


Fig. 1. Electrocardiogram time series T and two subsequences from which we can compute the distance $d_{i,j}$. The ventricular arrhythmia highlighted in the red box is a discord.

irregular codes. The advantages of *Logfit* over simpler heterogeneous schedulers for CPU+FPGA platforms are also investigated in [25,26]. However, the FPGA architecture is better exploited by regular algorithms as those devised for Time Series analysis, and in these cases, an adaptive scheduler introduces unnecessary overhead. In this work, we propose the *Fastfit* scheduler, designed for regular codes and tailored to make the most out of the FPGA features by minimizing the scheduling overheads. One important feature of the testbed platform is the FPGA support for High Bandwidth Memory (HBM) that is paramount for memory bound applications, like most of the Time Series algorithms. To the best of our knowledge, there are no published proposals to model and optimize the execution of Time Series applications on HBM-enabled FPGAs, which is another of the original contributions of this paper.

3. SCAMP description and optimizations

3.1. Time series and matrix profile

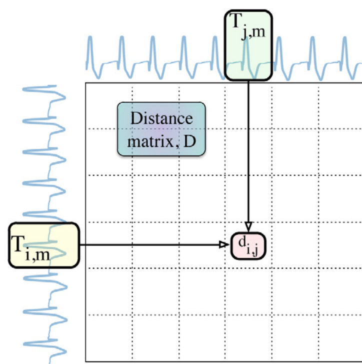
A time series is a sequential collection of data taken in time, as the electrocardiogram one depicted in Fig. 1. Following the related work's notation [4], a time series T is a sequence of real-valued numbers t_i : $T = t_1, t_2, \dots, t_n$, where n is the length of T. A subsequence $T_{i,m}$ is the local region (window) of T with consecutive values starting at position i and of length of m elements, i.e. $T_{i,m} = t_i, t_{i+1}, \dots, t_{i+m-1}$.

Our interest is to extract information from T by comparing each subsequence $T_{i,m}$ with all other subsequences $T_{j,m}$ of the same time series (self-join) or a different one (AB-join) with the indexes $1 \le i, j, \le n - m + 1$. For this purpose, we compute the distance profile as the vector $D_i = [d_{i,1}, d_{i,2}, \dots, d_{i,n-m+1}]$, where $d_{i,j}$ is the z-normalized Euclidean distance between $T_{i,m}$ and $T_{j,m}$. This distance can be efficiently computed using the Pearson correlation as we show in Eqs. (23)–(29) in the Appendix.

Note that due to the recursive computation of the covariance, $d_{i,j}$ depends on $d_{i-1,j-1}$.

Fig. 2 represents the distance matrix $D = [D_i]$ $(1 \le i \le n - m + 1)$, which contains all pairwise distances between all subsequences of T. The matrix D is symmetric ($d_{i,j} = d_{j,i}$), the values in the diagonal are zero ($d_{i,i} = 0$) and values near the diagonal are close to zero ($d_{i,i\pm k} \sim 0$ for small values of k) since neighbor subsequences are quite similar. To avoid these trivial matches an Exclusion Zone, *EZ*, surrounding the diagonal of D is enforced. We exploit the symmetry of D by computing only the upper triangular area of the matrix, excluding the main diagonal and the diagonals in the Exclusion Zone, *EZ*. As we will see next, the computation of this upper triangular region of D traverses the diagonals in order to preserve the $d_{i,j} \leftarrow d_{i-1,j-1}$ dependence.

The vector with the distances between each subsequence $T_{i,m}$ and its closest match (nearest neighbor), $MP = [min(D_1), min(D_2), \dots, min(D_{n-m+1})]$, is known as the matrix profile. Intuitively, it contains the minimum value of each column of the distance



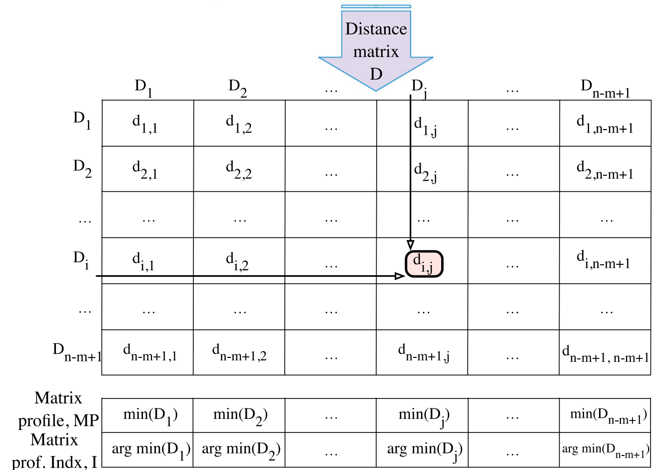


Fig. 2. Distance matrix, *D*, matrix profile, *MP* and matrix profile index, *I*.

matrix, hence the name "matrix profile". Motifs are the smallest values in *MP* whereas the discords are the largest ones. A matrix profile index, I is used to record where these motifs/discords are in T. Formally, $I = [I_1, I_2, \dots, I_{n-m+1}]$ where $I_i = j$ if $d_{i,j} = min(D_i)$, so it is computed as $I_i = arg\ min(D_i)$.

3.2. SCAMP algorithm and FPGA-oriented optimizations

Algorithm 1 shows the *SCAMP* sequential implementation of the matrix profile computation. In lines 1–3 we initialize n (length of T), the matrix profile, *MP*, the matrix profile index, I and the set of *Diagonals* that must be traversed. In line 4, df_i, dg_i, $norm_i$ and $Cov_{1,j}$, for $EZ < i < n$ are pre-computed as described in the Appendix and stored in their corresponding vectors to avoid repeating unnecessary computations. The outer loop (line 5) traverses the diagonals whereas the inner loop (line 6) processes each element of the diagonal in $O(1)$. The values $Cov_{i,j}$, $P_{i,j}$ and $d_{i,j}$ are not stored, but computed as C, P and d and later discarded after they are used to update *MP* and I in lines 12–13. These two checks of lines 12–13 are needed because although we traverse only the upper-triangular submatrix (because $d_{i,j} == d_{j,i}$), $MP_i = min(D_i)$ and I_i can be different to $MP_j = min(D_j)$ and I_j respectively.

For the FPGA and heterogeneous implementation, some relevant optimizations were applied to this algorithm:

- Instead of comparing with $d_{i,j}$ looking for the $min(D_i)$, we can just look for the largest $P_{i,j}$. That way we save some floating point operations and a square root. Now, the *MP* vector temporary stores Pearson correlations and at the end of computation motifs can be identified by the largest values, and discords by the smallest ones. If we really need

Algorithm 1: The SCAMP algorithm

```
   Input: A time series T, m and EZ
   Output: Matrix Profile MP, Matrix Profile Index I
1  n ← Length(T)
2  MP ← ∞, I ← zeros
3  Diagonals ← (EZ + 1 : n − m + 1)
4  df_i, dg_i, norm_i, Cov_{1,j} ← preCompute(T, m, EZ)
5  for k in Diagonals do
6  |  for i ← 1 to length(k) do
7  |  |  j ← i + k − 1
8  |  |  if i == 1 then C ← Cov_{1,k}
9  |  |  else C ← C + df_i · dg_j + df_j · dg_i
10 |  |  P ← C · norm_i · norm_j
11 |  |  d ← sqrt(2 · m · (1 − P))
12 |  |  if d < MP_i then MP_i ← d, I_i ← j
13 |  |  if d < MP_j then MP_j ← d, I_j ← i
14 |  end
15 end
16 return MP, I
```

the distance values, a quick traversal of MP applying Eq. (29) produces the Matrix Profile as we have defined in the previous section.

- The FPGA architecture excels at regular computations with the simplest control flow. We can remove the conditional expression (line 8 in Algorithm 1) using loop peeling, this is, unwinding the first iteration from the loop. The host (CPU) can take care of this first iteration and correspondingly update MP and I. This first iteration consumes less than 0.001% of the total execution time in our experiments. As a result, we save FPGA resources which translates into more SCAMP kernels (IPs) fitting into the FPGA fabric.
- Our heterogeneous implementation of *SCAMP* for CPU+FPGA platforms splits the *Diagonals* parallel iteration space in chunks of diagonals. Each thread (one per CPU core) and each FPGA IP (or FPGA kernel) can process different chunks in parallel. Each non-overlapping chunk of diagonals is identified by the range $[begin, end)$. Each CPU thread has a private copy of MP and I using TBB's combinable class that provides thread-local storage and a user-friendly reduction method. Each FPGA IP also has a private copy of MP and I, now using the High Bandwidth Memory banks available in our FPGA device. The required reduction phase that results in the final MP and I, consumes less than 0.09% of the total execution time, according to our experiments. The implementation details of the reduction operation are explained in [11], although in that paper we targeted a CPU+GPU platform and here we can have up to 40 FPGA IPs instead of a single GPU.

Algorithm 2 shows the pseudocode of the host code, that basically takes care of the initialization[1] (lines 1–4 in Algorithm 1) and then it precomputes the first iteration of the i-loop for all the diagonals. These computations are run sequentially but in our experiments the worst case consumes only 0.13% of the total execution time. The other 99.87% of the time is consumed in the `heterogeneous_parallel_for` call provided by our HBB library (Heterogeneous Building Blocks) [23] that we describe in Section 4.

Algorithm 2: SCAMP Host

```
1  n ← Length(T)
2  MP ← −∞, I ← zeros
3  Diagonals ← (EZ + 1 : n − m + 1)
4  df_i, dg_i, norm_i, Cov_{1,j} ← preCompute(T, m, EZ)
5  for k in Diagonals do
6  |  P ← Cov_{1,k} · norm_1 · norm_k
7  |  if P > MP_1 then MP_1 ← P, I_1 ← k
8  |  if P > MP_k then MP_k ← P, I_k ← 1
9  end
10 heterogeneous_parallel_for(Diagonals, Body)
11 return MP, I
```

As we describe later, this `heterogeneous_parallel_for` function requires a `Body` object that, among other things, encapsulates how to process a chunk of iterations (diagonals in this case) on the CPU and on the accelerator. Algorithm 3 shows the FPGA kernel implementation that takes cares of a chunk of diagonals in the range $[begin, end)$. Each FPGA kernel receives the initialized variables and writes in private MP and I arrays executing the i-loop starting at iteration $i = 2$. Note that we now compute and store the Pearson correlation instead of the distance.

Algorithm 3: SCAMP FPGA Kernel

```
   Input: MP, I, Cov, df, dg, norm, begin, end
   Output: MP and I
1  for k ← begin to end − 1 do
2  |  C ← Cov_{1,k}
3  |  for i ← 2 to length(k) do
4  |  |  j ← i + k − 1
5  |  |  C ← C + df_i · dg_j + df_j · dg_i
6  |  |  P ← C · norm_i · norm_j
7  |  |  if P > MP_i then MP_i ← P, I_i ← j
8  |  |  if P > MP_j then MP_j ← P, I_j ← i
9  |  end
10 end
11 return MP, I
```

The FPGA kernel is implemented in OpenCL and compiled into an FPGA bitstream using the Intel `aoc` compiler. As recommended in the FPGA optimization guide [27], we follow a single-task approach (also known as single work-item), in which the OpenCL kernel resembles a sequential C implementation. For these type of kernels the OpenCL NDRange[2] is set to (1, 1, 1), so a single thread is invoked on each FPGA IP. This results in loop pipelining and overlapping of data transfers and computations between loop iterations.

However, as we can see in Fig. 3, the inner i-loop that traverses a diagonal exhibits a loop carried dependence because iterations i and i' can RMW (read–modify–write) the same positions in MP and I. This dependence prevents the pipeline implementation of the loop and results in a highly inefficient FPGA execution.

However, a closer look at the loop body reveals that such a potential RMW conflict can be avoided in our implementation. Fig. 3 shows a simplification of the pipeline execution of the i-loop, where IL is the *Issue Latency* (a.k.a. Initiation Interval) or number of clock cycles between consecutive loop iterations. We also show CL, *Completion Latency* that we use in Section 4.2 to model the kernel throughput. The figure also shows the potentially conflicting statements D and E accessing the same MP

[1] Note that in contrast to Algorithm 1, now MP is initialized with $-\infty$ because it now stores Pearson correlations instead of distances.

[2] In the OpenCL standard, the NDRange represents the 3D space of parallel iterations.

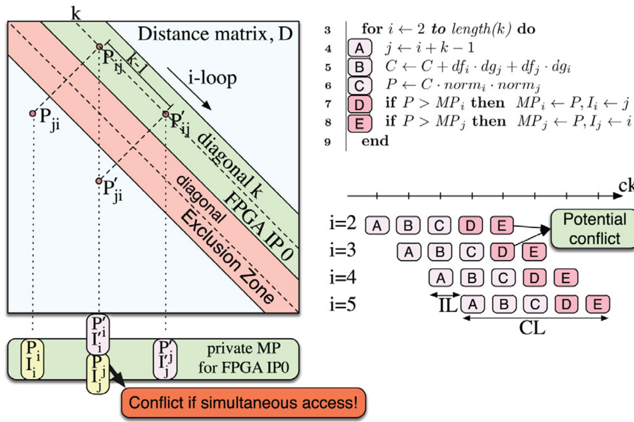


Fig. 3. Potential conflict due to loop-carried dependence in the inner loop and pipeline execution in the FPGA.

position. Since for a given diagonal k, index j walks through $j = i + k - 1$, two iterations i and $i' = i + k - 1$ can RMW the same position of MP: MP_j in E and $MP_{i'}$ in D. In the figure we simplify the situation assuming that $IL = 1$ and statements D and E require a clock cycle, but in general IL can be higher and the statements may require c cycles.

Therefore, if we can assure that statement E of iteration i finishes before statement D of i', then the loop can be safely pipelined. Without loss of generality, if statement E of iteration i access MP_j in the interval of cycles $[t, t + c)$, statement D of iteration i' access the same position in the interval $[t + IL \cdot (k - 1) - c, t + IL \cdot (k - 1))$, but these two intervals do not overlap if $t + c < t + IL \cdot (k - 1) - c$. In other words, if $c < (IL \cdot (k - 1))/2$.

In our algorithm, the first diagonal (the smallest k) is $k = EZ + 1$, and EZ is 256 as recommended in the literature [6,7]. On the other hand, the `aoc` compiler reports $IL \approx 6$, so the number of cycles, c, required to compute statements D and E should be smaller than 768 cycles. This is actually the case considering that each of these statements only includes a read operation from HBM, a comparison and two writes in HBM memory. If a smaller EZ is advised, we can always offload to the FPGA only the diagonals that are far enough from the main diagonal.

Therefore, as we know that there are not loop carried dependencies in the i-loop, we force the pipelining implementation with the `Pragma("ivdep")` directive just before the loop. Additionally, the FPGA kernels have been compiled with `-fp-relaxed -fpc`, that according to the FPGA OpenCL programming guide [28], result in floating-point optimizations including balanced tree hardware and elimination of intermediary rounding operations.

4. Fastfit: Hierarchical heterogeneous scheduler

4.1. Scheduling engine

Our scheduler is based on the Heterogeneous Building Blocks (**HBB**) library [23]. It is a C++ template library based on TBB, which takes advantage of heterogeneous processors and facilitates their usage and configuration. HBB aims to make easier the programming for heterogeneous processors by automatically partitioning and scheduling the workload among the CPU cores and OpenCL capable accelerators. HBB relies on OpenCL as the accelerator back-end for the sake of availability, portability, and programmability features, but the scheduling framework and policies of HBB could be easily adapted to other programming models or high level synthesis tools. Our library (HBB) offers an abstraction

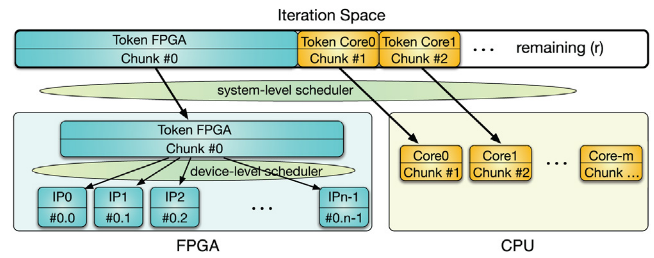


Fig. 4. System and device level schedulers used to partition the iteration space.

layer that hides the initialization and management details of TBB and OpenCL constructs (contexts, command queues, device_ids, etc.), thus the user can focus on his own application logic instead of dealing with thread management and synchronizations. The current version offers a `heterogeneous_parallel_for()` function template to run on heterogeneous CPU–GPU and CPU-FPGA systems.

Fig. 4 illustrates how the proposed hierarchical heterogeneous scheduler works. The system-level scheduler offloads chunks of iterations to the FPGA as soon as the FPGA becomes idle, and also assigns CPU chunks to each core in the CPU. The device-level scheduler takes care of partitioning each FPGA chunk into sub-chunks to appropriately feed each of the FPGA IPs. The iteration space includes: (i) chunks of iterations that have already been assigned to the FPGA (blue); (ii) chunks of iterations already assigned to the CPUs (orange); and (iii) remaining iterations (white).

The system-level scheduler is designed as a two-stage pipeline, Stage 1 and Stage 2, implemented on top of TBB. Thanks to the pipeline tokens we can easily control when the FPGA or the CPU cores are idle or busy. We initialize the pipeline object with one FPGA token and as many CPU tokens as the number of cores available. The tokens are circulating through the pipeline, being recycled at Stage 1 entry once they exit Stage 2. Depending on the arriving token, in Stage 1 the size of a CPU or FPGA chunk of iterations is computed (as we explain in Section 4.2), and the chunk is extracted from the set of remaining iterations, r. In the parallel Stage 2, either the CPU core or the FPGA processes the previously selected chunk.

We also initialize the TBB scheduler with as many worker threads as tokens (# of CPU cores plus 1 — the FPGA). That way, if we have one FPGA and two CPU cores, three worker threads are able to process three chunks of iterations in parallel. However, the FPGA can have several IPs (FPGA compute units) and therefore processing a chunk on the FPGA involves our device-level scheduler that evenly distributes the FPGA chunk among the available IPs (as described in Section 4.3). Note that the worker thread that processes the FPGA chunk/token, is the one: (i) running the device-level partitioning; (ii) offloading each sub-chunk to each IP; and (iii) blocking until all IPs have finished processing the sub-chunk. The oversubscription of the CPU cores is negligible because, although the TBB scheduler has one extra thread (one more than the number of cores), this thread is usually blocked while the FPGA is working. An alternative that we discarded consists in having one worker thread per FPGA IP and CPU core, but this results in too much oversubscription since in our platform we can have 40 IPs and 8 CPU cores, which translates into 48 worker threads. Besides, the FPGA OpenCL driver does not support concurrent offload requests from more than one worker thread.

4.2. Fastfit system-level scheduling algorithm

The system-level scheduler works at runtime and is designed as a two-phase strategy consisting of: the *Training Phase*, which finds the near-optimal chunksizes for the FPGA and the CPU that optimize the throughput in both devices while ensuring load balance; and the *Exploitation Phase*, which keeps this peak performance along the iteration space. Algorithm 4 depicts both phases.

A key component of the *Training Phase* is an analytical model that estimates the FPGA throughput, i.e., *elements per ms* computed when executing a chunk of parallel iterations. Our model assumes that an FPGA IP is internally implemented as a pipeline from which it estimates the near-optimal FPGA chunksize that maximizes the FPGA throughput. This model results in a good balance between accuracy and simplicity. The pipeline is characterized by two latencies: issue and completion latencies. The *Issue Latency*, *IL*, is the number of cycles required between issuing two consecutive independent iterations, which is also know as the *Initiation Interval*. On the other hand, the *Completion Latency*, *CL*, is the number of cycles until the result of a parallel iteration is available. Both latencies are in most cases sufficient to estimate the execution time of an FPGA kernel: the issue latency represents the time between dispatching two consecutive iterations of the kernel loop, while the completion latency depends on the depth of the pipeline and is the time required to fill it up.

Algorithm 4: Fastfit System-level scheduler

```
// Training Phase
Input: Frequency (F), δ, ρ
1 t_Cm(1), t_Fm(1) ← Eq. (2)
2 t_Fm(δ) ← Eq. (3)
3 CF ← Eq. (9) ← Eqs. (4) & (5)
4 CC ← Eq. (12) ← Eqs. (10) & (11)
5 return CF, CC
// Exploitation Phase
Input: CF, r, φ
6 CF = min(CF, r)
7 CC = CF/φ
8 return CF, CC
```

As we show in Algorithm 4, the *Training Phase* only requires to sample the CPU and FPGA throughput running one iteration (a diagonal in our application) on the CPU, and two chunks of iterations on the FPGA and then recording the corresponding execution times. In line 1, the first chunk for the FPGA and the CPU is made of 1 iteration each one. In line 2, the second chunk, only for the FPGA, contains a representative number of parallel iterations δ (in our study we find that 5% of the iteration space is enough to characterize the FPGA throughput for our application).

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

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

where DL represents the number of cycles required to traverse the pipeline after issuing a parallel iteration. The Completion Latency can be defined as $CL = IL{+}DL$. 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 $t_{F_m}(\delta)$ (lines 1 and 2 of Algorithm 4), we can solve 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 Eq. (1) and the previous expressions, we model the *FPGA estimated throughput*, λ_{F_e}, 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, in which the completion latency is hidden. Latency hiding is achieved by executing a large enough chunk of independent iterations. Ideally, when the DL is completely hidden ($DL/CF \rightarrow 0$), then the issue latency determines the run time and we attain peak performance. From Eq. (6) we compute the *peak performance* or *ideal 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 chunk 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 seek ρ values in the range $[0.9, 0.99]$, meaning that we aim to look for chunks that achieve throughputs that are within 90% and 99% of the peak performance. From Eqs. (6) and (7), and for a specified ρ, we know,

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

In other words, the near-optimal chunk size of parallel iterations that guarantee a throughput above a ρ threshold of the peak, CF_ρ, is computed as,

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

This steps can be summarized in Line 3 of Algorithm 4 where, using the execution times computed in Lines 1–2, Eqs. (4) and (5) can be solved. These solutions allow to solve Eq. (9) to get the near-optimal FPGA chunk size, 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 4 of Algorithm 4. As input we take the execution time of one parallel iteration in the CPU $t_{C_m}(1)$ – already computed – and the number of elements in the corresponding parallel iteration, N_C. This is, in fact, the number of elements in the corresponding single diagonal of the matrix. Also, we calculate N_F that represents the aggregated number of elements in all the diagonals of chunk CF_ρ. From them, we compute the throughput of the CPU and FPGA for both chunks as can be seen in Eqs. (10) and (11).

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

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

With this, 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 recommended CPU chunk size, CC, which can be approximately computed as:

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

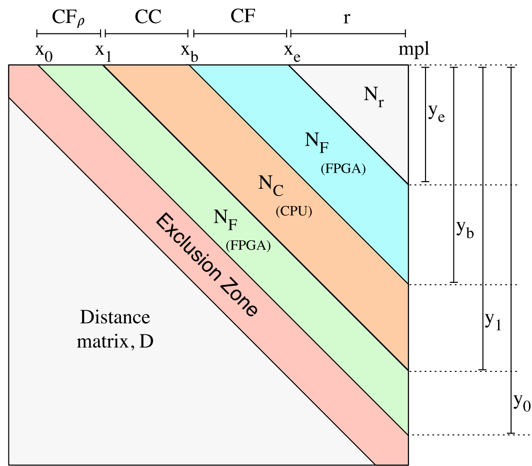


Fig. 5. Correction in the chunk size due to the triangular geometry of the problem.

After the first computation of chunk sizes CF and CC, the scheduler transitions to the *Exploitation Phase* (lines 6–8 in Algorithm 4), where we keep processing chunks of iterations on the CPU cores and FPGA and measuring the actual resulting throughput to update the relative speed among devices, φ. If the number of remaining iterations, r, is large enough, CF is kept as computed in the *Exploitation Phase* but when there are not enough iterations to feed the FPGA with CF iterations, then the remaining, r, are assigned as an FPGA chunk instead (see line 6). CC is recomputed each time to adapt to changes in φ (see line 7).

4.2.1. Fine tuning chunk size for SCAMP

In the previous section we have estimated a near-optimal FPGA chunk size, CF_ρ, that delivers an FPGA throughput close to the theoretical maximum. For any problem in which the geometry of the chunk is not an issue or the user does not have additional information about that, the previously computed chunk size can be a reasonable solution for the remainder of the application execution. However, we are aware that our problem exhibits a triangular geometry as can be seen in Fig. 5, and our goal is, once the optimal chunk size has been found in the *Training Phase*, to guarantee that each new chunk assigned to the FPGA always computes the same workload.

Let us suppose that the first FPGA chunk size found in the *Training Phase*, CF_ρ, traverses diagonals in the range $[x_0, x_1)$, accounting all of them to N_F elements. In the example of Fig. 5, the next chunk of iterations, CC, is computed on the CPU. Let us assume we transition to the *Exploitation Phase* and that a new FPGA chunk size, CF, has to be computed. Let us note that, in order to keep the desired FPGA throughput, we also have to keep almost constant the workload of all the FPGA chunks. So now, the problem is computing the next CF given that the first diagonal of the new chunk is x_b, so that the number of total elements in this chunk is also N_F.

The number of iterations/diagonals in CF is $CF = x_e - x_b$, for the new range $[x_b, x_e)$. Note that the index of the last diagonal is $mpl = n - m + 1$ (see Section 3). Fig. 5 shows that $x_i = mpl - y_i + 1$ where y_i is the number of elements in diagonal x_i. Since consecutive diagonals only differ in one element, in the chunk CF_ρ the aggregation of the first and last diagonal is $sum = y_0 + y_1 + 1$. This sum is equal to the aggregation of the adjacent interior diagonals, it is $sum = y_0 - 1 + y_1 + 2$, and so on. This results in,

$$N_F = \frac{CF_\rho}{2} \cdot (y_0 + y_1 + 1) \tag{13}$$

Now we want to compute CF and x_e, knowing N_F, x_b and that $CF = x_e - x_b = y_b - y_e$, using the same equation for the CF chunk:

$$N_F = \frac{CF}{2} \cdot (y_b + y_e + 1) = \frac{CF}{2} \cdot (2 \cdot y_b - CF + 1) \tag{14}$$

that is a quadratic equation from which we can easily solve CF and later x_e. That way, during the *Exploitation Phase*, we ensure that the number of elements computed on each FPGA chunk remain almost equal and that the FPGA yields an almost constant throughput as we validate in Section 6.

In our experiments, when activating this fine tuning of the chunk size in our scheduler we observe a 1% improvement in the performance w.r.t. the not geometrically aware scheduler (the default one).

4.3. Fastfit device-level scheduling algorithm

As introduced in Section 3.2, the FPGA can actually include N_{IP} compute units (or IPs) that work in parallel. The goal of the device-level scheduler is to partition each FPGA chunk of CF iterations among the N_{IP} IPs.

A naive distribution that disregard the geometry of our problem, would be the *Block* partition that just distributes the matrix diagonals in equal sub-chunks: $chunkIP = \frac{CF}{N_{\text{IP}}}$. This is the default policy in our scheduler. However, as we validate in Section 6, a better approach to perform the partition, which we call *Balanced*, do consider the number of elements in each diagonal.

Starting from Eq. (14), the N_F elements of the FPGA chunk CF have to be partitioned into N_{IP} sub-chunks, $CF_0, CF_1, \ldots CF_{N_{\text{IP}}-1}$, each one with approximately N_F/N_{IP} elements. Therefore we can compute each sub-chunk iteratively by following this expression,

$$\frac{N_F}{N_{IP}} = \frac{CF_i}{2} \cdot (2 \cdot y_{b_i} - CF_i + 1) \quad \forall i \in \{0 \ldots N_{\text{IP}} - 1\} \tag{15}$$

from which each CF_i can be computed starting with $i = 0$ and $y_{b_0} = y_b$, and updating at each step $y_{b_i} = y_{b_{i-1}} + CF_i$. With this *Balanced* partition strategy each IP gets approximately the same number of elements (± 1 diagonal). In our experiments this translates into a negligible unbalance among IPs (less than 10^{-5}%).

5. HBM exploitation

In case there are available resources on the FPGA fabric to instantiate several replicated FPGA IPs, we must tackle the issue of carefully placing and distributing among the HBM banks the data that each one of these IPs accesses, in order to efficiently exploit the memory bandwidth offered by the banks. Previous versions of the FPGA SDK for OpenCL compiler offered an optimization based on the generation of multiple compute units for enabling kernel replication through `pragma` and attribute `__attribute__((num_compute_units()))`. To emulate this feature, we replicate the kernel in the OpenCL source code N_{IP} times using C macros. This way the compiler implements each replicated kernel or FPGA IP as an unique pipeline.

One difference of the kernel replication with respect to the compute units compiler generation is that there is not a hardware scheduler unit built in the FPGA. That is the reason why we implement the device-level scheduler (see Section 4.3) responsible for the partition of the FPGA chunk and the dispatch of the corresponding sub-chunks among the different IPs. Another key difference of our kernel replication strategy is that it allows us to control the HBM bank where each IP will access local data. Trivially, one IP can access its local data from one HBM bank. Thus, by increasing the number of replicated IPs, each one accessing data from a different HMB bank, we can achieve higher throughput

when exploiting the aggregated memory bandwidth of the concurrent memory banks. However, increasing the number of active memory banks rises power consumption and requires additional FPGA resources to orchestrate all the bank's memory accesses. In case that one IP does not exhaust the available bank bandwidth, then two (or more) IPs could have allocated their data on one HBM bank. This bank sharing solution would optimize the memory bandwidth usage of each HBM bank, reduce the number of active HBMs, and decrease power consumption and FPGA resources, while obtaining the maximum aggregated bandwidth achievable for a given number of replicated IPs.

In this section we present a methodology that allows us to: (i) select the optimal number of IPs that can access each bank in order to ensure optimal memory bandwidth usage of a HBM bank; and also ii) set the minimum number of active banks that ensure the maximum aggregated memory bandwidth for a given number of IPs. This methodology is based on a model of the HBM bandwidth usage that we explain next. Fig. 6 illustrates the accuracy of our model and its applicability.

5.1. Modeling bandwidth usage for HBM

Let us start by modeling the memory bandwidth usage when one IP accesses data from one HBM bank and let us assume that the number of replicated IPs is given as N_{IP}. The *aocl* compiler reports the frequency, $F_{N_{\mathrm{IP}}}(j = 1)$ at which this implementation is synthesized, where j represents the number of IPs per memory bank (1 in this case). Being W the width (Bytes) of one HBM bank, then we can estimate the ideal bandwidth per IP and per bank as,

$$BW_{ideal}(j = 1) = F_{N_{\mathrm{IP}}}(j = 1) \cdot W \tag{16}$$

Using the FPGA Dynamic Profiler for OpenCL tool [27] we obtain $BW_m(j = 1)$, the measured memory bandwidth per IP and per HBM bank in our implementation. Now, we can define the memory bandwidth usage per IP and per HBM module, σ, as

$$\sigma = \frac{BW_m(j = 1)}{BW_{ideal}(j = 1)} \tag{17}$$

Through exhaustive experimentation with different implementations in which we keep N_{IP} fixed but increase the number of IPs that can access one HBM module ($j > 1$) while decreasing the number of active HMB modules, i ($i = N_{\mathrm{IP}}/j$), we find that factor σ represents a good estimation of the memory bus occupancy while the bus is not saturated, because the measured memory bandwidth per HBM bank increases linearly with the number of IPs per bank. Thus, we model or estimate the memory bandwidth per HBM as,

$$\begin{aligned} BW(j \geq 1) &= j \cdot BW_m(j = 1) \\ &= j \cdot \sigma \cdot BW_{ideal}(j = 1) \end{aligned} \tag{18}$$

In case of bus saturation, there is not headroom for one additional IP accessing data from a HBM module, in other words, the maximum achievable bandwidth per bank is,

$$BW_{max} = (1 - \sigma) \cdot BW_{ideal}(j = 1) \tag{19}$$

In summary, from Eqs. (18) and (19) we compute the aggregated memory bandwidth for i HMB modules as,

$$\begin{aligned} ABW(i, j \geq 1) &= j \cdot \sigma \cdot BW_{ideal}(j = 1) \cdot i \\ &\leq (1 - \sigma) \cdot BW_{ideal}(j = 1) \cdot i \end{aligned} \tag{20}$$

Fig. 6 represents a case of study when $N_{\mathrm{IP}} = 24$, where the IPs are distributed among different number of memory banks (i): from 24 HBM banks (i.e. $j = 1$) to 2 HBM banks ($j = 12$). The experimental setup is detailed in Section 6.1.

In Fig. 6(a), lines depict the aggregated memory bandwidth (MB/s) for different number of HBM modules. The dashed line is the aggregated memory bandwidth from Eq. (20), while the solid line is the actual aggregated memory bandwidth measured for each configuration. The values for $i = 24$ represent in fact $BW_{ideal}(j = 1)$ (Eq. (16)) and the measured $BW_{meas}(j = 1)$, which our model uses to compute σ (Eq. (17)). As we see, the model predicts accurately the behavior of the HBM system, being the deviation below 11%. The figure also shows the application throughput (elements/ms), and that the aggregated memory bandwidth is a good proxy of the performance behavior.

Fig. 6(b) depicts measured power (Watts) – lines – and energy (Joules) — bars. The main power components (due to the IPs and the UIB[3] bus consumption) demonstrate that decreasing the number of HBM banks reduce power consumption. Thus, it is advisable to deploy the minimum number of memory banks that guarantees optimal memory bandwidth usage. This is our definition of optimal number of memory banks. As shown in Fig. 6(a), maximum memory bandwidth is sustained from 24 to 6 banks. In fact, the energy consumption is the minimum for the same range of memory banks. Reducing the number of HBM modules below 6 increases the number of IPs per memory bank, which causes the saturation of each module memory bandwidth. As a consequence, both the aggregated memory bandwidth (thus, throughput) and energy consumption degrade.

With this model we compute the optimal number of memory banks. From Eqs. (18) and (19) we firstly find the optimal number of IPs per bank, j_{opt}, that is the maximum number of IPs that can access data from one HBM module without saturating the bus (ensuring this way optimal memory bandwidth usage),

$$j_{opt} = max(j, 1) \quad : \quad j \leq \left\lfloor \frac{1 - \sigma}{\sigma} \right\rfloor \tag{21}$$

Once we have found the optimal number of IPs per HBM bank, we calculate the optimal number of memory banks that ensure the maximum aggregated memory bandwidth as,

$$i_{opt} = \left\lceil \frac{N_{\mathrm{IP}}}{i_{opt}} \right\rceil \tag{22}$$

In our case of study we find that $\sigma = 0, 167$, so $j_{opt} = 4$ and $i_{opt} = 6$. From Fig. 6(a) we corroborate this finding.

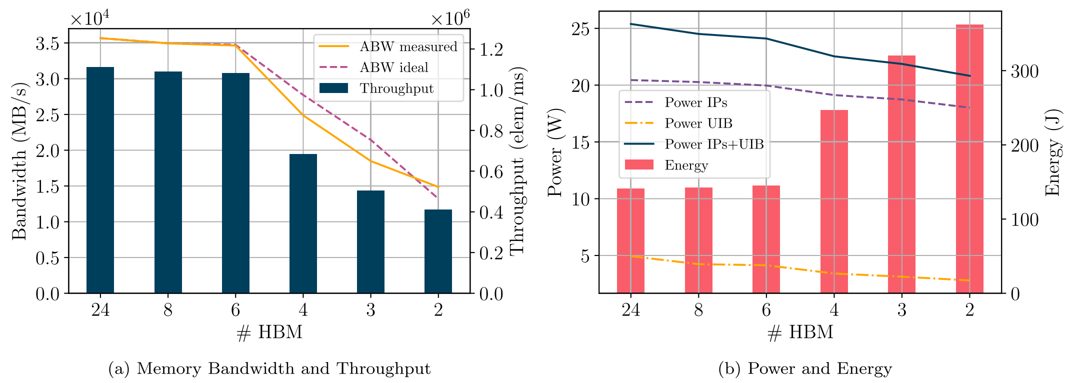
Please note that in the experimental section we have follow this methodology for selecting the optimal number of active HBM banks and optimal number of IPs per bank for any given number of IPs.

6. Experimental results

6.1. Setup

The experimental evaluation has been conducted on a CPU+FPGA platform. The CPU is an Intel Core i7-7820X, 3.60 GHz, 8 cores and 128 GB DDR4 RAM. The FPGA is an Intel Stratix 10 MX with 32 HBM memory banks, 512 MB per bank and 16GB total. The system runs CentOS 7.2.1511, OpenCL 1.0, FPGA SDK for OpenCL v.19.3, and GCC 4.8.5. All results (performance, energy, and profiling metrics) report the median value of 5 runs. The performance metric is throughput (elements per millisecond) and energy is reported in Joules. The normalized standard deviation for throughput (energy) measurements is always below 3% (4%). Unless otherwise stated, our heterogeneous runs simultaneously exploit 8 CPU cores and, as motivated in Section 6.2.1, 40 IP/FPGA compute units. Energy results were obtained using the Processor Counter Monitor (PCM) library for the CPU part, and the self-developed Stratix-Monitor library [29] for the FPGA device.

[3] Universal Interface Bus that powers the HBM DRAM.



(a) Memory Bandwidth and Throughput (b) Power and Energy

Fig. 6. Model estimation and performance evaluation for 24 IPs and different number of HBMs: (a) Comparing ideal and measured aggregated memory bandwidth vs application throughput — the higher the better; (b) Comparing Power dissipation vs energy consumption — the lower the better-.

We consider 4 time series of different sizes: 2^{17} (131 072), 2^{18} (262 144), 2^{19} (524 188) and 2^{20} (1 048 576). These time series are random-walk time series that are commonly used for benchmarking in time series analysis algorithms [30].

The *Fastfit* scheduler discussed in Section 4.2 is invoked with $\rho = 0.99$ and $\delta = 0.2\%$, and it is compared with three previous schedulers that were initially devised for CPU+GPU platforms [23]:

- *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/NumCores`.
- *Dynamic*: it lazily splits the iteration space dynamically. Each time the FPGA is idle, it takes a chunk from the iteration space. The size of this chunk is user-provided using the `chunkFPGA` input argument. The CPU cores also take chunks of the iteration space, but now `chunkCore=chunkFPGA/`φ, where φ is the relative speed of the FPGA w.r.t. the CPU core (i.e. if the FPGA is 2x faster than a CPU core, $\varphi = 2$, so `chunkCore` is half the size of `chunkFPGA`). A guided self-scheduling [31] is used when there are not enough remaining iterations to enforce the previous equations.
- *Logfit*: it also dynamically splits the iteration space, but the user does not provide a constant `chunkFPGA` size. On the contrary, this `chunkFPGA` 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 systems [23].

6.2. FPGA-only evaluation

In this subsection, we evaluate the performance of our schedulers when the FPGA is the only device computing the matrix profile. Heterogeneous executions are considered in the next subsection.

6.2.1. Kernel replication exploration

One of the FPGA optimizations described in 3.2 was kernel replication [27]. This optimization results in a better utilization of the FPGA resources, leading to performance gains if there is enough bandwidth to feed all the replicated kernels or IPs. We explore the performance for different number of replicated IPs applying the methodology given in Section 5 for selecting the optimal number of active HBM banks and optimal number of IPs per

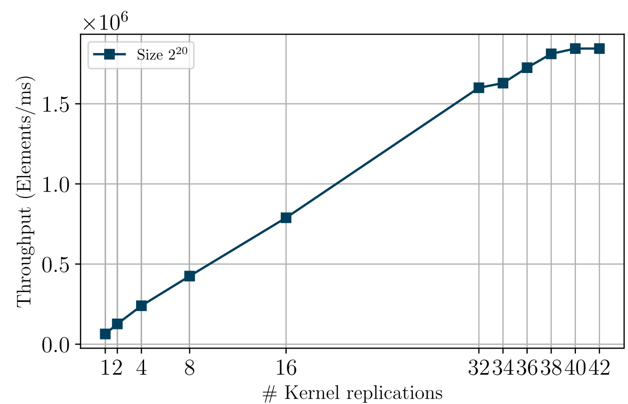


Fig. 7. Exploring the number of kernel replications, N_{IP}, for the 2^{20} time-series and only-FPGA execution.

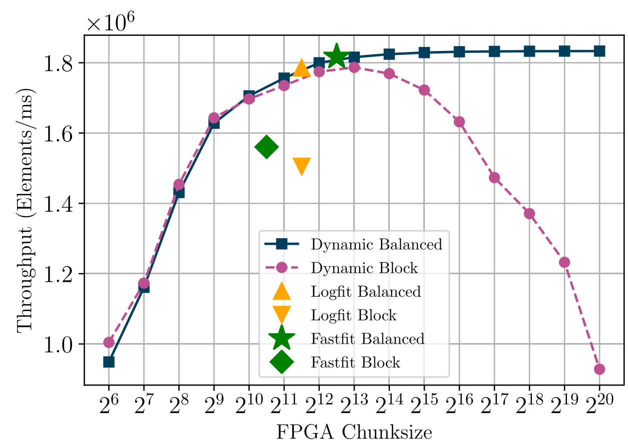


Fig. 8. Device-level scheduler exploration for different schedulers and a time series size of 2^{20}. X-axis represents the FPGA chunk size and the Y-axis the throughput. The higher the better.

bank for each case. The results for time series input 2^{20} are shown in Fig. 7. The maximum number of IPs that fit in our FPGA is 42, but as can be seen in Fig. 7 maximum performance is obtained for 40 IPs. Smaller time series exhibited the same behavior. Unless explicitly stated, from now on, we fix the number of FPGA IPs to $N_{\mathrm{IP}} = 40$.

6.2.2. Evaluation of partition strategies at device-level

As explained in Section 4.3, once the system-level scheduler assigns an FPGA chunk to the FPGA, the device-level scheduler

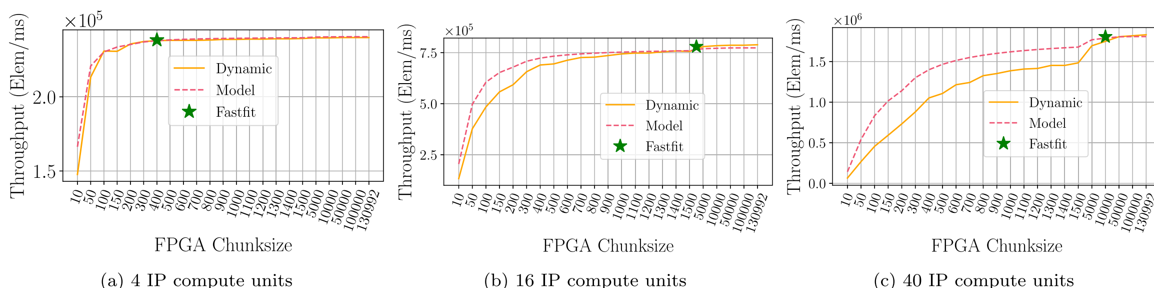


Fig. 9. Comparison of *Fastfit* model simulation, *Dynamic* and *Fastfit* executions for time series size 2^{17}. X-axis represents the FPGA chunk size and the Y-axis the throughput. The higher the better.

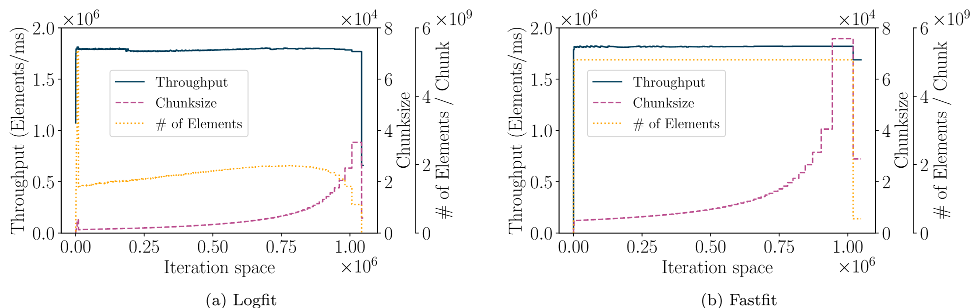


Fig. 10. *Logfit* and *Fastfit* evolution of the throughput, FPGA chunk size and total number of elements of the chunks for the 2^{20} time series. X-axis represents the iteration space.

has to partition this chunk among the different IPs. In this section, we quantitatively validate the throughput improvements due to the use of our proposed *Balanced* partition strategy with respect to a naive *Block* one that is not aware of the different diagonal lengths.

Fig. 8 shows the impact in the throughput for the different system-level schedulers: *Dynamic*, *Logfit* and *Fastfit*. Although in this experiment only the FPGA is used (there are no CPU cores collaborating in the computation), we assess these three system-level schedulers since they produce different FPGA chunk sizes as well as a significant number of chunks. In the figure, we plot the throughput of the FPGA for the *Dynamic* scheduler when configured with FPGA chunk sizes from 2^6 to 2^{20} (being 2^{20} the whole iteration space in our largest time series). Since *Logfit* produces variable FPGA chunk sizes during the computation, the average chunk size is shown in Fig. 8, while for the *Fastfit* scheduler we depict the FPGA chunk size found in the *Training Phase*. Note that since *Logfit* and *Fastfit* schedulers compute the FPGA chunk size depending on the FPGA throughput, using *Balanced* or *Block* partition strategy in the device-level scheduler, may have an impact on the FPGA chunk sizes found, as well as on performance.

Fig. 8 evidences that the workload unbalance of the *Block* partition can have a remarkable impact on the throughput, especially for large FPGA chunk sizes, as we see in the *Dynamic* scheduler for chunk sizes larger than 2^{14}. On the other hand, when using the *Block* strategy, *Logfit* and *Fastfit* tend to find smaller FPGA chunk sizes than using *Balanced*. This is because the FPGA throughput measured during the training (in both schedulers) is smaller due to load unbalance among the IPs which results in sub-optimal chunk size estimation.

In the three schedulers, using the *Balanced* partition strategy always achieves the best performance. In summary, *Balanced* results in 97.37% better throughput than *Block* for the highest chunk size in the *Dynamic* scheduler, and 18.66% and 16.45% improvements in throughput for *Logfit* and *Fastfit*, respectively. This result motivates us to keep using the *Balanced* partition strategy in the rest of the evaluation.

6.2.3. Validation of Fastfit model

In this subsection, we validate the *Fastfit* model described in Section 4.2. This model is devised to predict the FPGA throughput for any FPGA chunk size. For it, after obtaining F, IL and DL, we use Eq. (11) to compute the throughput for any chunk size CF. Fig. 9 shows the estimated throughput (`Model`) computed for $\rho = 0.99$ vs. the actual measured one (`Dynamic`) for different FPGA chunk sizes and different number of replicated IPs for the 2^{17} time series. In addition, in the figure we mark with a green star the throughput for the estimated near-optimal FGPA chunk size, CF_ρ, obtained after the Training Phase in a real execution of *Fastfit*. It is worth remarking that: (i) the model is accurate, especially for smaller number of IPs where the device-level scheduler overhead and potential load unbalance among IPs have a less noticeable impact on the real throughput; (ii) the execution of *Fastfit* end up using an FPGA chunk size that results in an almost optimal throughput. For instance, the throughput predicted by the model for the near-optimal chunk size is between 97%–99% of the actual measured throughput for the selected chunk size. Similar accuracy was achieved for different input sizes; (iii) the FPGA chunk sizes found leave room for CPU collaboration and CPU+FPGA heterogeneous co-execution; and iv) all in all, our model allows to obtain the desired throughput out of the FPGA without having to perform the manual exploration required by *Dynamic*.

Although it was initially devised for CPU+GPU platforms and irregular algorithms, *Logfit* [23] is a related scheduler that can also save the exploration time that was needed with *Dynamic*. Both *Logfit* and *Fastfit* avoid the manual exploration of a suitable chunk size using a two phases scheme: a *Training phase* and a *Exploitation phase*. However *Logfit* spends more time than *Fastfit* in the *Training Phase* and continuously recompute new FPGA chunk sizes during the *Exploitation phase*, which can suppose additional overhead. In particular, the *Training phase* of *Logfit* requires more time because it samples the throughput obtained with monotonically increasing chunk sizes until the throughput stop growing. Using four of the previous samples, it computes the

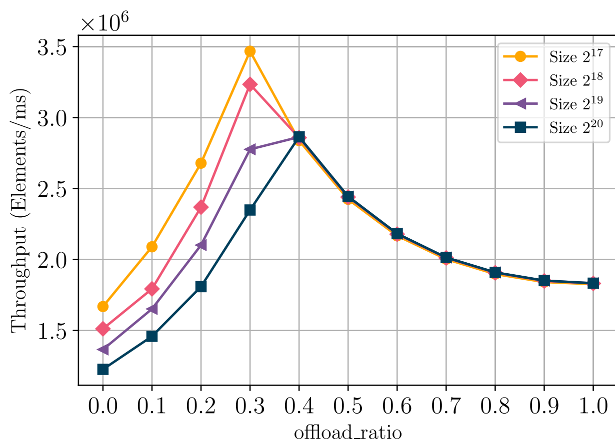


Fig. 11. Throughput for *Static* and different time series. *X*-axis represents the percentage of iterations (as a ratio) offloaded to the FPGA. The higher the better.

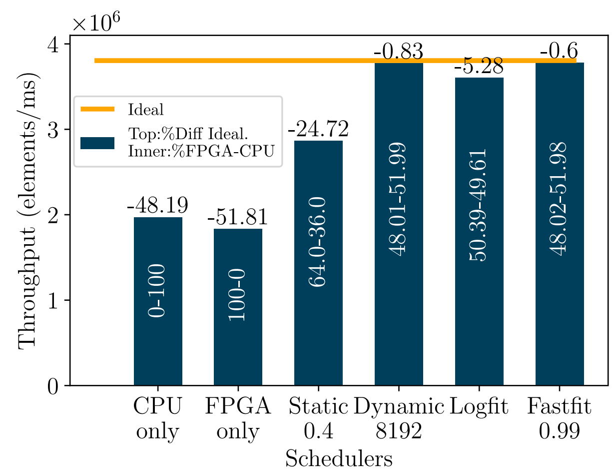


Fig. 12. Throughput comparison for all schedulers and 2^{20} input using the FPGA and 8 CPU cores. The numbers inside the bars indicate the percentage of Elements (not diagonals) computed on the FPGA (bottom) and CPU (top). The numbers above the bars are the percentage of performance degradation with respect to the ideal throughput (CPU only + FPGA only throughput) represented as an horizontal line.

logarithmic function that fit these samples and with this it can select the near-optimal accelerator chunk size (see [23] for more details). The pipeline architecture of FPGA IPs allows the *Fastfit* simple model to be accurate enough with only two throughput samples and no logarithmic fitting.

In order to better understand the different behavior between *Logfit* and *Fastfit*, Fig. 10 shows the evolution of the throughput (`Throughput`) and FPGA chunk size using two metrics: number of diagonals per chunk (`Chunksize` = *CF*) and number of total elements in the chunk (accumulating all the elements of all the diagonals in the chunk, `# of Elements/Chunk` = N_F). In Fig. 10(a) we can see a glitch, in the `Chunksize` and `# of Elements/Chunk` curves, at the beginning of the iteration space were several samples are needed until we can move on to the *Exploitation phase*. In Fig. 10(b), although it is hardly noticeable, we only test two different chunks sizes that let us to quickly move to the next phase.

The *Exploitation phase* of *Logfit* is also more complex and introduces more overhead since it keeps adapting the accelerator chunk size to suits to throughput changes that can be frequent in irregular codes. However, the *Exploitation phase* of *Fastfit* assumes the code is regular and that the chunk size estimated in the previous phase is valid for the whole iteration space. As we can see in Fig. 10(b), *Fastfit* sustains a more stable throughput and an almost constant `# of Elements/Chunk` that is directly proportional to the workload per chunk of iterations and higher than the workload per chunk assigned by *Logfit*. This explains the higher average throughput observed in *FasFit*. Note that in both schedulers, the chunk size increases to keep the `# of Elements` constant since the diagonals are becoming shorter as we sweep the iteration space. Also note that the throughput can take a performance hit at the end of the iteration space when there are not enough iterations to fully utilize the FPGA pipeline.

6.3. Evaluation of heterogeneous CPU+FPGA co-executions

In this section, we validate the four system-level heterogeneous schedulers, *Static*, *Dynamic*, *Logfit* and *Fastfit* when using both the CPU (8 cores) and the FPGA co-executing simultaneously. We first focus on the obtained performance and later on the energy efficiency.

6.3.1. Performance analysis

Remember that *Static* requires that the user provides an `offload_ratio` stating the percentage of iterations offloaded to the FPGA. Fig. 11 shows the throughput obtained for `offload_ratio` between 0 (only CPU execution) and 1 (only FPGA execution) and four time series with different sizes. After this manual exploration, we found that the smaller time series exhibit maximum throughput when 30% of the iteration space is computed on the FPGA, but for larger time series it is better to offload 40% of the iteration space to the FPGA and compute the rest on the CPU. Different time series may require different `offload_ratio` values and a more precise search might pay off (in steps of 1% instead of 10% as in the figure) although more time should be devoted to the exploration.

As we saw in Fig. 8, *Dynamic* also requires an offline profiling in order to find a suitable FPGA chunk size, that for the 2^{20} time series ends up being 2^{13} diagonals. Note that larger chunk sizes result in similar throughput but makes less likely to achieve CPU+FPGA work-sharing and load balance. *Logfit* and *Fastfit* automatically find, without user intervention, the suitable sizes for the FPGA and the CPU cores, at a small training overhead.

In order to compare the performance of all the evaluated schedulers, Fig. 12 shows a throughput comparison for the best results obtained with each scheduler and the 2^{20} time series. The number inside the bars indicates the percentage of Elements processed on each device (FPGA–CPU), so the CPU-only and FPGA-only executions show 0-100 and 100-0, respectively. Note that although in *Static* we set `offload_ratio` = 0.4 which distributes 40% of the iterations (diagonals) to the FPGA and 60% to the CPU, the FPGA ends up processing 64% of the Distance Matrix Elements since the first 40% of the diagonals are larger than the remaining 60%. The upper orange horizontal line indicates the Ideal throughput, estimated as the aggregation of the CPU-only throughput and the FPGA-only one. This ideal throughput does not account for the unbalance and scheduling overheads so it is an upper bound used to quantitatively estimate the impact of these overheads.

The best *Dynamic* execution is obtained for FPGA chunksize equal to 8192, `chunkFPGA` = 2^{13} as we saw in Fig. 8. With this manual configuration it only looses 0.83% of performance w.r.t. the ideal due to the partitioning overhead, although it requires the offline exploration to find the best `chunkFPGA` input argument. *Logfit* departs 5.28% of the ideal due to the scheduler overhead (training and logarithmic re-fitting) that were mentioned in Section 6.2.3. However *Fastfit* delivers almost ideal performance, automatically finding a very good initial FPGA chunksize

Table 1
Summary of performance, energy consumption and energy efficiency. Diff. represents the degradation with respect ideal or best. The optimal implementation for each criterion in boldface.

	CPU 8 cores	FPGA only	Static	Dynamic	Logfit	Fastfit
Throughput (Elements/ms)	1 971 300	1 833 240	2 864 100	3 772 960	3 603 670	**3 781 880**
%Diff. Ideal Throughput (CPU+GPU)	−48.19%	−51.81%	−24.72%	−0.83%	−5.28%	**−0.6%**
Energy consumption (Joules)	33 464.5	**11 853.5**	18 714.69	20 799.82	21 112.66	20 836.12
%Diff Best Energy consumption	182.32%	**0.0%**	57.88%	75.47%	78.11%	75.78%
Energy efficiency (Elements/J)	16 425 539.16	**46 372 164.78**	29 371 176.08	26 426 789.04	26 035 206.14	26 380 749.16
%Diff Best Energy efficiency	−64.58%	**0.0%**	−36.66%	−43.01%	−43.86%	−43.11%

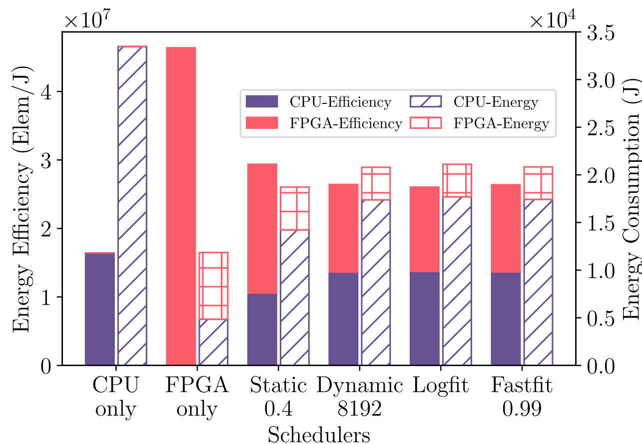


Fig. 13. Energy metrics for all schedulers and 2^{20} time series using the FPGA and 8 CPU cores. Solid bars represent the Energy efficiency (the higher the better), whereas patterned bars depict Energy consumption (the lower the better).

of 9,455 diagonals. Subsequent FPGA chunk sizes are updated just to maintain a constant workload (number of Elements) as explained in Section 4.2.1. This small difference with respect to *Dynamic* makes *Fastfit* delivers a slightly better throughput, highlighting an almost negligible overhead (0.6%). Similar results have been obtained for different input sizes. This finding, along with the advantage of avoiding the manual search of a near-optimal FPGA chunk size, turn *Fastfit* into an excellent scheduler for co-execution of regular algorithms on CPU+FPGA platforms.

6.3.2. Energy analysis

Fig. 13 depicts a breakdown of energy efficiency (in Elements per Joule) and energy consumption (in Joules) in the same conditions explained in the previous section. FPGA energy is measured thanks to an in-house library (publicly available [29]) built on top of the BMC (Board Management Controller) library provided by the FPGA vendor (BittWare). Energy efficiency has been computed dividing the number of computed elements by the total number of Joules consumed.

First, paying attention to the one-device only results at the left of Fig. 13, it can be observed that the FPGA exhibits the highest energy efficiency and the lowest energy consumption. The CPU requires almost 3x more energy to carry out the same computation. Now, the energy consumed by the heterogeneous schedulers is roughly proportional to the workload processed by each device (CPU and FPGA) as pointed out in Fig. 12. For example, *Static* offload more work to the FPGA (64% of the elements) and consequently exhibits better energy efficiency than *Dynamic*, *Logfit* and *Fastfit*. Actually, energy consumption and energy efficiency in these last three schedulers are similar. Due to *Logfit* being the slowest of the last three schedulers, it also consumes more energy than *Dynamic* and *Fastfit*.

Table 1 includes the relevant data already presented in previous charts. Summarizing, *Fastfit* is the best scheduler if our goal is to achieve maximum performance. Although *Dynamic* also achieves good results, let us recall that in this case the user needs to explore offline exhaustively all possible chunk sizes to find the near optimal, whereas in *Fastfit* the best chunk size is automatically discovered at runtime. On the other hand, if the target is energy consumption, it is better to switch off the computation on the CPU cores and resort to the FPGA-only execution. Again, similar conclusions can be obtained for different input sizes.

7. Conclusions

In this work we have proposed a novel hierarchical scheduler named *Fastfit*, to efficiently balance the workload in a heterogeneous system while ensuring near-optimal throughput, and we have used *SCAMP* – a state-of-the-art time series algorithm class – to illustrate its applicability. *Fastfit* is a system-level scheduler based on an analytical model of the FPGA pipeline IPs that helps us to find the FPGA chunk size that guarantees near-optimal FPGA throughput, and from that, the CPU chunk size that ensures load balance among devices. Besides, *Fastfit* includes a device-level scheduler that provides an effective partition of the FPGA chunk into sub-chunks for each FPGA IP.

Through exhaustive evaluation, we validate the accuracy of our models and the optimality of *Fastfit* for getting the near-optimal FPGA chunk size, finding that our model prediction is within the 97%–99% of the actual measured best throughput. We also compare different strategies for performing the device-level partition of the FPGA chunk among IPs, finding that the *Balanced* strategy that is aware of the triangular geometry of the problem improves the performance of a naive Block one by 16.45%. We also find that a simple model of the HBM usage bandwidth and the sharing of banks among IPs allow us to set the minimum number of active banks that ensure the maximum aggregated memory bandwidth while reducing power consumption.

We compare our proposed scheduler with previous scheduling strategies (*Static*, *Dynamic* and *Logfit*) and we demonstrate that our new scheduler improves all of them in terms of performance. Moreover, *Fastfit* is 4.68% better than *Logfit*, a previous state-of-the-art adaptive scheduler that finds the near-optimal chunk size for each device without the need of offline profiling. The reason of that improvement is that our new proposal avoids the logarithmic fitting overheads of *Logfit*. In fact, *Fastfit* is only 0.6% away from the ideal heterogeneous execution. However, if our goal is to minimize energy consumption, offloading all the workload to the FPGA is the best choice.

As future work, we plan to explore the use of fixed-point arithmetic to further optimize the FPGA implementation. Currently, our FPGA with HBM banks is not supported by the oneAPI framework that uses SYCL instead of OpenCL to implement the FPGA kernel. We will look into the necessary modifications to the FPGA BSP (Board Support Package) and alternatives to enable the use of SYCL while permitting the exploitation of the HBM memory banks.

CRediT authorship contribution statement

Jose Carlos Romero: Investigation, Software, Validation, Writing - original draft, Visualization, Data curation. **Angeles Navarro:** Conceptualization, Methodology, Formal analysis, Writing - review & editing. **Antonio Vilches:** Software, Validation, Resources, Visualization. **Andrés Rodríguez:** Software, Validation, Investigation, Visualization. **Francisco Corbera:** Software, Methodology, Investigation. **Rafael Asenjo:** Conceptualization, Data curation, Resources, Writing - review & editing.

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 has been supported by the Spanish project TIN2016-80920-R, by Junta de Andalucía under research projects UMA18-FEDERJA-108. Funding for open access charge: Universidad de Málaga/CBUA.

Appendix. Computation of Euclidean distance using Pearson correlation

The z-normalized Euclidean distance, $d_{i,j}$, between subsequences $T_{i,m}$ and $T_{j,m}$ can be efficiently computed using the following equations:

$$df_1 = 0; \quad df_i = \frac{t_{i+m-1} - t_{i-1}}{2} \tag{23}$$

$$dg_1 = 0; \quad dg_i = (t_{i+m-1} - \mu_i) + (t_{i-1} - \mu_{i-1}) \tag{24}$$

$$Cov_{1,j} = \sum_{k=0}^{m-1} (t_{1+k} - \mu_1)(t_{j+k} - \mu_j) \tag{25}$$

$$Cov_{i,j} = Cov_{i-1,j-1} + df_i \cdot dg_j + df_j \cdot dg_i \tag{26}$$

$$norm_i = \frac{1}{\|T_{i,m} - \mu_i\|} \tag{27}$$

$$P_{i,j} = Cov_{i,j} \cdot norm_i \cdot norm_j \tag{28}$$

$$d_{i,j} = \sqrt{2 \cdot m \cdot (1 - P_{i,j})} \tag{29}$$

where μ_i is the mean of $T_{i,m}$. Basically, the distance between subsequences i and j is computed in Eq. (29) using the Pearson correlation. In turn, Pearson is obtained in Eq. (28) from the covariance of the pair of subsequences (Eq. (26)) and the norms of both subsequences (Eq. (27)). Eq. (26) computes the non-scaled covariance for the range of indexes $2 \leq i \leq n, i < j \leq n$ based on the initialization performed in Eq. (25). Note that indexes i and j start at 1, as done in related works [4,7] for the sake of simplifying notation.

References

[1] C. St-Onge, N. Kara, O.A. Wahab, C. Edstrom, Y. Lemieux, Detection of time series patterns and periodicity of cloud computing workloads, Future Gener. Comput. Syst. 109 (2020) 249–261.

[2] Y. Zhang, Y. Wang, G. Luo, A new optimization algorithm for non-stationary time series prediction based on recurrent neural networks, Future Gener. Comput. Syst. 102 (2020) 738–745.

[3] R. Tomás, J.L. Pastor Navarro, M. Béjar Pizarro, R. Bonì, P. Ezquerro Martín, J.A. Fernández-Merodo, C. Guardiola-Albert, G. Herrera García, C. Meisina, P. Teatini, F. Zucca, C. Zoccarato, A. Franceschini,

[4] C.-C.M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H.A. Dau, D.F. Silva, A. Mueen, E. Keogh, Matrix profile i: all pairs similarity joins for time series: a unifying view that includes motifs, discords and shapelets, in: Data Mining (ICDM), 2016 IEEE 16th Intl. Conf. on, IEEE, 2016, pp. 1317–1322.

[5] Y. Zhu, Z. Zimmerman, N.S. Senobari, C.M. Yeh, G. Funning, A. Mueen, P. Brisk, E. Keogh, Matrix profile II: Exploiting a novel algorithm and GPUs to break the one hundred million barrier for time series motifs and joins, in: 2016 IEEE 16th Intl. Conf. on Data Mining (ICDM), 2016, pp. 739–748.

[6] Y. Zhu, C.-C.M. Yeh, Z.F. Zimmerman, K. Kamgar, E. Keogh, Matrix profile XI: SCRIMP++: Time series motif discovery at interactive speeds, in: 2018 IEEE Intl. Conf. on Data Mining (ICDM), 2018, pp. 837–846.

[7] Z. Zimmerman, K. Kamgar, N.S. Senobari, B. Crites, G. Funning, P. Brisk, E. Keogh, Matrix profile XIV: scaling time series motif discovery with GPUs to break a quintillion pairwise comparisons a day and beyond, in: Proceedings of the ACM Symposium on Cloud Computing, 2019, pp. 74–86.

[8] G. Pfeilschifter, Time Series Analysis with Matrix Profile on HPC Systems (Masterarbeit), Technische Universitẗ München, 2019.

[9] M. Zymbler, A. Polyakov, M. Kipnis, Time series discord discovery on intel many-core systems, in: Parallel Computational Technologies, 2019, pp. 168–182.

[10] I. Fernandez, A. Villegas, E. Gutierrez, O. Plata, Accelerating time series motif discovery in the intel xeon phi KNL processor, J. Supercomput. 75 (11) (2019) 7053–7075.

[11] J.C. Romero, A. Vilches, A. Rodríguez, A. Navarro, R. Asenjo, ScrimpCo: scalable matrix profile on commodity heterogeneous processors, J. Supercomput. (2020).

[12] W. Hussain, F.K. Hussain, M. Saberi, O.K. Hussain, E. Chang, Comparing time series with machine learning-based prediction approaches for violation management in cloud SLAs, Future Gener. Comput. Syst. 89 (2018) 464–477.

[13] E.F. Fama, K.R. French, Comparing cross-section and time-series factor models, Rev. Financ. Stud. 33 (5) (2019) 1891–1926.

[14] S. Torkamani, V. Lohweg, Survey on time series motif discovery, Wiley Interdiscip. Rev.: Data Min. Knowl. Discov. 7 (2) (2017) e1199.

[15] A. Raoofy, R. Karlstetter, D. Yang, C. Trinitis, M. Schulz, Time series mining at petascale performance, in: P. Sadayappan, B.L. Chamberlain, G. Juckeland, H. Ltaief (Eds.), High Performance Computing, 2020, pp. 104–123.

[16] P. Przymus, K. Kaczmarski, Time series queries processing with GPU support, in: New Trends in Databases and Information Systems, Springer, 2014, pp. 53–60.

[17] A.D. Pano-Azucena, E. Tlelo-Cuautle, S.X.-D. Tan, L.G. De la Fraga, FPGA-based implementation of a multilayer perceptron suitable for chaotic time series prediction, Technologies 6 (4) (2018) 90.

[18] X. Zang, Z. Gao, M. Li, X. Wang, FPGA implementation of pulse coupled neural network on for time series of an image, in: Intl. Conf. on Electronics and Electrical Engineering Technology, 2018, pp. 212–216.

[19] J. Liu, J. Wang, Y. Zhou, F. Liu, A cloud server oriented FPGA accelerator for LSTM recurrent neural network, IEEE Access 7 (2019) 122408–122418.

[20] 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.

[21] Intel, Intel oneAPI Programming Guide, 2020.

[22] Khronos Group, SYCL specification: SYCL integrates OpenCL devices with modern C++, v1.2.1, 2019.

[23] A. Navarro, F. Corbera, A. Rodriguez, A. Vilches, R. Asenjo, Heterogeneous parallel_for template for CPU–GPU chips, Int. J. Parallel Program. (2018).

[24] M. Voss, R. Asenjo, J. Reinders, Pro TBB: C++ Parallel Programming with Threading Building Blocks, Apress, 2019.

[25] 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).

[26] 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. Archit. 98 (2019) 27–40.

[27] Intel, Intel FPGA SDK for opencl pro edition, best practices guide, 2020.

[28] Intel, Intel FPGA SDK for opencl pro edition: Programming guide, 2020.

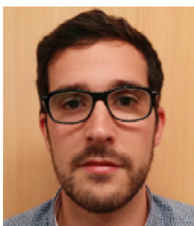
[29] A. Vilches, StratixMonitorLib, Zenodo, 2020, http://dx.doi.org/10.5281/zenodo.3948283.

[30] C.R. Nelson, C.R. Plosser, Trends and random walks in macroeconmic time series: some evidence and implications, J. Monet. Econ. 10 (2) (1982) 139–162.

[31] D. Rudolph, C. Polychronopoulos, An efficient message-passing scheduler based on guided self scheduling, in: 3rd Intl. Conf. on Supercomputing, in: ICS'89, 1989, pp. 50–61.

Jose Carlos Romero received the engineering degree in industrial engineering in 2016 and the Master degree in mechatronics engineering in 2017, both in the University of Malaga. He is currently working toward the Ph.D. degree in the Department of computer Architecture, University of Malaga. His research interests include heterogeneous architectures and parallel programming.

Angeles Navarro obtained a Ph.D. in Computer Science from the University of Malaga, Spain, in 2000. She was an Associate Professor in the Computer Architecture Department at University of Malaga from 2001 until 2019 and Full Professor since then. Currently she is the Vice Dean for Postgraduate Studies in the Computer Science School at the University of Malaga. She was Visiting Scholar in the University of Illinois at Urbana-Champaign (UIUC) in 1996 and 1997, and Visiting Research Associate in the same University in 1998. He also was Research Visitor at IBM T.J. Watson in 2008 and at Cray Inc. in 2011. She has contributed to the Cray's Chapel runtime development since 2011. She has served as a program committee member for several High Performance Computing related conferences as PPoPP, SC, ICS, PACT, IPDPS, ICPP, EuroPar, ISPA. She is the co-leader of the Parallel Programming Models and Compilers group at the University of Malaga. Her research interests are in programming models for heterogeneous systems, analytical modeling, compiler and runtime optimizations.



Antonio Vilches received the MS degree in computer sciences with honors in 2012 from the University of Málaga, Spain. Later, he spent one year working in industry, where he was optimizing on demand web apps with millions of requests per minute. He obtained his Ph.D. degree in 2017 in the Department of Computer Architecture, University of Málaga. His research interests include time series analysis, heterogeneous systems, parallel computing, runtime optimizations, and machine learning methods.

Andrés Rodríguez obtained a Ph.D. in Computer Science Engineering from the University of Malaga, Spain in 2000. From 1996 to 2002, he was an Assistant Professor in the Computer Architecture Department at University of Malaga, 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 and tools for heterogeneous architectures.

Francisco Corbera received the BS and MS degrees in computer science in 1994, from the University of Granada, Spain, and the Ph.D. degree in computer science in 2001, from the University of Málaga, Spain. From 1996 to 2001, he was an assistant professor in the Computer Architecture Department at University of Málaga, and has been an associate professor in the same department since 2002. He lectures on computer technology and architecture. His research interests are in parallelizing compilers and multiprocessor architectures.

Rafael Asenjo Professor of Computer Architecture at the University of Malaga, Spain, obtained a Ph.D. in Telecommunication Engineering in 1997 and was an Associate Professor at the Computer Architecture Department from 2001 to 2017. 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 U. 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 has co-authored the latest book (open access) on Threading Building Blocks (Pro TBB). He is the co-leader of the Parallel Programming Models and Compilers group at the University of Malaga and is ACM Member.