J Supercomput
https://doi.org/10.1007/s11227-018-2367-9

Simultaneous multiprocessing in a software-defined heterogeneous FPGA

Jose Nunez-Yanez[1] · Sam Amiri[1] · Mohammad Hosseinabady[1] · Andrés Rodríguez[2] · Rafael Asenjo[2] · Angeles Navarro[2] · Dario Suarez[3] · Ruben Gran[3]

Abstract Heterogeneous chips that combine CPUs and FPGAs can distribute processing so that the algorithm tasks are mapped onto the most suitable processing element. New software-defined high-level design environments for these chips use general purpose languages such as C++ and OpenCL for hardware and interface generation without the need for register transfer language expertise. These advances in hardware compilers have resulted in significant increases in FPGA design productivity. In this paper, we investigate how to enhance an existing software-defined framework

✉ Sam Amiri
ma17215@bristol.ac.uk

Jose Nunez-Yanez
j.l.nunez-yanez@bristol.ac.uk

Mohammad Hosseinabady
m.hosseinabady@bristol.ac.uk

Andrés Rodríguez
andres@ac.uma.es

Rafael Asenjo
asenjo@ac.uma.es

Angeles Navarro
angeles@ac.uma.es

Dario Suarez
dario@unizar.es

Ruben Gran
rgran@unizar.es

1 University of Bristol, Bristol, UK

2 Universidad de Málaga, Málaga, Spain

3 Universidad de Zaragoza, Zaragoza, Spain

Published online: 16 April 2018

to reduce overheads and enable the utilization of all the available CPU cores in parallel with the FPGA hardware accelerators. Instead of selecting the best processing element for a task and simply offloading onto it, we introduce two schedulers, Dynamic and LogFit, which distribute the tasks among all the resources in an optimal manner. A new platform is created based on interrupts that removes spin-locks and allows the processing cores to sleep when not performing useful work. For a compute-intensive application, we obtained up to 45.56% more throughput and 17.89% less energy consumption when all devices of a Zynq-7000 SoC collaborate in the computation compared against FPGA-only execution.

Keywords FPGAs · Heterogeneous · Interrupts · Dynamic scheduler · LogFit scheduler · Performance improvement · Energy reduction

1 Introduction

Heterogeneous systems are seen as a path forward to deliver the required energy and performance improvements that computing demands over the next decade. In heterogeneous architectures, specialized hardware units accelerate complex tasks. A good example of this trend is the introduction of GPUs (graphics processing units) for general purpose computing combined with multicore CPUs. FPGAs (field programmable gate arrays) are an alternative high-performance technology which offer bit-level parallel computing in contrast with the word-level parallelism deployed in GPUs and CPUs. Bit-level parallel computing fits certain algorithms that cannot be parallelized easily with traditional methods. Recently, the traditional entry barrier of FPGA design, low-level programming languages, has started to being replaced with high-level languages such as C++ and OpenCL successfully [1]. These new programming models and the acceleration capabilities of FPGAs for certain tasks have increased the interest in computing systems that combine CPUs and FPGAs; significant efforts are done not only by FPGA manufactures but also other players such as Intel with their HARP program [2], Microsoft with their Catapult framework [3] and IBM introducing coherent ports for FPGA acceleration in OpenPower [4].

In a typical configuration, the host CPU offloads work onto the FPGA accelerator and then it idles until the accelerator completes the task. In this research, we investigate a cooperative strategy in which both the CPU and FPGA perform the same task on different regions of input data. The proposed scheduling algorithms dynamically distribute different chunks of the iteration space between a dual-core ARM CPU and an FPGA fabric present in the same die. The objective is to measure if simultaneous computing among these devices could be more favorable from energy and/or performance points of view compared with task offloading onto FPGA, while the CPU idles. The FPGA and CPUs are programmed with the same C/C++ language using the SDSoC (software-defined SoC) framework that enables very high productivity and simplifies the development of drivers to interface the processor and logic parts. The novelty of this work can be summarized as follows:

1. Two scheduling algorithms which monitor the throughput of each computing device (CPU cores and FPGA) during the execution of the iteration space and

use this metric to adaptively resize iteration chunks in order to optimize overall throughput and prevent under-utilization and load imbalance;
2. A productive programming interface based on an extension of `parallel_for` template of TBB (Threading Building Blocks) task framework and its integration with the SDSoC framework to allow its exploitation on heterogeneous CPU-FPGA chip processors;
3. A novel solution to the limitations of the SDSoC framework with a new interrupt-based platform eliminating CPU active waiting, reducing energy and clock cycle wasting.

The rest of this paper is organized as follows. Section 2 presents an overview of related work in the area of heterogeneous computing. Section 3 introduces the details of the novel SDSoC simultaneous multiprocessing platform and Sect. 4 the scheduler and programming interface. Section 5 presents the considered benchmarks and their implementation details, and delves into the performance and energy evaluation. Finally, Sect. 6 concludes the paper.

2 Background and related work

The potential of heterogeneous computing to achieve greater energy efficiency and performance by combining traditional processors with unconventional cores such as custom logic, FPGAs or GPGPUs has been receiving increasing attention in recent years. This path has become more attractive as general purpose processors have been unable to scale to higher performance per watt levels with their instruction based architectures. Energy studies have measured over 90% of the energy in a general purpose von Neumann processor as overhead [5] concluding that there is a clear need for other types of cores such as custom logic, FPGAs and GPGPUs. Current efforts at integrating these devices include the Heterogeneous System Architecture (HSA) Foundation [6] that offers a new compute platform infrastructure and associated software stack which allows processors of different types to work efficiently and cooperatively in shared memory from a single source program. The traditional approach of using these systems consists of offloading complex kernels onto the accelerator by selecting the best execution resource at compile time. Microsoft applied their reconfigurable Catapult accelerator which uses Altera FPGAs to a large-scale datacenter application with 1632 servers [3]. Each server contains a single Stratix V FPGA, and the deployment is used in the acceleration of the Bing web search engine. The results show an improvement in the ranking throughput of each server by a factor of 95% for a fixed latency distribution. SnuCL [7] also proposes an OpenCL framework for heterogeneous CPU/GPU clusters, considering how to combine clusters with different GPU and CPU hardware under a single OS image.

The idea of selecting a device at run-time and performing load balancing has been explored in the literature mainly around systems that combine GPUs and CPUs. For example, selection for performance with desktop CPUs and GPUs has been done in [8], which assigns percentages of work to both targets before making a selection based on heuristics. Energy-aware decisions also involving CPUs and GPUs have been considered in [9], requiring proprietary code. Other related work in the context

of streaming applications [10] considers performance and energy when looking for the optimal mapping of pipeline stages to CPU and on-chip GPU. The possibility of using GPU, CPU and FPGA simultaneously and collaboratively has also received attention in diverse application areas such as medical research [11]. The hardware considered uses multiple devices connected through a common PCIe backbone, and the designers optimized how different parts of the application are mapped onto each computing resource. Data are captured and initially processed in the FPGA and then moved to the CPU and GPU components using DMA engines. This type of heterogeneous computing can be considered to connect devices vertically since the idea is to build a streaming pipeline with results from one part of the algorithm moving to the next. The heterogeneous solution achieves a 273 speed up over a multicore CPU implementation. A study of the potential of FPGAs and GPUs to accelerate data center applications is done in [12]. 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. However, it is important to note that the paper does not use high-level languages to increase FPGA productivity as done in this work, and the power measurements for the FPGA are based on worst-case tool estimations and not direct measurements.

In this research, we explore a horizontal collaborative solution. Similar work to ours is done in [13] that presents a tool to automatically compose multiple heterogeneous accelerator cores into a unified hardware thread pool. In comparison, our approach includes host CPU for task execution too and relies on commercially available tools to abstract drivers and run-time creation and focuses on performance enhancements and scheduling. The work closest to ours is [14] that focuses on a multiple device solution and demonstrates how the Nbody simulation can be implemented on a heterogeneous solution in which both FPGA and GPU work together to compute the same kernel on different portions of particles. While our approach uses a dynamic scheduling algorithm to compute the optimal split, in this previous work the split is calculated manually with $\frac{2}{3}$ of the workload assigned to FPGA and the rest to GPU. The collaborative implementation is 22.7 times faster than the CPU-only version.

In summary, we can conclude that the available literature has largely focused on advancing the programming models to make the use of FPGAs in heterogeneous systems more productive, comparing the performance of GPGPUs, FPGAs and CPUs for different types of applications in large-scale clusters, and creating systems that manually choose the optimal device for each part of the application and moving data among them. In contrast in this paper we select a state-of-the-art high-level design flow based on C/C++ for single-chip heterogeneous CPU + FPGA and extend it to support simultaneous computing performing dynamic workload balancing. The selected off-the-shelf devices integrate CPUs and FPGAs and future work will extend this work to integrated GPUs when silicon becomes available.

3 Simultaneous multiprocessing SDSoC platform

3.1 Hardware description

The SDSoC environment generates a hardware acceleration block that can integrate DMA engines to autonomously transfer input/output data without host intervention [15]. The host starts the execution of this block using memory mapped registers accessible with an AXI_LITE slave interface. The host monitors the contents of these registers to determine when the IP block has completed its operation. Additional master interfaces are typically implemented in the acceleration block to communicate input/output data with main memory as it is processed. Depending on the pragmas used at C/C++ level, different types of DMA engines, e.g., DMA_SIMPLE or DMA_SG, are implemented in the programmable logic to stream data from memory to the device and back and require that the data-accesses by algorithm are sequential in virtual memory. The mapping between virtual/physical addresses must be physically contiguous when using DMA_SIMPLE, and memory must be allocated with the SDSoC construct `sds_alloc`, whereas the more complex DMA_SG (Scatter/Gather) does not have this restriction and is used with the standard `malloc`.

Streaming data using DMA engines is generally the most efficient way of FPGA processing; however, this is not applicable if data must be accessed non-sequentially in virtual memory or it must be read multiple times. Another problematic case is if large dataset must be copied to scarce internal FPGA BRAM memory with a hard limit of 16 KB imposed by the tools. Under these circumstances, `zero_copy` pragma can be used to read data directly from external memory when needed using AXIMM interfaces. In our proposed platform, a dedicated AXIMM interface is used to enable the acceleration hardware block to generate interrupts to the host once processing has completed. This removes the need for the host to constantly monitor the status of the hardware registers for completion using the default slave interface and effectively frees one processor for other tasks. This circumstance arises because, in principle, calling a function mapped to hardware in SDSoC means that the host must wait until the function completes before further processing is possible. SDSoC offers `async` and `wait` pragmas to enable asynchronous execution so that the hardware function returns immediately and the host thread can perform other functions until the `wait` statement is reached causing the host to enter into a spin-lock that keeps it busy waiting for the hardware operation to complete. This is rather inefficient since the CPU cannot perform any useful work and consumes energy with 100% utilization. To perform useful work until the `wait` statement is reached requires that the correct amount of work is accurately allocated to the host thread to avoid load imbalance. To address this problem, this work proposes extending the SDSoC framework with an interrupt mechanism so that the host thread is put to sleep before the `wait` statement is reached and will only wake up once the hardware accelerator has completed its task. During this sleep time, the schedulers proposed in Sect. 4 are able to allocate useful work to all the available CPUs. To implement this approach, a new hardware platform is proposed consisting of an additional IP block capable of generating shared interrupts to the main processor as illustrated in Fig. 1.

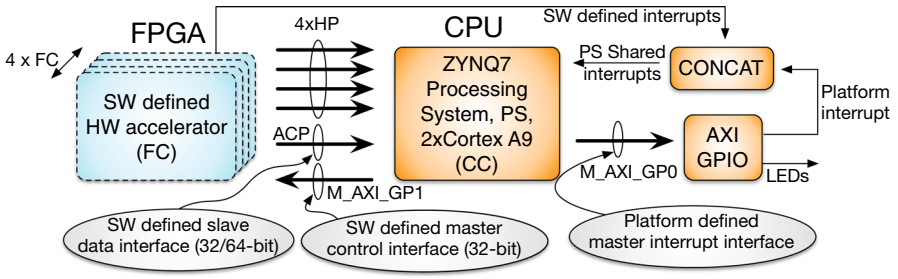

Fig. 1 SDSoC multiprocessing platform

The proposed IP is based on an AXI GPIO block whose interrupt line connects through a concatenate block to the processor interrupt inputs. The concatenate block is a requirement of SDSoC since the tool itself will use interrupts when more than one accelerator block is present. The Linux device tree is modified to inform the Linux kernel of the presence of this hardware block and its capability to generate level sensitive interrupts. The processing system configuration has to be modified to enable interrupts and inform the system which interrupt lane is being used. The Linux kernel driver created by SDSoC for the GPIO is replaced with a custom kernel driver capable of putting the host thread to sleep when requested by the user application. Once the hardware block completes its operation, it writes an AXI GPIO register that generates the interrupt that wakes up the host thread. The function in line 2 of Fig. 2 puts the host thread to sleep, and it is executed by the driver when the user application calls IOCTL after launching the FPGA execution. Once the interrupt takes place, the interrupt service routine present in the driver executes line 3 to wake up the host thread and continue execution.

3.2 Hardware interrupt generation

A critical step is how to get the software-defined hardware accelerator to write the AXI GPIO block and generate interrupts. A direct hardware interface between both blocks is not a feasible option since SDSoC is geared toward communication with main memory using the variables and arrays part of the function prototype. The direct I/O mode available in SDSoC is used to move large amounts of data to the FPGA pins using AXI streaming interfaces and is not useful in this case. Our requirement is writing a single value which triggers interrupt generation. Lines 6–12 of Fig. 2 show the function declaration and calling style for a hardware function called `mmult_top`. The first three parameters of `mmul_top` are pointers identifying the input/output memory areas used to receive/send data, and `size` defines how much data in these memory areas must be processed. When using `async`/`wait` pragmas for hardware function calls, SDSoC constraints require that the function prototype cannot contain a return value. An additional parameter must be supplied, called `scalar` in this example, that will be modified by the hardware function. `scalar` is implemented using AXI_LITE interface and read by the host thread to learn when hardware processing has completed.

```
1  //Linux hardware driver sleep call:
2   wait_event_interruptible(wq, flag != 0);
3   wake_up_interruptible(&wq);
4
5  //Hardware function declaration and calling style:
6   //.h
7   void mmult_top(float *in_a, float *in_b, float *out_c, int size, int *scalar,
        int *interrupt, int enable);
8   //.cpp
9   #pragma SDS async(1)
10  mmult_top(in_a, in_b, out_c, line_count, scalar, interrupt, enable);
11  ret_value = ioctl(file_desc, IOCTL_WAIT_INTERRUPT, 0);
12  #pragma SDS wait(1)
13
14 //Hardware AXIMM interface for interrupt generation:
15  #pragma SDS data zero_copy(interrupt[0:1])
16
17 //Virtual to physical address translation:
18  file_desc = open("/dev/my_driver", O_RDWR);
19  ......
20  sds_mmap((void *)HW_ADDR_GPIO_INT, 4, (void *)interrupt);
21
22 //Hardware interrupt trigger:
23  if (enable == 1)               //enable interrupt generation
24    *interrupt = 0xFFFFFFFF;     //trigger interrupt
25
26 //Interrupt generation with parallel hardware accelerators:
27  enable = 0;    //disable
28  #pragma SDS async(1)
29  aes_enc_hw(state1, cipher1, ekey, block_count, scalar, interrupt, enable);
30  enable = 1;    //enable
31  #pragma SDS async(2)
32  aes_enc_hw(state2, cipher2, ekey, block_count, scalar, interrupt, enable);
33  ret_value = ioctl(file_desc, IOCTL_WAIT_INTERRUPT, 0);
34  #pragma SDS wait(1)
35  #pragma SDS wait(2)
```

Fig. 2 Interrupt mechanism implementation

Having slave interface type AXI_LITE means that this parameter cannot be used to generate the interrupt since it will only be read during the execution of `wait` pragma. The Boolean `enable` parameter is used to enable/disable interrupt generation by the IP block. The mechanism for interrupt generation itself needs a master interface that can write the GPIO register without host intervention. Therefore, an additional pointer is introduced in the function prototype called `interrupt` that must be implemented as an AXIMM master interface. To generate this interface, the `zero_copy` pragma in line 15 is added for `interrupt`, which creates the correct interface so that changes to `interrupt` pointer are reflected directly as desired, without internal buffering. `interrupt` points to the AXI GPIO register that controls interrupt generation to the processor and is written by the IP once processing has completed.

In the code snippet for `mmult_top` we can also see the Linux IOCTL function, line 11, which asks the kernel driver to put the host thread to sleep. The user application opens the device associated with the AXI GPIO peripheral and obtains a virtual address for it using an mmap operation. This opening of the kernel driver is part of the main function and is shown in line 18. It enables the association of a shared processor interrupt with the AXI GPIO device and the use of IOCTL to communicate with the driver. This virtual address is then passed to the hardware function as the interrupt parameter. The SDSoC run-time must translate this virtual address into a physical one

to be usable for the FPGA logic. The mapping between the two addresses must be made explicit to SDSoC with `sds_mmap` in line 20. The first parameter is the physical hardware address for the interrupt registers, the second is the size of the mapping in bytes and the last is the virtual address obtained with the mmap operation for the same registers. Without `sds_mmap`, the SDSoC run-time would not know the mapping between the physical and virtual address. The end result is that the hardware block can write to the interrupt register of the AXI GPIO block simply with the code shown in lines 23–24 which must be located at the end of the hardware function.

Selectively enabling interrupt generation is useful when the FPGA implements several hardware accelerators working in parallel. In this case, only one has its interrupt generation capabilities enabled. The code in lines 27–35 shows an example with a configuration of more than one accelerator. Only the second core generates interrupts, and it is expected to be the last one to complete its operation since it is activated after the first core. Even if this was not the case, functionality will not be affected since the `wait` statements will correctly synchronize the accelerator execution with the rest of the algorithm.

4 Programming environment

In this section, we introduce our heterogeneous building blocks (HBB) library API. HBB is a C++ template library that facilitates the configuration and use of heterogeneous computing units by automatically partitioning and scheduling the workload among the CPU cores and the accelerator. It builds on top of SDS (Xilinx SDSoC library) and TBB libraries and offers a `parallel_for()` function template to run on heterogeneous CPU-FPGA systems. HBB could also be developed on top of FastFlow [16] instead of relying on TBB. However, TBB is more suitable to our needs as the task-based work-stealing feature of TBB improves load balance and avoids oversubscription when composing or nesting different parallel patterns. Figure 3 shows an MPSoC with one FPGA and two CPU cores (CCs) as the system used in our experimental evaluations. The FPGA itself can contain a number of hardware accelerators or FPGA compute units (FCs), depending on resource availability and FPGA configuration.

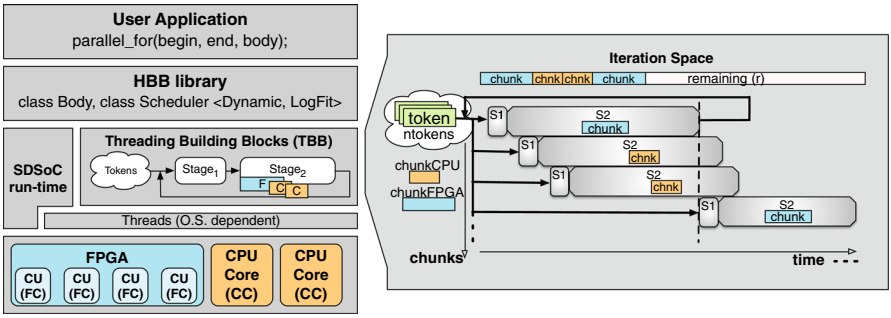

Fig. 3 Our heterogeneous scheduling design

The left part of Fig. 3 shows the software stack that supports the user application. Our library (HBB) offers an abstraction layer that hides the initialization and management details of TBB and SDS constructs; thus, the user can focus on the application instead of dealing with thread management and synchronization. The library includes a heterogeneous scheduler that takes care of splitting the iteration space into chunks and processes each chunk on a CC or an FPGA device. In the more complex version of such scheduler, the size of the chunks offloaded to the FPGA or processed on the CC is adaptively computed so that it fully utilizes the computational resources while maximizing the load balance among the CCs and FPGA. The right part of this figure shows that the internal engine managing the `parallel_for()` function is a two-stage pipeline, Stage_1 (S1) and Stage_2 (S2), implemented with the TBB pipeline template. The iteration space consists of the chunks already assigned to a processing unit and the remaining iterations waiting to be assigned. An execution of the pipeline with three tokens (1 FPGA + 2 CCs) is also shown. The tokens represent the number of chunks of iterations that can be processed in parallel on our computing resources (FPGA and 2 CPU cores). S1 is a serial stage so it can host just a token at a time, whereas S2 is parallel so there can be several tokens in this stage simultaneously. Once a token enters in S1, the FPGA device is checked; if idle, it is acquired and initialized with a FPGA chunk. Otherwise a CPU core is idle and is initialized with a CPU chunk. In either case, the partitioner extracts the corresponding chunk of iterations from the set of remaining iterations, r. Then, the token proceeds to S2 where the selected chunk is executed on the corresponding computing resource. When S2 finishes the computation of a token on FPGA, it releases the FPGA by changing its status to idle and allowing other tokens to acquire it. The processing speed of an offloaded chunk on the FPGA will depend on the number of FCs implemented, and this number is transparent to our schedulers. The time required for the computation of each chunk on the FPGA or CPU core is recorded and used to update the relative speed of the FPGA w.r.t. a CPU core, called f. Factor f is used to adaptively adjust the size of the next chunk assigned to a CPU core.

A major advantage of this `parallel_for` implementation is its asynchronous mode of computing [17] because each computing resource hosts one of the tokens that traverses the pipeline, independently carrying out the corresponding work. Thus, unnecessary synchronization points between computing resources with different performances are avoided. In contrast, other state-of-the-art approaches [18,19] suffer from load imbalance due to use of `fork-join` patterns with implicit synchronization points between CPU and accelerator.

4.1 Parallel_for template and HBB classes

Before using the `parallel_for()` function, the user must define a `Body` class, the body of the parallel loop, as shown in lines 2–11 of Fig. 4. This class should implement two methods: one that defines the code which each CPU core will execute for an arbitrary chunk of iterations, and the same for the FPGA device using a C++ function call. The `operatorCPU()` method (lines 4–7) defines the CPU code of the kernel in C++, and the `operatorFPGA()` method (lines 8–10) calls a hardware function

```
1  //Definition of Class Body:
2   class Body{
3   public:
4   void operatorCPU(int begin, int end){
5    for(i=begin; i!=end; i++)
6           c[i] = a[i] * b[i];
7   }
8   void operatorFPGA() (int begin, int end){
9    int result = kernelVectorMul((float*) a, (float*) b, (float*) c, begin, end);
10  }
11  };
12  ...
13
14 //Using the parallel_for() function template
15  #include "hbb.h"
16  ...
17  int main(int argc, char* argv[]){
18   Body body;
19   Params p;
20   InitParams(argc, argv, &p);
21  //Instantiate task scheduler
22  //Dynamic * hs = Dynamic::getInstance(&p);
23   LogFit * hs = LogFit::getInstance(&p);
24   ...
25   hs->parallel_for(begin, end, body);
26   ...
27  }
```

Fig. 4 Applying parallel execution

that has already been implemented in the FPGA using the SDSoC development flow. The array pointers passed to the FPGA are normally shared between CPU and FPGA, and the preferred option to connect the FPGA accelerator to the processing system is to use the ACP coherent port.[1] Using this approach, SDSoC automatically manages the data movement from global memory to the FPGA and back.

Orchestrating the body execution and handling heterogeneous devices require a `Scheduler` class that provides methods isolating the `parallel_for()` function template and the scheduling policies from device initialization, termination and management. The compartmentalization simplifies the adoption of different devices and more importantly enables the programmers to focus on scheduling policies, such as the Dynamic and LogFit policies described in Sect. 4.2. Instead of implementing error prone low-level management tasks such as thread handling or synchronization operations, the `Scheduler` leverages TBB for them.

A `main` function is shown in lines 17–27 of Fig. 4 with required component initialization to make the `parallel_for()` function template work. This is the main component of the `HBB` library, made available by including `hbb.h` header file. The user has to create a `Body` instance (line 18) that will later be passed to the `parallel_for()` function. Program arguments such as the number of threads and scheduler configuration can be read from the command-line, as seen in line 19. The benchmarks that we evaluate in Sect. 5 accept at least three command-line arguments: `<num_cpu_tokens>`, `<num_fpga_tokens>` and `<sch_arg>`. The first one sets the number of CPU tokens, which translates into how many CPU cores will be processing chunks of the iteration space. The second one can be set to 0 or 1 to dis-

[1] The ACP port allows FPGA and CPU cores to access the shared L2 cache.

able or enable the FPGA as an additional computing resource. The last argument is a positive integer setting the size of iteration chunks to be offloaded onto the FPGA. The CPU chunk sizes are adaptively computed as described in the next section. The `parallel_for()` function template receives three parameters (line 25): the first two parameters, `begin` and `end`, define the iteration space, and the third one is the `Body` instance which has the implementation of the CPU and FPGA body loop.

4.2 Scheduler implementation

This section explains how the chunk sizes for execution by the CPU cores and the FPGA are calculated. We have designed and implemented two scheduling methods: Dynamic and LogFit.

With Dynamic scheduler, the FPGA chunk size is manually set by the user (S_f = `<sch_arg>`), whereas the CPU chunk size is automatically computed by a heuristic. This heuristic aims to adaptively set the chunk size for the CPU cores. To that end, the model described in [17] recommends that: *each time that a chunk is partitioned to be assigned to a compute unit, its size should be selected such that it is proportional to the compute unit's effective throughput*. This heterogeneous Dynamic scheduler is a combination of the OpenMP dynamic scheduler [20] for the FPGA chunks and the OpenMP guided scheduler for the CPU chunks. Therefore, if r is the number of remaining iterations (initially $r = n$), the computation of the CPU chunk, S_c, is:

$$S_c = \min\left(\frac{S_f}{f}, \frac{r}{f + nCores}\right) \tag{1}$$

where f represents how faster the FPGA is with regard to a CPU core, and it is recomputed each time a chunk is processed, as explained in Sect. 4. In other words, S_c is either (S_f/f) (the number of iterations that a CPU core must perform to consume the same time as the FPGA) when the number of remaining iterations, r, is sufficiently high, or $r/(f + nCores)$ (a "guided self-scheduling strategy" [21]), when there are few remaining iterations (this is when $r/(f + nCores) < S_f/f$).

The Dynamic scheduling has the limitation of requiring the user to input the FPGA chunk size which might not result in the best performance. LogFit scheduler [22] avoids this limitation by dynamically computing the FPGA chunk size at run-time. LogFit consists of three phases:

1. Exploration phase: the throughput for different FPGA chunk sizes is examined and a logarithmic fitting is applied to find the FPGA chunk size that maximizes the FPGA throughput. CPU chunk sizes which balance the load are also computed in this phase by Eq. 1.
2. Stable phase: FPGA throughput is monitored and adaptive FPGA chunk sizes are re-adjusted following the previously computed logarithmic fitting. CPU chunk sizes are accordingly computed using Eq. 1 in order to keep the load balanced.
3. Final phase: When there are not enough remaining iterations to feed all computing resources, the best possible partitioning to minimize the computation time is

estimated. In this phase, these remaining iterations are assigned only to the CPU, only to the FPGA, or to both, depending on which configuration is the fastest one.

The overhead of our Dynamic and LogFit schedulers can be measured as the time consumed in Stage_1, which in our experiments is always less than 0.2% of the total execution time.

5 Experimental evaluation

The Dynamic and LogFit schedulers introduced in this article are evaluated with four benchmarks that vary in terms of compute and memory intensity. Every benchmark is coded in C/C++ for both CPU and FPGA targets; g++ (of GCC) and sds++ (of Xilinx SDSoC) compilers are used to compile for CPU and FPGA, respectively.

The Advanced Encryption Standard (AES) is a cryptographic algorithm for the encryption of electronic data widely used in embedded systems. Our implementation of AES uses a 256-bit key length and every block consists of 16-bytes. Four basic byte manipulation functions are employed according to the standard: subbytes, mixcolumn, addroundkey and shiftrow, that repeat for a total of 15 rounds. In our evaluation, we encrypt a 16MB file which results in a `parallel_for` with 1048576 iterations.

HotSpot is a stencil algorithm that has been extracted from the Rodinia benchmark collection. This algorithm estimates the temperature propagation on the surface of a chip through thermal simulation which iteratively solves a series of differential equations. We simulate a chip discretized in 1024×1024 points, and the `parallel_for` traverses the outermost loop with 1024 iterations. This experiment is repeated 50 times. The FPGA implementation is based on the shift register pattern (SRP) optimization that is known to deliver near optimal performance [23].

Nbody is a traditional high-performance physics simulation problem that computes how a dynamic system of particles behaves under the influences of forces such as gravity. We consider the brute force Nbody algorithm which is highly regular but has a time complexity of $O(n^2)$, with n being the number of simulated bodies. Nbody is run with 50 K bodies in our experiments which coincides with the iteration space of the `parallel_for`. For efficient implementation on FPGA, the particle data should be buffered in the internal memories to avoid constant reloading of data from external memories. For this application, because the same data must be accessed multiple times and its size exceeds the 16 KB SDSoC limitation for internal memory, SDSoC DMA-based approaches are not feasible and an AXI master interface is necessary which is efficient as long as AXI bursts are generated.

General matrix multiplication (GEMM) is a dense matrix–matrix multiplication application. The large matrix sizes considered (1024×1024 floating-point values) prevent us from buffering the whole data in FPGA memory. Therefore, the implementation relies on the tiling optimization to exploit locality. We parallelize the outermost loop that contains 1024 iterations. The interfaces are based on AXIMM that can also be very high performance since the long burst modes available in AXI can be used effectively by the created custom logic.

These benchmarks are evaluated on a ZC702 board equipped with a Zynq 7020 device. This board includes a PMBus (Power Management Bus) power control and

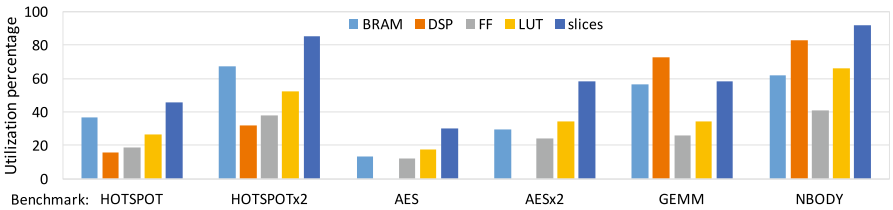

Fig. 5 FPGA resource utilization (×2 indicates two FPGA compute units)

monitoring system which enables the reading of power and current every 0.1 s. For the power measurements, the power values corresponding to the processing system (CPU cores), programmable logic (FPGA) and memory are measured and added together. The overall energy consumption is obtained by integrating the power readouts during the execution time of the benchmark.

Figure 5 shows the percentages of FPGA resource utilization for our benchmarks. The implementations for Nbody and GEMM already use the available resources with a single FPGA compute unit (1FC) configuration, but for AES and HotSpot, compute unit replication is possible so we can implement 2 compute units (2FC) in these cases. The GEMM benchmark has DSP capacity as its limiting factor, while for the other benchmarks it is the slice utilization.

In the following, performance and power/energy results of the benchmarks implementations on ZC702 board are reported and discussed. For the Dynamic scheduler, these results are given for each combination of the two available CPU cores (CCs) and one or two FCs. For the LogFit scheduler, we only report the throughput obtained when fully exploiting the platform (utilizing all CCs and FCs). In general, there are several trends in the obtained results:

- The throughput of CPU-only implementations is independent of the block size, but the overall performance is improved with larger blocks when the FPGA is involved. The FPGA benefits with less overheads due to setting up the data movement and filling the deep pipelines with large blocks.
- The performance and energy costs obtained by the LogFit scheduler are comparable to the best performance and energy costs obtained by the Dynamic scheduler.
- As the number of cores increases the overall energy consumed generally decreases despite the increase in power due to each extra core added to the configuration.

5.1 Performance evaluation

In Fig. 6, performance is reported as throughput (number of iterations processed per second) in the *y*-axis, and the *x*-axis indicates the FPGA chunk size. For the Dynamic scheduler, each line in the plot shows the achieved throughput for each platform configuration (only CPU –1CC, 2CC–, only FPGA –1FC, 2FC– and various heterogeneous combinations) and for different fixed FPGA chunk sizes. The LogFit result is indicated with a single green plus, "+," sign that shows the resulting average throughput and average FPGA chunk size, for the best platform configuration.

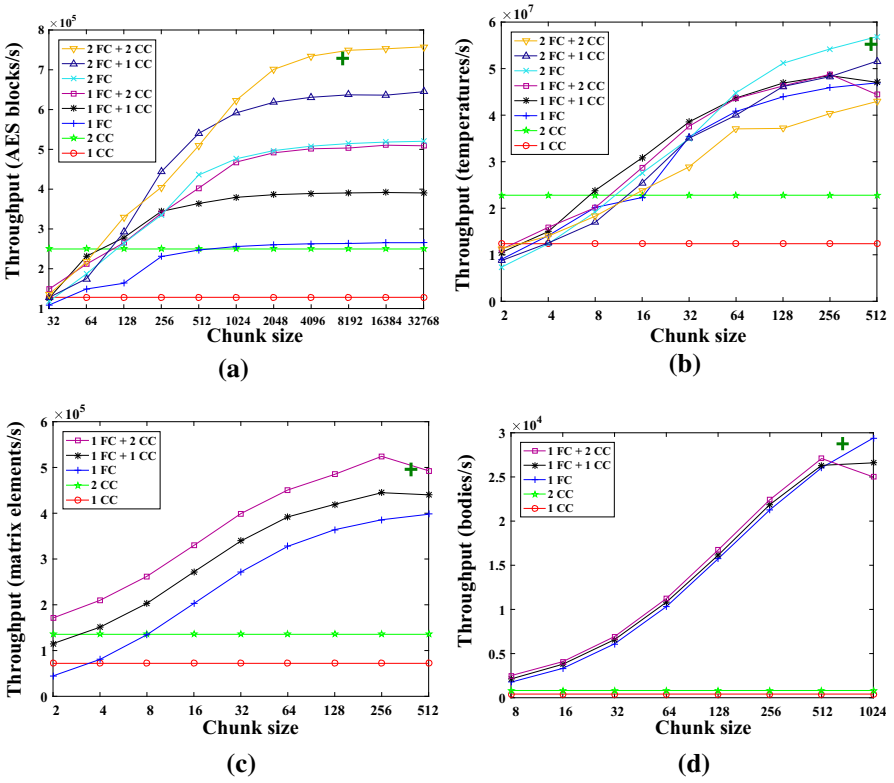

Fig. 6 **a** AES, **b** HotSpot, **c** GEMM and **d** Nbody performance evaluation

AES performance results are shown in Fig. 6a. With Dynamic scheduler, a higher performance is achieved as the number of participating cores and also chunk size increase. In "2FC + 2CC", 69% of the workload is performed by the FCs, achieving 45.56% higher throughput than the FPGA-only configuration (2FC). In contrast, "1FC + 2CC" configuration offloads only 53% of the workload to the FC and obtains 48.84% less throughput than "2FC + 2CC." This figure also shows the high performance obtained using the LogFit scheduler compared with the Dynamic scheduler.

HotSpot can run in a full streaming configuration and needs a very high bandwidth access to main memory to maintain the execution units busy. This high bandwidth demand represents a bottleneck when several cores compete for access. Moreover, the filtering operation of HotSpot can be better accelerated in FPGA than CPU. Hence in this memory-intensive application, the addition of extra CPU cores may not help with the performance. Figure 6b shows the throughput evaluation of this application. The "2FC" configuration has the best performance and adding extra CPU cores degrades it by 24.49%. With "2FC + 2CC" configuration, 27% of the workload is assigned to CCs. Since cache coherent interconnect (CCI) unit provides DDR memory accesses for both CPU cores and the accelerator when ACP port is used, including CPU cores to an already high bandwidth demanding FPGA overwhelms CCI with memory accesses

that decreases overall throughput. For the "2FC + 2CC" configuration, the performance obtained with LogFit is 28.15% higher than the Dynamic one. This is because LogFit automatically detects that the FPGA is faster and offloads 93% of the iteration space to the accelerator.

Figure 6c shows the performance gains for GEMM application resulting from the heterogeneous execution and the benefits of assigning larger blocks for hardware execution when Dynamic scheduler is used. The fastest configuration is "1FC + 2CC" where 75% of workload is assigned to FPGA and an improvement of 35.90% is obtained w.r.t. FPGA-only solution.

The Nbody algorithm has a very high computational intensity, and its best FPGA implementation outperforms its dual-core CPU one by a factor of $\sim 35\times$. The heterogeneous solution of this application hardly improves the throughput due to the highly efficient FPGA implementation, as shown in Fig. 6d. At the fastest "1FC + 2CC" configuration, over 97% of the workload is assigned to the FPGA. In this case, offloading the whole iteration space to the single FPGA core delivers the best performance.

In summary, a higher efficiency heterogeneous solution is observed when memory access bottlenecks are not a factor and the computational efficiency of one device is not marginal compared to the others.

5.2 Power and energy evaluation

Figure 7 illustrates the power dissipation and energy consumption evaluations for the four benchmarks. Dynamic scheduling results are reported for all platform configurations and the best FPGA chunk size. LogFit results are shown for the best platform configuration.

For AES, the results of Fig. 7a indicate that energy reduces with faster configurations despite the additional power costs of using the extra cores. For this benchmark, the "2FC + 2CC" configuration results in 17.89% less energy consumption than "2FC". Regarding HotSpot, the evaluations in Fig. 7b show that the most energy efficient configuration is "2FC" as this configuration has the best performance too. Energy use increases for the most complex configurations due to the additional power requirements which do not translate into significant faster execution. The power/energy values shown in Fig. 7c for GEMM indicate that the energy costs are almost equivalent for all configurations which use the FPGA and lower than the CPU-only configurations. With respect to Nbody, the measurements in Fig. 7d confirm that the most energy efficient solution is the FPGA despite the significant increase in power observed when the FPGA core is used due to the high arithmetic intensity.

5.3 Impact of the interrupt implementation

The interrupt-enabled platform described in this paper provides an efficient utilization of all the CPU cores. To show the impact of this technique on throughput and energy, Tables 1 and 2 list the throughput and energy changes, respectively, when enabling the utilization of the second CPU core with and without the interrupt mechanism for the Dynamic and LogFit schedulers. While AES and GEMM evidently gain higher

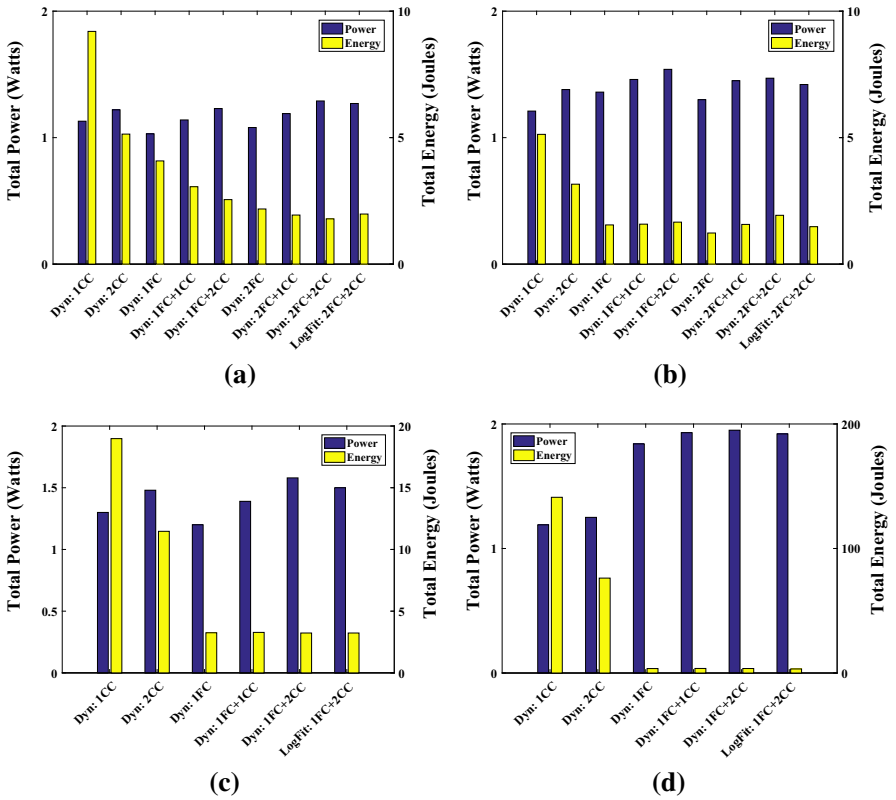

Fig. 7 **a** AES, **b** HotSpot, **c** GEMM and **d** Nbody power and energy evaluation

Table 1 Throughput change when the second CPU core is utilized

	AES (%)	HotSpot (%)	GEMM (%)	Nbody (%)
Dynamic with interrupt	+ 17.49	− 16.79	+ 17.85	+ 2.93
Dynamic without interrupt	0.00	− 3.09	+ 2.61	− 2.65
LogFit with interrupt	+ 26.85	− 16.17	+ 20.59	+ 1.48
LogFit without interrupt	− 19.62	− 9.05	+ 4.71	− 0.07

Positive values indicate improvement in throughput. The higher the better

throughput exploiting the interrupt mechanism, HotSpot loses some throughput and Nbody shows marginal improvement. These results indicate that freeing up CPUs with the interrupt mechanism does not help performance if the CPUs are assigned tasks which are executed extremely efficiently by the FPGA or if the algorithm memory intensity introduces additional contention. The energy changes of the benchmarks follow almost the same improvement pattern as the throughput changes.

Table 2 Energy change when the second CPU core is utilized

	AES (%)	HotSpot (%)	GEMM (%)	Nbody (%)
Dynamic with interrupt	− 7.73	+ 22.93	− 1.52	− 1.64
Dynamic without interrupt	− 1.53	+ 2.14	+ 13.64	+ 11.67
LogFit with interrupt	− 19.51	+ 16.54	+ 2.87	+ 6.89
LogFit without interrupt	+ 16.00	+ 10.16	+ 1.12	+ 7.59

Negative values indicate improvement in energy consumption. The lower the better

6 Conclusion

This paper introduces a multiprocessing methodology to dynamically distribute workloads in heterogeneous systems composed of CPU and FPGA compute resources. To minimize execution time, the methodology tunes the amount of work each device computes for each application depending on its compute performance.

We create two scheduling strategies and enhance the SDSoC framework with an interrupt mechanism to signal task completion. Overall, we find that heterogeneous execution improves throughput and reduces energy as long as the performance of the FPGA is not significantly higher than the CPU cores and enough memory access bandwidth is available for all the cores. Even with a modest amount of CPU participation, e.g., 25% in GEMM, a noticeable performance gain can be achieved. The single coherence port available in the Zynq family negatively affects memory-intensive applications such as HotSpot. Our future work involves porting this methodology to more capable devices such as the Zynq Ultrascale family that uses a quad-core 64-bit configuration and provides additional cache coherent ports to access main memory.

Acknowledgements We acknowledge with gratitude the funding received from EPSRC UK in grant EP/N002539/1: (ENEAC). The framework and algorithms created in this work are available at https://github.com/eejlny/gphcu to enable reproducible results and further research.

References

1. Arcas-Abella O et al (2014) An empirical evaluation of high-level synthesis languages and tools for database acceleration. In: FPL '14, pp 1–8
2. Xeon+FPGA platform for the data center. www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf. Accessed 2018 Mar 05
3. Putnam A et al (2014) A reconfigurable fabric for accelerating large-scale datacenter services. In: International symposium on computer architecture, ISCA '14, pp 13–24
4. Enabling coherent FPGA acceleration. www.openpowerfoundation.org/wp-content/uploads/2015/03/Cantle_OPFS2015_Nallatech_031315_final.pdf. Accessed 2018 Mar 05
5. Chung ES, Milder PA, Hoe JC, Mai K (2010) Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In: Micro '10
6. HSA Foundation. www.hsafoundation.com. Accessed 2018 Mar 05

7. Auerbach J et al (2012) A compiler and runtime for heterogeneous computing. DAC '12, pp 271–276
8. Pandit P, Govindarajan R (2014) Fluidic kernels: cooperative execution of OpenCL programs on multiple heterogeneous devices. CGO '14, pp 273:273–273:283
9. Dolbeau R, Bodin F, de Verdiere GC (2013) One OpenCL to rule them all? In: MuCoCoS '13, pp 1–6
10. Vilches A et al (2016) Mapping streaming applications on commodity multi-CPU and GPU on-chip processors. IEEE Trans Parallel Distrib Syst 27(4):1099–1115
11. Meng P, Jacobsen M, Kastner R (2012) FPGA-GPU-CPU heterogeneous architecture for real-time cardiac physiological optical mapping. In: FPT '12, pp 37–42
12. Prongnuch S, Wiangtong T (2014) Heterogeneous computing platform for data processing. In: ISPACS '16, pp 1–4x
13. Korinth J, de la Chevallerie D, Koch A (2015) An open-source tool flow for the composition of reconfigurable hardware thread pool architectures. In: FCCM '15, pp 195–198
14. Tsoi KH, Luk W (2010) Axel: A heterogeneous cluster with FPGAs and GPUs. FPGA '10, pp 115–124
15. SDSoC environment user guide. www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug1027-sdsoc-user-guide.pdf. Accessed 2018 Jan 22
16. Aldinucci M, Danelutto M, Kilpatrick P, Torquati M (2017) Fastflow: high-level and efficient streaming on multicore. Wiley, Hoboken, pp 261–280
17. Navarro A, Vilches A, Corbera F, Asenjo R (2014) Strategies for maximizing utilization on multi-CPU and multi-GPU heterogeneous architectures. J Supercomput 70:756–771
18. Luk CK, Hong S, Kim H (2009) Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: Proceedings of the Micro, pp 45–55
19. Wang Z, Zheng L, Chen Q, Guo M (2014) CPU+GPU scheduling with asymptotic profiling. Parallel Comput 2:107–115
20. Dagum L, Menon R (1998) OpenMP: an industry standard API for shared-memory programming. IEEE Comput Sci Eng 5(1):46–55
21. Rudolph DC, Polychronopoulos CD (1989) An efficient message-passing scheduler based on guided self scheduling. ICS '89, pp 50–61
22. Vilches A et al (2015) Adaptive partitioning for irregular applications on heterogeneous CPU-GPU chips. Procedia Compu Sci 51:140–149
23. Jia Q, Zhou H (2016) Tuning Stencil codes in OpenCL for FPGAs. In: ICCD '16, pp 249–256