Exploring Heterogeneous Scheduling for Edge Computing with CPU and FPGA MPSoCs

Andrés Rodríguez[a], Angeles Navarro[a], Rafael Asenjo[a], Francisco Corbera[a], Rubén Gran[b], Darío Suárez[b], Jose Nunez-Yanez[c]

[a] *Dept. of Computer Architecture, Universidad de Málaga, Spain. {andres, angeles, asenjo, corbera}@ac.uma.es*
[b] *Computer Architecture Group. Universidad de Zaragoza, Spain. {rgran,dario}@unizar.es*
[c] *Dept. of Electrical & Electronic Engineering. University of Bristol, UK. J.L.Nunez-Yanez@bristol.ac.uk*

Abstract

This paper presents a framework targeted to low-cost and low-power heterogeneous MultiProcessors that exploits FPGAs and multicore CPUs, with the overarching goal of providing developers with a productive programming model and runtime support to fully use all the processing resources available. FPGA productivity is achieved using a high-level programming model based on OpenCL, the standard for cross-platform parallel heterogeneous programming. In this work, we focus on the *parallel_for* pattern, and as part of the runtime support for this pattern, we leverage a new scheduler that strives to maximize the number of iterations per joule by dynamically and adaptively partitioning the iteration space between the multicore and the accelerator when working simultaneously. A total of 7 benchmarks are ported and optimized for a low-cost DE1 board. The results show that the heterogeneous solution can improve performance up to 2.9x and increases energy efficiency up to 2.7x compared to the traditional approach of keeping all the CPU cores idle while the accelerator computes the workload. Our results also demonstrate two interesting insights: first, an adaptive scheduler able to find at runtime the right chunk size for each type of application and device configuration is an essential component for these kinds of heterogeneous platforms, and second, device configurations that provide higher throughput do not always achieve better energy efficiency when only the running power (excluding the idle power component) is considered.

1. Introduction and Motivation

The trend towards embedding connected sensors everywhere pushes for a computational model where part of the compute is done at the edge instead of on cloud. Therefore, we see more powerful devices at the edge that also have to drain a limited amount of power. For example the analysis in [1] highlights that edge computing is well equipped to handle privacy and connectivity issues but some advantages such as low latency are only realizable if enough local computation power is provided. A possible solution to these performance requirements is low-power heterogeneous systems suitable for edge computing.

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. Recent research projects in the context of exascale have proposed new architectures based on multicore CPUs and integrated reconfigurable resources (e.g. FPGAs) that target to significantly improve the performance over power dissipation ratio [2], [3]. Moreover, processor vendors have integrated general-purpose multicore CPUs and FPGA into the same chip for data center acceleration or specialized embedded applications [4].

In this research work we target low-cost and low-power heterogeneous embedded systems that tightly couple an ARM CPU multicore and an on-chip FPGA, usually called heterogeneous MultiProcessor System-on-Chip or MPSoC.

ARM processors are receiving significant attention and have gained a lot of traction in the market as low-power alternatives to the X86 architecture. The selected low-cost platform combines a dual-core 32-bit

ARM processor and a small low-power FPGA fabric. This makes it suitable to work at the edge near data-collecting sensors instead of a classical high-power and high-performance computing set-up. In this arrangement, the near sensor device runs the same algorithms as the high-end server-class device but over coarse data, and it identifies events that contain anomalies that are then sent to the server over the network for further processing at higher resolution. For example, unexpected temperature variations in the surface of a device that exceed a threshold and require further processing to identify a possible fault condition. For this reason, in this work, we also select some benchmarks from the high performance computing world to run in the near-edge heterogeneous platform. An alternative to use accelerators tightly coupled to the ARM processor is to build SoCs with processors with different levels of complexity, power and performance based on a single instruction set architecture such as the commercially successful ARM big.LITTLE multicores [5]. In our work we focus on heterogeneous system that need further hardware acceleration to obtain the required performance.

Edge computing platforms calls for advances in three conflicting requirements: productivity, flexibility, and excellent power-performance ratio. To provide all three, we adapt a framework originally designed for heterogeneous compute on HPC systems to low-cost CPU+FPGA MPSoCs. Excellent power-performance ratio comes from the use of the FPGA, and productivity and flexibility comes from the use of high-level synthesis with OpenCL and a simple yet powerful scheduling policy, named HAP, that automatically balances the workload between the FPGA and the cores.

This work addresses these challenges by combining OpenCL -to code the functions that can be executed on the FPGA- with Threading Building Blocks (TBB [6]) -to orchestrate and distribute the iterations of a parallel_for among the cores and the FPGA-. We have recently proposed a scheduling algorithm that dynamically distributes different chunks[1] of the parallel iteration space among CPU cores and a GPU [7]. To this end, the scheduler monitors the throughput of each computing unit during the execution of the iterations and uses this metric to adaptively resize the CPU and GPU chunks in order to optimize overall throughput and to prevent underutilization and load unbalance between GPU and CPU cores. In this work, we extend this scheduler to work on FPGA + CPU configurations and tune the evaluated benchmarks to fully exploit both devices. The results indicate that simultaneous computing on all devices is more favorable in many cases from the energy and/or performance points of view compared to just off-loading all the workload to the accelerator and keep the CPU idling.

Summarizing, this paper proposes the following novel contributions:

1. An extension of our tunable scheduling framework, now enabling both a CPU and an OpenCL-capable FPGA to concurrently perform the same task on disjoint regions of the data.
2. An evaluation of our heterogeneous scheduler, called HAP (from Heterogeneous Adaptive Partitioner), which adaptively and continuously adjusts the size of the chunk of iterations offloaded to CPU cores and the FPGA, a feature particularly relevant for irregular applications. We compare this scheduler against Static and Dynamic scheduling strategies and find that HAP always discovers the near-optimal chunk sizes for all device configurations without requiring manually tuning, and at a reasonable cost even for regular applications.
3. A thorough exploration of the performance and energy efficiency interactions for different device configurations using 7 applications in which the main kernels have been carefully tuned for the low-power Cyclone V FPGA. We discover that heterogeneous configurations that simultaneously exploit the CPU cores and the FPGA are the highest performance and energy efficient for irregular applications, while for regular applications a careful selection of the devices must be considered depending on the metric to optimize: performance, energy efficiency due to running power (excluding the idle power) or energy efficiency due to total power.

The rest of the paper is organized as follows. The next section introduces the problem as well as related works that tackle a similar problem. Section 3 briefly describes the proposed programming interface and a summary of implementation details for the different scheduling strategies that have been studied. The

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

evaluated benchmarks and their optimization to target a low-cost and low-power FPGA are covered in Section 5. In Section 6 we present the testbed as well as a thorough and extensive experimental evaluation. Finally, we wrap up with conclusions in Section 7.

2. Background and related work

Heterogeneous systems featuring hardware accelerators are being used for a wide range of applications yielding significant speedups compared to CPU-only execution, specifically in the context of image processing, data mining, encryption or network packet processing [8]. However, the development of an easy programming framework and its runtime support for a broad class of applications in these systems is still an open problem. The success of these systems as general-purpose architectures heavily depends on how easy we can map application-level parallelism onto the available heterogeneous hardware resources. In this respect, the acceptance of FPGAs has been hampered due to their slow and error-prone development environments until a new generation of High Level Synthesis (HLS) and implementation tools have managed to break from the academic and research environments to the commercial arena [9]. This has resulted in renewed interest on how to integrate FPGAs in domains such as High Performance Computing and the data center. For example, Altera (now Intel) introduced their OpenCL 1.0 compliant compiler (i.e. AOCL) in 2013. This technology is now being developed by Intel in their Xeon+FPGA accelerator systems for the data center market. Xilinx has also invested heavily in this area with their Vivado HLS tools and the SDAccel OpenCL and SDSoC C/C++ frameworks that use Vivado HLS as their back-ends [10].

The traditional approach of using heterogeneous platforms consists of offloading complex kernels to the accelerator by selecting the best execution resource at compile time. For example, IBM proposed an extension of Java called Lime [11] to target CPU+FPGA systems. Based on a task-based data-flow programming, Lime allows program partitioning at task granularity level similar to a block-level diagram of an FPGA circuit. Its main drawback is the need of introducing an unfamiliar language and the lack of support from the FPGA vendors. SnuCL also proposes [12] an OpenCL framework for heterogeneous CPU/GPU clusters, considering how to combine clusters with different GPU and CPU hardware under a single OS image. Clusters of independent devices have also been studied in papers such as [13]. Their experimental results based on four application examples confirm that different applications have different favorite computing architectures.

However, there are cases in which it makes sense to exploit all the available computing devices at once. In those cases, run-time approaches are more appropriate to automatically partition and balance the workload. This idea has been explored previously, but mainly for heterogeneous systems composed of CPU+GPU. For example, Fluidic [14] allows for CPU+GPU collaborative execution of parallel loops. The GPU starts computing a data buffer in ascending order in parallel with the CPU that process sub-ranges (chunks) of iterations in descending order. The GPU is responsible of aborting its execution when it tries to execute an iteration that has been already processed by the CPU. In some cases a conflicting section of the buffer can be computed twice (once by each device). In addition to parallel_for, the pipeline pattern has also been implemented for heterogeneous chips composed of CPU plus an integrated GPU [15]. In that work an analytical model is coupled with on-line profile data in order to automatically identify which pipeline stages should be mapped onto the GPU, according to throughput and energy consumption metrics.

The possibility of using CPU+GPU and FPGA simultaneously and collaboratively has also received attention in diverse application areas such as medical research [16]. The hardware considered uses multiple devices connected through a common PCIe backbone, and the designers optimized how different parts of the application are mapped to each computing resource. This type of heterogeneous computing can be considered to connect devices vertically (i.e. exploit temporal parallelism) since the idea is to build a streaming pipeline with results from one part of the algorithm moving to the next. Data is captured and initially processed in the FPGA and moved with DMA engines to the CPU and GPU components. A study of the potential of FPGAs and GPUs to accelerate data center applications is done in [17]. The paper confirms that FPGA and GPU platforms can provide compelling energy efficiency gains over general purpose processors, but it also indicates that the possible advantages of FPGAs over GPUs are unclear due to the similar performance per watt and the significant programming effort of FPGAs. In any case, it is important to note that the

paper does not use high level languages to increase FPGA productivity as done in our work, and the power measurements for the FPGA are based on worst case tool estimations and not direct measurements.

In our research, we explore a horizontal (i.e. exploit data parallelism) collaborative solution more closely related to the work done in [18]. This previous research also focuses on a multiple device solution similar to our work and demonstrates how the N-Body algorithm can be implemented in a heterogeneous solution in which both FPGA and GPU work together to compute the same algorithm kernel on different sets of particles. While our approach uses an adaptive scheduling algorithm to compute the optimal split, in the previous work the split is calculated manually with ⅔ of the workload to FPGA and the remaining ⅓ to GPU.

In summary, we can conclude that the available literature has largely focused on exploring how to use the FPGA on heterogeneous systems as well as comparing the performance of GPUs, FPGAs and CPUs for different types of applications in large scale clusters. These approaches require the programmer to manually choose the optimal device for each part of the application and move data among them. In contrast, our proposal aims at providing an easier programming model that strives to hide most of the hardware details and OpenCL nuances when programming parallel loops for CPU+FPGA architectures. The user still has to provide the FPGA kernel, but the TBB based run-time takes care of evenly partitioning the iteration space and making the data accessible to both the CPU and FPGA. To do so, we select a state-of-the-art high-level scheduler called LogFit [19, 7] that was recently developed for CPU+GPU chips and we extend it to support simultaneous computing on CPU+FPGA. LogFit has been also used for Xilinx chips composed of ARM cores and FPGA [20], but in that case, instead of OpenCL, SDSoC and C were used to generate the FPGA computing units for regular applications. In this work we extend the analysis to irregular parallel loops that can be found in applications such as BarnesHut and SPMV, that were not considered in the SDSoC case. By irregular parallel loops we refer here to loops that exhibit different computational load per iteration and that may access non-coalesced data of an irregular data structure. Moreover, the main contribution in [20] was the implementation of an interrupt mechanism to prevent one of the ARM cores to waste time busy waiting for the FPGA finalization. In our work here, we target a Cyclone V chip with an integrated Altera FPGA that can be programmed in OpenCL. With this testbed, the interruption mechanism is already provided by the OpenCL driver and irregular loops can be efficiently implemented on the FPGA. Additionally, for a more efficient implementation of the wide range of applications that we cover in this paper, some modifications have been required in the LogFit scheduler, as we explain in more detail in the next section, so that this represents a novel contribution of this work.

3. Heterogeneous Building Blocks framework

This section introduces the Heterogeneous Building Blocks (**HBB**) library API and scheduler. It is a C++ template library that takes advantage of heterogeneous processors and facilitates its usage and configuration. HBB aims to make easier the programming for heterogeneous processors by automatically partitioning and scheduling the workload among the cores and the accelerator. The current version offers a `parallel_for()` function template, originally devised to exploit CPU+GPU platforms [7], which we extended to run on heterogeneous systems comprising CPU cores and OpenCL capable FPGAs.

HBB offers an abstraction 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 her own application instead of dealing with thread management and synchronization. The engine managing the *parallel_for* pattern is implemented with a two-stage pipeline TBB template, that we explain in Subsection 3.3.

3.1. User interface: parallel_for

Figure 1 shows a `main` function with all the required component initialization to make the `parallel_for()` function template work. This is the main component of the `HBB` library and it is made available by including the `hbb.h` header file.

The `parallel_for()` function template receives three parameters (line 10): the first two parameters, `begin` and `end` describe the parallel iteration space. The third parameter is the `Body` instance that should

```
1  #include "hbb.h"
2
3  int main(int argc, char* argv[]){
4    Body body;
5    Params p;
6    InitParams (argc, argv, &p);
7    // Instantiate task scheduler
8    auto *hs = {Static, Dynamic, HAP}::getInstance(&p);
9    ...
10   hs->parallel_for(begin, end, body);
11   ...
12 }
```

Figure 1: Using the `parallel_for()` function template.

include the user implementation of the CPU and FPGA loop body (this is, a function that defines what to compute on the CPU and a similar function for the FPGA). More details in the next Subsection. Program arguments can be read from the command-line, as can be seen in line 6. The benchmarks that we evaluate in Section 6 accept at least three command-line arguments: `<num_cpu_cores>`, `<num_fpgas>` and `<sch_arg>`. The last argument meaning depends on the particular implementation of the partitioning strategy of the heterogeneous scheduler selected. Currently, our `HBB` library provides three heterogeneous schedulers, each one with a different partitioning strategy: Static, Dynamic and HAP. They can be instantiated as shown in Figure 1 at line 8 and will be briefly discussed in Section 4.

3.2. Programming Interface implementation details

```
1  class Body{
2
3  public:
4    void operatorCPU(int begin, int end) {
5       for(i=begin; i!=end; i++){
6          ... }                                    // body statements in C++
7    }
8
9    void operatorFPGA() (int begin, int end){
10      clEnqueueUnmapMemObject(...);               // Zero-copy Host-to-device}
11      setKernelArgument(...);                     // OpenCL arguments
12      clEnqueue{NDRangeKernel|Task}(begin, end); // Launch OpenCL kernel}
13      ... = clEnqueueMapBuffer(...);              // Zero-copy Device-to-Host}
14   }
15 };
16 ...
```

Figure 2: Details of Class Body.

The user must implement a `Body` class in order to define the body of the parallel loop, as we see in Figure 2. This class must implement two methods: one that defines the code that each CPU core has to execute for an arbitrary chunk of iterations, and the same for the FPGA device using OpenCL. The `operatorCPU()` method (lines 4-7 in Figure 2) defines the CPU code of the kernel in C++, and the `operatorFPGA()` method (lines 9-14) comprises the usual OpenCL steps required to offload work to the FPGA, namely: i) host-to-device operations to transfer the array regions that will be processed on the FPGA; ii) kernel arguments specification; iii) kernel launching to dispatch the computation of a chunk of iterations on the FPGA; and iv) device-to-host operations to retrieve the results of such computation. If the arrays are allocated on a memory region shared by the CPU and the FPGA, steps i) and iv) should be substituted by the corresponding `map` and `unmap` OpenCL operations [21] following zero-copy semantics. In Figure 2 we rely on `clEnqueueUnmapObject` and `clEnqueueMapBuffer` functions to this end. The latter returns a pointer that is valid on the CPU (the CPU view of the shared buffer) and informs the OpenCL runtime that the CPU will access the corresponding region of the buffer. The complementary unmap function is on the other hand stating that the FPGA becomes the owner of a particular region of the buffer that can be

now read and written by the FPGA. Note that the user is responsible of having a compiled version of the OpenCL kernel for the FPGA at hand. In other words, for the `operatorFPGA()` method the user has to: 1) write and optimize the OpenCL code for the FPGA; and 2) compile the kernel offline by using the Intel/Altera `aoc` compiler. The HBB library takes care and hides all the OpenCL boilerplate (kernel loading, context and command queue initialization, etc.).

3.3. Scheduler implementation

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

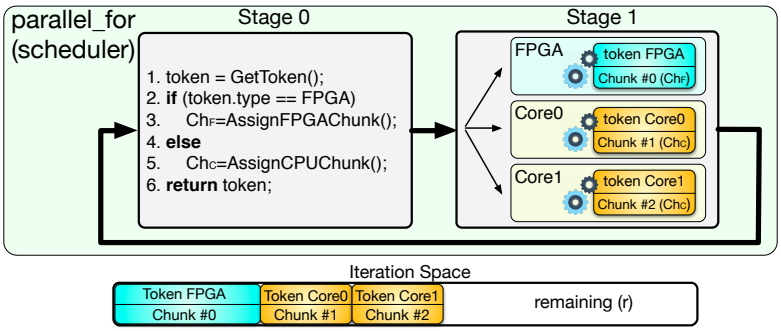


Figure 3: Scheme of the scheduler implementation. Stage 0 performs the partitioning and scheduling, while Stage 1 computes the assigned chunks.

Figure 3 depicts a simplified scheme of how the scheduler for our `parallel_for()` has been built. The engine is implemented with a two-stage pipeline TBB template with Stage 0 and Stage 1. Each token corresponds to a compute unit class, FPGA or CPU core in our system, and the sum of tokens represents the number of chunks of the iteration space that will be processed simultaneously. In the figure, we have two CPU tokens and one FPGA token being processed concurrently, two on the CPU cores and one on the FPGA. The first stage, Stage 0, gets an available token (line 1) and adaptively selects the chunk size for FPGA and CPU tokens, and extracts the corresponding chunk of iterations (Ch_F and Ch_C) from the set of remaining iterations, r (lines 2-6). The procedure to select the FPGA and CPU chunk sizes is covered in Section 4. Then, when a token reaches the second stage, Stage 1, the chunk gets processed on the corresponding device. The time required for the computation of the chunk on the FPGA and on a CPU core is recorded. This time[2] is used to update the relative speed of the FPGA w.r.t. a CPU core, that we call f. Factor f will be required to adaptively adjust the size of the next chunk assigned to a CPU core. Tokens are recycled until there are no remaining iterations. Stage 0 is a serial stage, so it can host just a token at a time, whereas Stage 1 is parallel, so several tokens can be working in this stage simultaneously.

4. Partitioning strategies

In the Static scheduler the partition of the iteration space is statically computed at the beginning of the execution according to the user-defined `offload_ratio`, *olr*, initialized from the command-line argument (`<sch_arg>`=*olr*). *olr* is a float value in the range [0.0, 1.0] representing the ratio of iterations offloaded to the FPGA. If $olr = 0$ the FPGA is not used. The result of this block partitioning may result in load unbalance if the *olr* ratio is not fine tuned. Moreover, for irregular codes in which the `parallel_for()` is invoked several times, there is usually an optimal *olr* for each one of the invocations, so the Static scheduler results

[2]For the FPGA, the registered computation time includes the data transfer -or map/unmap- times

in load unbalance if a single *olr* is provided. On the other hand, this scheduler minimizes the partitioning overhead (Stage 0 and Stage 1 are executed only once for each compute device).

In the Dynamic scheduler the FPGA chunk size of parallel iterations is manually set by the user as a positive integer value (<sch_arg>=Ch_F), whereas the CPU chunk size, Ch_C, is automatically computed by a heuristic. This heuristic aims to adaptively set the chunk size for the CPU cores by taking into account the factor f introduced in the previous subsection. This factor represents how faster is the FPGA w.r.t. a CPU core so we call it the *relative speed*. The ratio f is recomputed each time a chunk is processed either on a CPU core or the FPGA. If there are enough remaining iterations, r, the CPU chunk size is computed as $Ch_C = (Ch_F/f)$ (the number of iterations that a CPU core must perform to consume the same time as the FPGA), or $Ch_C = r/(f + nCores)$ (a *guided self-scheduling strategy* [22]), when there are few remaining iterations, this is when $r/(f + nCores) < Ch_F/f$. A downside of this scheduler flavor is that this scheduler assumes that a single FPGA chunk size i) is provided by the user; and ii) results in good performance, which might not be the case, specially when targeting irregular applications.

To avoid these Dynamic scheduler limitations, HAP (Heterogeneous Adaptive Partitioner) dynamically computes both the FPGA and CPU chunk sizes at run-time. The scheduler is based on the LogFit scheduler [7] and is designed as a three-phase strategy consisting of: the Exploration Phase (EP), the Stable Phase (SP) and the Final Phase (FP). Algorithms 1 and 2 show the pseudocode for functions `AssignFPGAChunk()` and `AssignCPUChunk()` that are responsible for selecting the FPGA and CPU chunks (Ch_F / Ch_C) to be assigned to the FPGA or CPU device, respectively (see lines 3 and 5 of Stage 0 in Figure 3).

```
1  Function AssignFPGAChunk()
2      switch Phase do
3          case ExplorationPhase do      /* Exploration */
4              if StableCondition then
5                  Phase=StablePhase;
6                  Ch_F=ComputeFitting();
7              else
8                  Ch_F=Ch_F * 2;
9              end
10             break;
11         case StablePhase do                /* Stable */
12             if StopCondition then
13                 Phase=FinalPhase;
14             else
15                 Ch_F=ComputeFitting();
16                 break;
17             end
18         case FinalPhase do                  /* Final */
19             Ch_F=ComputeLastFPGAChunk();
20         end
21     end
22     r=r-Ch_F;
23     return Ch_F;
```

Algorithm 1: Assign FPGA chunk function in HAP.

```
1  Function AssignCPUChunk()
2      if StopCondition then
3          Ch_C=ComputeLastCPUChunk();
4      else
5          Ch_C=Ch_F/f;
6      end
7      r=r-Ch_C;
8      return Ch_C;
```

Algorithm 2: Assign CPU chunk function in HAP.

In the Exploration Phase, EP (lines 3-10 in Algorithm 1), we look for an initial FPGA chunk size (Ch_F) that maximizes the throughput in the device. To that end, we carry out an exploration in which we keep increasing the size of the FPGA chunks at the beginning of the iteration space (line 8), and sample its throughput. The algorithm stays at this phase until the explored size is bigger enough and the throughput stabilizes (*StableCondition* becomes true, line 4). At this point, the samples are used to heuristically compute an appropriate FPGA chunk size using a logarithmic fitting of the measured throughputs (line 6), as we describe in [7]. The selected FPGA chunk ensures a near-optimal throughput at this point. Meanwhile, during this phase, for each sampled FPGA chunk size, we compute a CPU chunk size (Ch_C) that balances

the load for both devices (line 5 in Algorithm 2). Note that we use here the previously mentioned factor f.

Then, in the Stable Phase, SP (lines 11-17 in Algorithm 1), any time that a new FPGA chunk must be selected, we perform again the logarithmic fitting of the measured throughputs (line 15), which enables re-adjusting the FPGA chunk size to cope with the application irregularities (see [7] for details). Again, CPU chunk sizes are also dynamically re-computed for each CPU chunk (line 5 in Algorithm 2).

The Final Phase, FP (lines 18-20 in Algorithm 1), is activated when there are few remaining iterations (*StopCondition* is true in line 12) and we have to pay particular attention to find out the best possible partitioning of the remaining iterations, so function `ComputeLastFPGAChunk()` is invoked (line 19). The goal is to minimize the time to compute the remaining iterations by finding $T_{min} = min(T_{CPU}, T_{FPGA}, T_{HET})$ where T_{CPU} and T_{FPGA} estimate the time required to execute those iterations only on the CPU or the FPGA, respectively, whereas, T_{HET} (heterogeneous time) is evaluated assuming that we can partition the remaining iterations between the CPU and FPGA while ensuring that they finish at the same time. Similarly, when *StopCondition* is true in line 2 in Algorithm 2, function `ComputeLastCPUChunk()` is invoked (line 3) following the same logic. In both cases, a last FPGA/CPU chunk (that could have 0 iterations) is computed. Before returning the corresponding chunk, the selected new chunk is removed from the remaining iterations (lines 22 and 7 in Algorithms 1 and 2, respectively).

In all these phases, we approximate (estimate) the FPGA throughput as a logarithmic function of the FPGA chunk size as done by Belviranli et al. [8]. However, in Belviranli's work, after the Exploration Phase, EP, a guided-self scheduling is used to keep feeding the accelerator and the CPU until the end of the iteration space. We found that this solution is sub-optimal, especially for irregular applications. In our proposal, we keep re-computing the FPGA chunk sizes applying logarithmic fittings to continuously optimize the accelerator throughput, and we add a Final Phase that strives to find the assignment of the last iterations to the corresponding device(s) so that time is minimized.

As mentioned, more details about the Exploration, Stable and Final phases can be found in [7], where we introduce LogFit. However this previous scheduler, LogFit, targets CPU+GPU chips. In the HAP scheduler that we present in this work, we have extended the Exploration Phase significantly to adapt to some particularities of an OpenCL-capable FPGA:

- In LogFit, the exploration starts with an initial chunk size nEU, which is the number of Execution Units on the target GPU[3]. Now in HAP, we start searching for smaller chunk sizes that were not appropriate for GPUs, where at least all the Execution Units must get one iteration. For CPU+FPGA architectures, using smaller initial values resulted in finding better FPGA chunk sizes at the end of the exploration. In Section 6.3 we discuss the pros and cons of searching smaller chunk sizes during the EP and present the new initial chunk size that works well for our benchmarks.

- Since the EP explores now smaller chunk sizes in HAP, the execution time for these chunks is less reliable, which leads us to also modify the stabilization criteria. In the new scheduler, we keep increasing the FPGA chunk size, and sampling the corresponding throughput, until there are 3 samples for which the throughput does not increase more than a given threshold value, θ, that in our experiments is 1% of the previously measured maximum throughput. When this criteria is met, then we have reached the *StableCondition* and we switch to the SP.

- Another optimization in HAP is that now we ensure that the samples used to compute the first logarithmic fitting increase monotonically. That is, if a throughput sample is smaller than the previous one, the sampling procedure restarts, now using the corresponding FPGA chunk size as the first one. At least four monotonically increasing samples have to be collected to compute the first logarithmic function that approximates the FPGA throughput of the Exploration Phase.

[3] nEU = clGetDeviceInfo(deviceId, CL_DEVICE_MAX_COMPUTE_UNITS).

5. OpenCL optimizations for the FPGA

In this section, we first describe the benchmarks and later discuss the different optimizations that we apply in order to get the best FPGA implementations using the OpenCL API.

5.1. Benchmarks

This subsection describes our benchmark collection. This collection comprises both integer and floating point applications from different domains: linear algebra (Sparse Matrix Vector Multiplication-SPMV), cryptography (AES), gravitational dynamics (NBody-NB and BarnesHut-BH), physics simulation (HotSpot-HS), 3D graphics (Bezier-BZ), and risk management (Black Scholes-BS). All our applications are memory bound, which represents a challenge for heterogeneous scheduling.

5.1.1. NBody, NB

We study two different implementations of an N-body problem: a regular approach called Nbody [23] and an irregular one called BarnesHut. These two benchmarks consist of a sequential outer loop that represents a sequence of time-steps. In each time-step all the body-body gravitational interactions are computed. To that end, each time-step contains a parallel loop that traverses all the bodies and an inner loop that, in the NBody code, for each body of the parallel loop, computes the interactions with all other bodies. Nbody is considered a brute force method that computes all the interactions that the n bodies in the system induce upon each other. Thus the inner loop always computes n iterations. For our study an input set of 100,000 bodies and 15 time-steps were simulated.

5.1.2. BarnesHut, BH

BarnesHut is a particular implementation of a N-body problem, which employs an octree data structure [24]. The number of interactions of each body varies depending on its location, so in the inner loop of BarnesHut, depending on the iteration, some bodies are approximated by a single "virtual body" with the aggregated mass and located at their center of mass. Thus, for each parallel loop iteration, the inner loop has a different number of iterations (or interactions). This results in a parallel loop with variable time per iteration, not only because the parallel iterations exhibit different computational load, but also because the access to the octree data structure causes non-coalesced and fickle data references. An input set of 100,000 bodies and 15 time-steps were simulated.

5.1.3. Sparse Matrix-Vector Multiplication, SPMV

SPMV is a sparse matrix-dense vector multiplication from Bell and Garland work [25], also representing an irregular application since each row has a different number of non-zeros, which are stored using a Compressed Sparse Representation, CSR, data structure. The GL7d13 sparse matrix from the University of Florida Sparse Matrix Collection that exhibits a triangular-like profile was selected as input.

5.1.4. AES

This benchmarks comes from the Hetero-Mark suite [26]. It implements the Advanced Encryption Standard (AES) algorithm. The program takes plain text as input and encrypts it using a given encryption key. In our experiments we took an input text of 20 Mbytes and a key of 256 bits.

5.1.5. Bezier, BZ

This benchmark computes a 3D Bezier surface from a normal surface. The original algorithm is taken from the Chai benchmark [27] and comprises 4 nested loops. The two most outer loops traverse the output matrix that is filled after traversing all elements from the two most inner loops, representing the input image. For this benchmark, the size of the input was 4 by 4, and the output resolution was 300 by 300.

5.1.6. Black-Scholes, BS

This benchmark consists of an Asian Option Pricing Algorithm based on the one provided by Intel-Altera in [28]. The computation in this kernel is divided in 3 stages. First stage uses the Mersenne Twister random number generator in order to feed the second stage. The second stage uses the geometric brownian motion as described by the black scholes model. The last stage sums each of the results of the previous stage to produce a final reduction an produce the total payoff of the option. We performed 10,000 simulations for each experiment.

5.1.7. HotSpot, HS

HotSpot [29] is a well known application used to estimate processor temperature based on an architectural floorplan and simulated power measurements. The thermal simulation iteratively solves a series of differential equations. Each output cell in the computational grid represents the average temperature value of the corresponding area of the chip. In these experiments an input of 2048x2048 points was considered.

Both BarnesHut and SPMV can be considered irregular benchmarks, whereas NBody, HotSpot, AES, Black-Scholes and Bezier are representative of regular applications.

5.2. Platform specific OpenCL optimizations

This subsection describes the main optimizations applied to our benchmark collection. Optimizing OpenCL code for FPGA requires using a plethora of techniques that have to be tailored per application [21, 30, 31]. Our benchmarks have not been an exception as Table 1 shows. No single technique has been effective for all of them. Having said that, for each benchmark, the best combination of techniques always improves the efficiency of the memory transfers.

Table 1 summarizes the most commonly applied optimizations. Task (first column) refers to single task kernels (a.k.a. single work-item kernels), which is equivalent to launching an OpenCL kernel with an NDRange size of (1, 1, 1)[4]. Intel-Altera [21] suggests to structure kernels this way, if possible. This optimization exploits loop pipelining allowing the overlapping of data transfers and computations between loop iterations, therefore improving the throughput of the FPGA implemented pipeline. In our study, almost half of the benchmarks have benefited from single task kernels. Compute Unit Replication clones the kernel pipeline when there is enough availability of resources in the FPGA for improving throughput as well. However multiple compute units competing for memory bandwidth might lead to undesired memory access patterns that could degrade performance, so it must be used carefully. In our set of benchmarks, Compute Unit Replication is only beneficial for 2 of them. Vectorization (fourth column) packs multiple scalar work-items into a single vector (SIMD) work-item, and alongside Loop Unrolling of inner loops, both allow increasing the number of parallel operations as well as coalescing memory accesses, effectively improving the throughput. Both Vectorization, Loop Unrolling and Compute Unit Replication optimizations are limited by the availability of FPGA resources (logic, DSPs, etc). Locality (fifth column) encompasses techniques aiming for maximizing data reuse. For example, Register Privatization (RP) forces the compiler to keep alive the scalar variable in register rather than in memory to avoid the usual pattern of load to memory, operation, and store to memory within loops. Constant and Local Memory (CM, LM) reduce global memory accesses. Shift Register Pattern (SRP) and Channels (AC) are specific optimizations for FPGAs. SRP reduces the number of global memory accesses by keeping previously accessed data cached in a shift register. AC provides a fast communication mechanism to pipe data among parallel kernels by eliminating some of the global memory accesses.

On our target platform (DE1, described in the next section), it is possible to exploit the OpenCL1.2 zero-copy buffer optimization since the CPU and the FPGA can share the same memory region[5]. However, due to current driver limitations, CPU caching is disabled for the shared memory region, which severely hits

[4] In the OpenCL standard, the NDRange represents the 3D space of parallel iterations. Using (1,1,1) means that a single thread is invoked on the accelerator.

[5] This is achieved by allocating the region with clCreateBuffer(..., CL_MEM_ALLOC_HOST_PTR, size, ...) and then mapping this region to a CPU accessible pointer using clEnqueueMapBuffer().

	Task	Compute Unit Replication	Loop Unrolling	Vectorization	Locality
BH	✗	1	0	1	RP
SPMV	✗	1	4	2	RP
BZ	✓	1	0	1	LM, CM
AES	✗	2	0	1	CM, LM, SRP
NB	✗	1	2	1	RP
BS	✓	1	0	1	LM, AC
HS	✓	3	0	2	SRP

Table 1: Summary of applied optimizations per benchmark. Legend: RP-Register Privatization, CM-Constant Memory, LM-Local Memory, SRP-Shift-Register Pattern, AC-Altera Channels.

CPU performance. Therefore, to also exploit the CPU cores we do not use shared memory. Instead, we rely on two possible alternatives. For coarse-grained benchmarks for which the cost of offloading the data is a small fraction of its computation time, the traditional OpenCL host-to-device and device-to-host operations are used (BH, BZ, AES, NB, BS). Otherwise (SPMV, HS), we allocate two different memory regions: one explicitly allocated and aligned for the CPU, and a different buffer on physically contiguous memory that is efficiently accessed by the FPGA via DMAs in the DDR memory controller. After the `parallel_for()` execution, an extra overhead is paid to merge the final output results produced by the CPU and the FPGA. As soon as Altera OpenCL driver provides efficient shared memory between the CPU and the FPGA, we will save the currently implemented data movement overheads, so better performance and energy efficiency figures can be expected.

	Logic (%)	Reg. (%)	Mem. (%)	DSP (%)	CLK (MHz)
BH	72	70	63	18	120
SPMV	23	21	21	1	145
AES	70	79	16	0	103
BZ	80	98	25	72	120
NB	65	64	36	29	143
BS	60	62	6	94	139
HS	29	25	10	32	144

Table 2: AOC Compiler Resource Usage Summary per category (% of total resources) compiling for Altera DE1-SoC.

Table 2 summarizes the resource usage for all benchmarks under test and the resultant FPGA frequency in MHz. The variability is large in both categories and benchmarks because resource utilization mainly depends on kernel organization and arithmetic intensity. For example, DSP utilization ranges between 94% and 1% for BS and SPMV; although both of them only produce floating point results. On top of that, there is neither correlation between resource usage and performance nor between resource usage and energy consumption, as shown in the sequel.

6. Experimental results

In this section, we present a comparative study of performance and energy results of our proposals. Firstly, in Subsection 6.1, we present the platform and software framework. After that, in Subsection 6.2 we show performance results for the Static, Dynamic and HAP schedulers, while in Subsection 6.3, we study in more detail the behavior of the HAP scheduler: we report its overhead, the FPGA chunk size evolution, and discuss how we tuned the initial FPGA chunk size for the Exploration Phase. Finally, in Subsection 6.4 we analyze the energy efficiency of HAP.

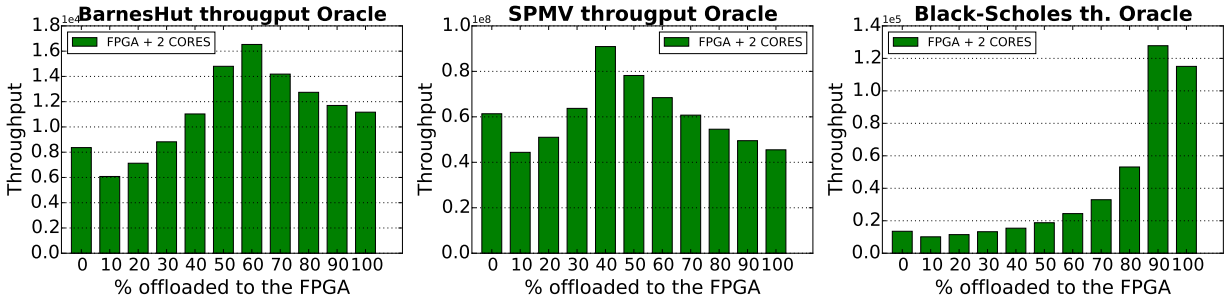


Figure 4: Throughput of benchmarks when invoking the Static scheduler for different ratio of iterations offloaded to the FPGA on an FPGA+2CORE configuration. The higher the better. Throughput units are the same as in Figure 5.

6.1. Experimental Settings

We run our experiments on the Terasic DE1-SoC board that we briefly describe here. This board features the Cyclone V SoC 5CSEMA5F31C6 chip that comprises two ARM Cortex-A9 at 925MHz and an FPGA with 85K Programmable Logic Elements, 64,140 registers, 4,450 Kbits of embedded memory and 87 DSPs (in Table 2 we refer to these resources as Logic, Reg., Mem. and DSP). We use the *de1soc_sharedonly* configuration in which the 1GB DDR3 SDRAM is accessible by both the CPU and the FPGA. The CPU has 32 KB L1 instruction cache, 32 KB L1 data cache and 512K shared L2. The board runs Linaro 12.11, and the CPU codes have been compiled by gcc 4.6 with -O3. The FPGA runs kernels compiled with Altera OpenCL Compiler 14.1, supporting OpenCL 1.2.

In order to measure the energy consumption in this platform, we rely on the Power Cape board (derived from an infrastructure proposed in [32]) that is connected to a BeagleBone Black (BBB) board. The Power Cape contains several TI INA291 current/power monitors that are connected to the BBB via I2C bus. The BBB runs Ubuntu and the server part of the pmlib library [33]. This pmlib server is a thread that samples the INA291 at 100Hz. Finally, the DE1-SoC is connected by Ethernet to the server and the benchmarks running on the DE1-SoC are instrumented with pmlib client calls (start_counter, stop_counter, etc) so that power readouts are available from the benchmark. Note that the thread reading the INA291 power values is running on the BBB. That way, the 2 cortex-A9 of the DE1-SoC are not interfered due to power measurements, since the heavy work is done on the sibling BBB board.

6.2. Exploring the chunk size on each scheduler

We used the Static scheduler to explore the optimal assignment of iterations between the FPGA and the CPU, which ensures maximum throughput. Our goal was to compare the Dynamic and HAP schedulers against an Oracle-like approximation. Figure 4 shows the throughput for three benchmarks that summarize the different scenarios that we found during our study. The throughput is measured as elements processed per second, thus the higher the better. In particular, the throughput represents bodies/s, non-zeros/s and simulations/s for BarnesHut, SPMV and Black-Scholes, respectively. In the figure we show the results for the configuration for which we got the higher throughputs: FPGA+2CORE (i.e. FPGA and 2 CPU cores). The X-axes represent the % of computations offloaded to the FPGA. In other words, the *orl* parameter that we explore from 0 (0%) to 1 (100%), taking steps of 10%.

As we see in Figure 4, the optimal throughput can be found from 40% of iterations offloaded to the FPGA (e.g. SPMV) up to 90% (e.g. Black-Scholes). In any case, we compared this optimal throughput against the best throughput obtained for the Dynamic and HAP schedulers shown in Figure 5. We found that for regular benchmarks, Static outperforms Dynamic and HAP by 4% and 6% at most, respectively, while for irregular benchmarks like BarnesHut, Static degrades throughput by 36% and 34% when compared to Dynamic and HAP, respectively. As pointed out in Subsection 3.3 load unbalance due to a single chunk assignment to both devices explains this behavior. As anticipated here, our Dynamic and HAP schedulers introduce low overhead for regular benchmarks and even outperform the performance for irregular ones. We explore these dynamic strategies in more detail next.

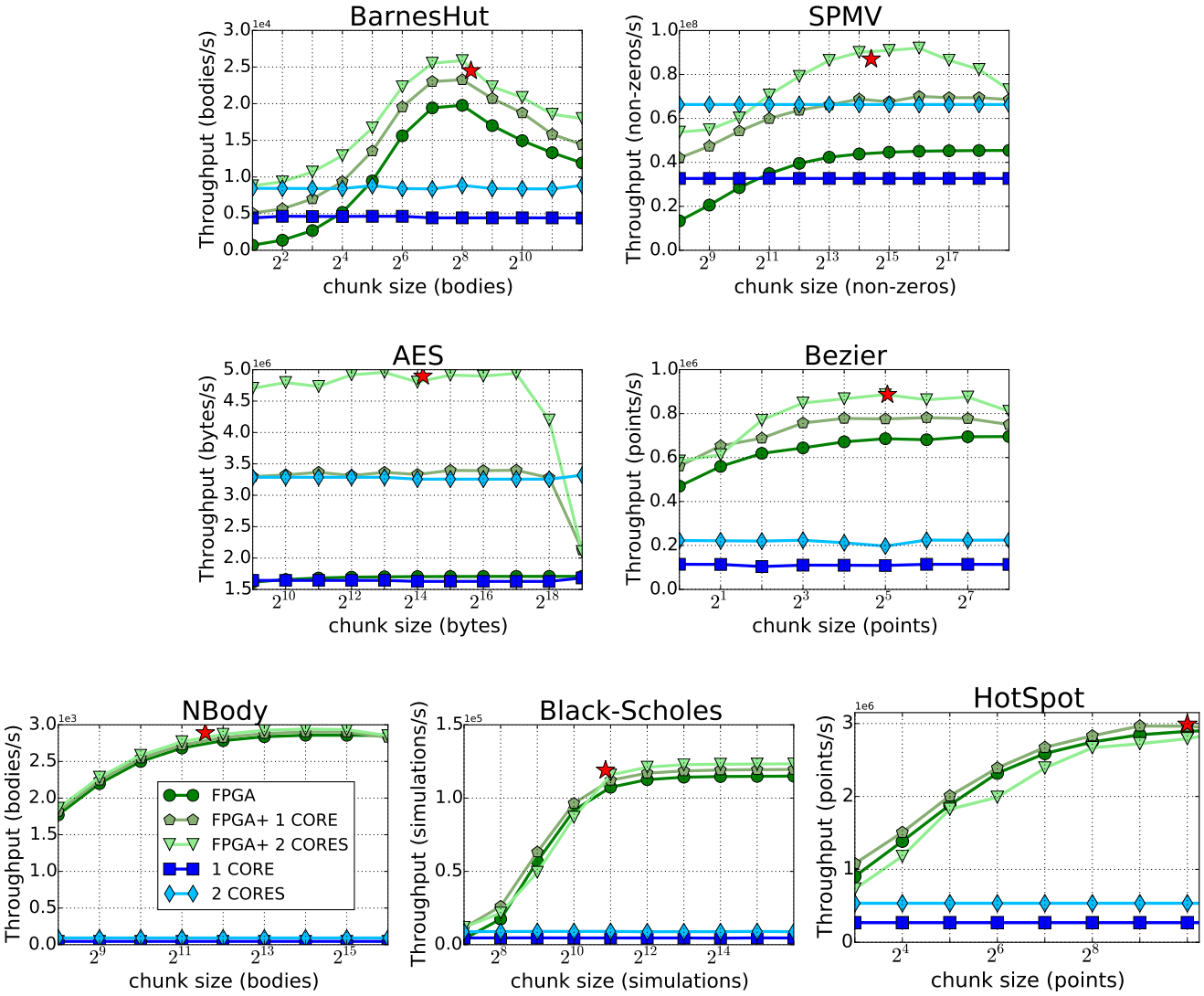


Figure 5: Throughput of the benchmarks for different chunk-size when invoking the Dynamic and HAP schedulers. Each line represents a system configuration: 1CORE, 2CORE and FPGA (one-device configurations), plus FPGA+1CORE and FPGA+2CORE (two-device configurations). The higher the better.

Figure 5 shows a study about how the size of the chunk of parallel iterations assigned to each device affects the throughput of the execution when using our dynamic schedulers. For the one-device configurations (1CORE, 2CORES or only FPGA) we apply a classic dynamic scheduling in which we fix the size of the chunk on each experiment. However for the two-device configurations (FPGA+1CORE or FPGA+2CORES) we apply the heterogeneous Dynamic and HAP schedulers presented in Subsection 3.3. For the Dynamic scheduler the size of the chunk offloaded to the FPGA is fixed and shown on the X-axis, while the chunk assigned to a CPU core is computed adaptively by the scheduler. For the HAP, considering that the FPGA chunk size is continuously readapted, we only show one point, depicted by a red five-point star (★), which coordinates are (<average FPGA chunk size>, <average throughput>) for the FPGA+2CORES configuration. This is, instead of showing the range of chunk sizes used throughout the execution (something that we illustrate in Subsection 6.3), the X-axis value of the ★ represents the average of all the computed FPGA chunk sizes.

In general, two-device executions always outperform those with one device, even in memory bound applications as the ones we explore in this work. Moreover, adding CPU cores to collaborate with the FPGA usually increases throughput. For irregular benchmarks like BarnesHut and SPMV, and regular benchmarks like AES and Bezier, the FPGA is between 1x to 5x faster than 1CORE executions. Thus we can say that these benchmarks map relatively well on the FPGA. In these cases, we measure from 1.3x to 2.9x throughput improvement for the two-device executions compared to the only FPGA executions (1.4x to 3.9x compared to 2CORE ones). For other regular benchmarks like NBody, Black-Scholes, and HotSpot for which the FPGA is more than 10x faster than 1CORE executions, meaning that they map very well on the FPGA, the improvement of the two-device executions is marginal, as expected (but it goes from 5.4x to 32x compared to 2CORE executions).

Our results hint an interesting finding: while the throughput for the one-device CORE executions is kept constant, regardless of the chunk size, this is not true for the only FPGA or for the two-device executions. In these configurations the size of the chunk should be carefully selected. Small chunks may degrade performance due to underutilization of the deep pipeline FPGA implementations. On the other hand, very big chunks may create load unbalance in the two-device executions and this explains the throughput degradation when the FPGA chunk size, Ch_F, is close to the total number of iterations, n. For example in AES and SPMV, having $Ch_F = 2^{18}$ precludes the scheduler from adapting to the ideal partition (in these cases the ideal would be $orl \approx 33\%$ that is below the above mentioned chunk size). Moreover, in memory-bound irregular benchmarks, like BarnesHut, chunks beyond a certain size may exacerbate the memory bus contention due to the potentially exponential increment in the number of random memory accesses required, which will degrade performance even for the only FPGA execution. Another observation is that for the same program, the best chunk size for each configuration (only FPGA, FPGA+1CORE, and FPGA+2CORES) may differ. Thus, an adaptive scheduler able to find at runtime the right chunk size for each type of application and device configuration seems a must-have component for this kind of heterogeneous platforms, not just for two-device executions but also for only FPGA ones.

	BH	SPMV	BZ	AES	NB	BS	HS	Average
Throughtput	-2.6%	-6.0%	-0.2%	-1.3%	-1.4%	-5.8%	+7.0%	-1.6%
Energy	-2.6%	-5.8%	-0.2%	-1.6%	-1.6%	-5.9%	+5.3%	-1.8%

Table 3: Throughput and Energy improvement of HAP compared to the best Dynamic configuration. The higher the better.

In this regard, by looking at the red star, ★, we see how good HAP is automatically finding near-optimal chunk-sizes for the FPGA. Also, Table 3 reports, in detail, the efficiency of HAP with respect to the best Dynamic, both in terms of throughput and total energy (computed from total power, measured as explained in Subsection 6.1). On average, HAP results in 1.6% lower throughput and 1.8% more energy consumption than the solution that we have found by the off-line exhaustive searching of different FPGA chunk sizes with the Dynamic Scheduler. The worst case is SPMV where the average throughput for HAP degrades 6%, with a 5.8% increase in energy consumption. Here, the average FPGA chunk size found by HAP tends to be smaller than the optimal found with Dynamic. This effect causes HAP to deliver slightly lower throughput. On the other hand, HAP can behave better than Dynamic in some cases, and for instance HAP results in 7% better throughput and 5.3% more energy efficiency than Dynamic for HotSpot. In this case, some re-adjusting of the FPGA chunk size during the benchmark execution improves the performance achieved by HAP. We will discuss these two benchmarks in more detail in Subsection 6.3. In any case, on average, the performance achieved by HAP on-line is close to the best performance found by Dynamic off-line.

Theoretical analysis: To better understand how close to the ideal throughput the HAP scheduler results are, Table 4 shows the ratio of the average throughput measured in HAP against that ideal throughput for each code. This ideal is estimated as the aggregation of the maximum throughput measured on each device individually (FPGA, 2CORES). That aggregation is based on the assumption that both devices can work concurrently without exhausting the memory bandwidth. The efficiency of HAP is higher for benchmarks with higher arithmetic intensity (AI): it goes from 99% of the ideal for a coarse-grained regular benchmark as NBody (with the highest AI, which is approximately 0.83FLOP/Byte) to 78% of the ideal for a fine-grained

irregular benchmark as SPMV (with the lowest AI, which is 0.12FLOP/Byte, much lower than the threshold to reach the peak performance in this processor). As we see, the lower the AI the more contention in the memory bus for the heterogeneous execution, which explains why the achieved performance of HAP degrades with respect to the ideal.

BH	SPMV	BZ	AES	NB	BS	HS
87%	78%	94%	98%	99%	95%	88%

Table 4: Efficiency of HAP compared to the ideal throughput. The higher the better.

As a summary, we see that HAP finds near-optimal solutions saving the burden of manually tuning the FPGA chunk size at a reasonable cost. From now on, we focus our analysis on HAP. In the next subsection we explore the overheads associated to this scheduler in more detail.

6.3. Exploring HAP behavior

The cost of the HAP scheduler is mainly due to the Exploration Phase in which sub-optimal FPGA chunk sizes are being explored. Other sources of overheads due to logarithmic fitting and chunk size computation are negligible (below 0.001%).

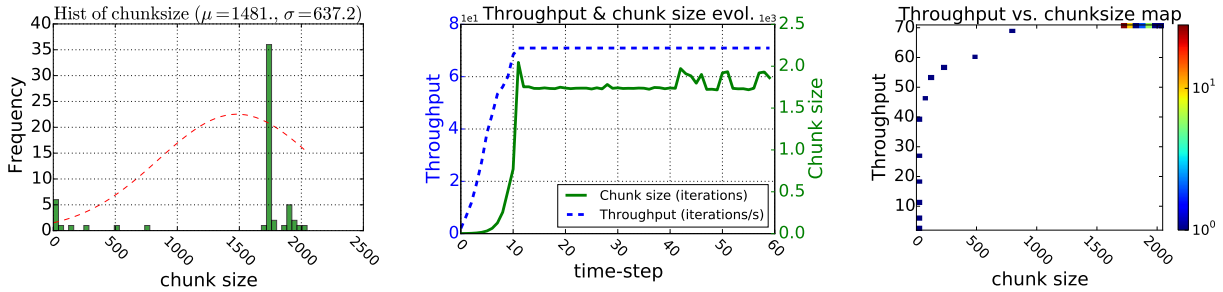


Figure 6: HAP FPGA chunk size exploration for HotSpot. Throughput units are parallel iterations per second.

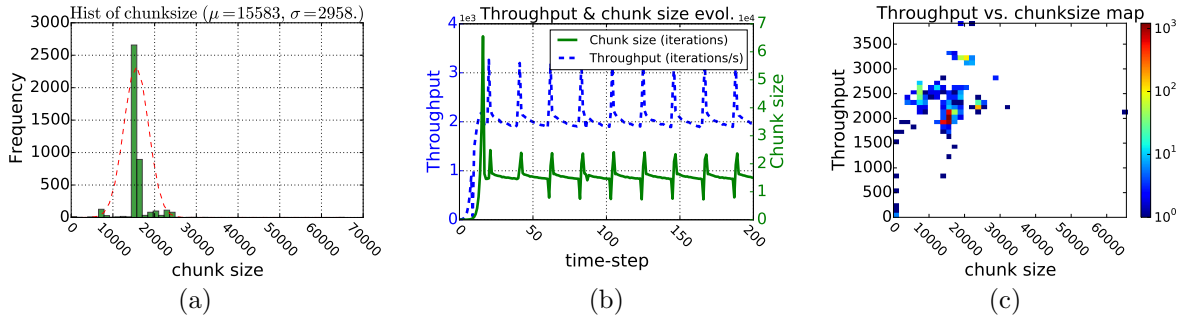


Figure 7: HAP FPGA chunk size exploration for SPMV. Throughput units are parallel iterations per second.

In Figures 6 and 7 we illustrate the FPGA chunk size evolution for two representative cases: HotSpot and SPMV for which HAP achieves the highest and lowest performance improvement over Dynamic (7% and -6%, respectively). The throughput we depict in these figures represents the FPGA's throughput. It is reported as parallel iterations per second, while the chunk size is the number of parallel iterations. In Figures 6a and 7a we can see the chunk size histogram and, in a dashed red line, the gaussian function for the average and standard deviation of the different chunk sizes. Interestingly, although HotSpot is a regular benchmark and the majority of the chunks get the 1,740 size, we see that HAP still tries bigger chunk sizes.

This is due to small throughput variations during the code execution to which our algorithm reacts selecting new chunk sizes, which in turn improves the average throughput of the application and pays off for the cost of the adaptation.

For SPMV the variability of the throughput is higher, so HAP tries bigger and smaller chunks when the application throughput increase and decrease, respectively, due to the change of density in the rows of the input sparse matrix. This is clearly shown in Figures 6b and 7b where we depict the evolution of the throughput (dashed blue line) and chunk size (solid green line) throughout the applications time-steps. The left side in these figures show the effect of the Exploration Phase.

Also, in Figures 6c and 7c we plot the throughput for each assigned chunk size using a color map to indicate the frequency of each pair (chunk size, throughput), where we see that the behavior of HAP for a regular application tends to be more stable than for a irregular one. We start the Exploration Phase with an initial FPGA chunk of size 1. For the regular HotSpot, the throughput can be clearly approximated by a logarithmic curve. For the irregular SPMV, more variability is observed, but also, logarithmic functions can fit the cloud of points. In any case, the map for this benchmark indicates that our HAP approach tries several fittings with bigger and smaller chunk sizes even during the Stable Phase, effectively adapting to throughput variations.

Starting with FPGA chunk size 1 implies that the number of chunks explored before moving on to the Stable Phase ranges between 8 (for Bezier) and 17 (for SPMV). On average, 13 chunk sizes are needed to exit the Exploration Phase, but considering that on average our benchmarks compute around 1000 FPGA chunks, this only represents around 1.3% of the available chunks.

Although the throughput in the Exploration Phase is farther from the optimal one, useful work is also accomplished while processing sub-optimal chunks. In other words, we can see the percentage of time consumed in the Exploration Phase (EP) as a pessimistic estimation of overhead due to the exploration. We have also evaluated other initial FPGA chunk sizes, considering initial sizes that go from 0.02% to 0.2% of iterations of the total iteration space of the parallel loop. For the evaluated benchmarks, we have found that using an initial size equal to 1 results in 4.6% of the time (on average) being consumed in the EP. Using initial size of 0.02% and 0.2% lead to similar percentage of time (on average) spent in the EP, but in several cases the fitting provides less optimal FPGA chunk sizes. This is because starting with initial bigger chunk sizes that already produce a high throughput leads to flatter curves when using the logarithmic fitting. These kind of curves tend to be less reactive to throughput variations and produce more stable chunk sizes during the Stable Phase, which in turn results in not exploring chunk sizes away from the one found in the Exploration Phase and therefore missing more optimized chunk sizes. Due to these reasons we ended up using initial FPGA chunk size of 1 as the winning trade-off, being small the cost of exploration.

6.4. Exploring Energy Efficiency

To fully understand the impact of heterogeneity on energy efficiency, we report two sets of results for our HAP approach. The first set only accounts for the *running power*, or power drained by the application running when excluding fixed or *idle power* [34]. The idle power accounts for the power drained when the devices are idle, i.e. only the OS is working. The second set of experiments analyzes total power, including both running and idle power. The distinction helps to understand the different trade-offs on two key scenarios: always-on and on/off turnable devices. For the former, always-on, applications are dispatched through recurrent requests, so the minimization of energy consumption due to running power is the main goal. For the later, on/off turnable devices, the goal is to minimize the energy consumption due to total power because devices can go off when there is no work to do.

To compute the running power, we measure the total board power as described in Subsection 6.1 while running the benchmark and subtract the idle power measured when the system is idle. The energy for each case is computed by integrating the corresponding instant power over time. In our experiments, we found that, on average, the idle power drained by the platform is 7.4 Watts, while the total power varies between 8 Watts (SPMV) and 8.7 Watts (NBody).

6.4.1. Running Power

Figure 8 represents the energy efficiency[6] when only the running power is considered for one-device and two-device configurations on each benchmark. Y and X-axes represent the energy efficiency and throughput, respectively.

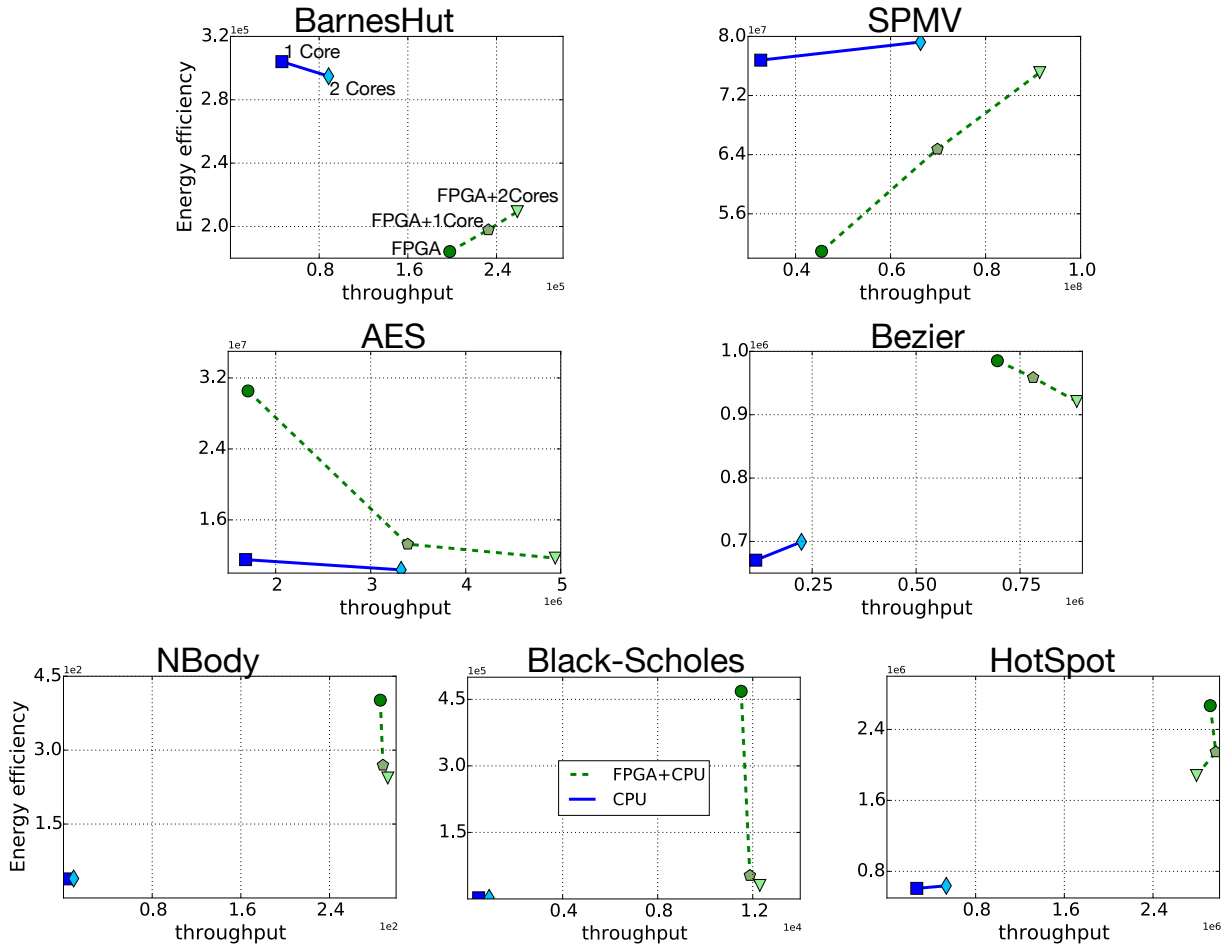


Figure 8: Energy efficiency-Performance plots for all benchmarks when HAP scheduler is invoked. Only the energy due to running power is accounted vs throughput. The higher the better in both dimensions. Throughput units are the same as in Figure 5. Energy efficiency is throughput per Watt.

When analyzing the energy consumption due to running power, we notice the following observations:

- For irregular benchmarks like BarnesHut and SPMV, the one-device CORE executions (1CORE or 2CORES) are more energy efficient than those including the FPGA even though the throughput for 1CORE or 2CORES might be worse than for FPGA-only, as we see for BarnesHut.

  For these benchmarks, all two-device executions are always more energy efficient and provides more throughput than FPGA-only execution. Energy efficiency goes from 1.2x (BarnesHut) to 1.5x (SPMV) when compared to FPGA-only.

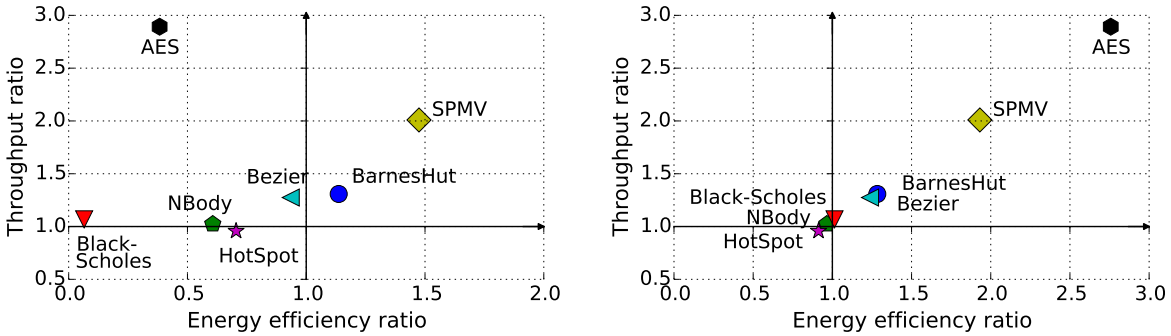
- For regular benchmarks that map relatively well on FPGA like AES and Bezier, we can see that executions that include the FPGA are always more energy efficient than those that only consider

[6]Energy efficiency is measured as iterations per Joule or throughput per Watt.

one-device CORE executions, although the throughput for FPGA-only execution might be worse than 2CORES as happens in AES.

Interestingly, for these benchmarks two-device executions are less energy efficient while providing more throughput than FPGA-only configuration, contrary to what happens for irregular benchmarks. Within this group, the energy efficiency degrades from 0.9x (Bezier) to 0.4x (AES) when compared to FPGA-only.

- For regular benchmarks that map very well on FPGA like NBody, Black-Scholes or HotSpot we observe that executions that include FPGA are always more energy efficient than those that only consider one-device CORE executions. For these benchmarks, two-device executions are less energy efficient and reduce throughput compared to FPGA-only configuration. The degradation ranges between 0.6x (NBody) and 0.1x (Black-Scholes).



(a) Throughput vs. Energy Eff. for running power

(b) Throughput vs. Total Energy Eff.

Figure 9: Throughput ratio (FPGA+2CORE to FPGA-only) vs Energy efficiency ratio (FPGA+2CORE to FPGA-only) for all the benchmarks when HAP scheduler is invoked.

Figure 9a plots the ratio of the throughput of the two-device FPGA+2CORE execution to the throughput of the FPGA-only execution versus the ratio of the energy efficiency of the 2-device FPGA+2CORE execution to the energy efficiency of the FPGA-only execution for all benchmarks. Again, results only account for running power. Benchmarks above the horizontal solid line are those whose two-device configuration improves the throughput of the FPGA-only configuration. Benchmarks on the right-side of the vertical solid line are those whose energy efficiency for the two-device configuration improves the energy efficiency of the FPGA-only execution. Basically, the figure summarizes what has been explained above. Irregular benchmarks like BarnesHut and SPMV improve throughput and energy efficiency when executing on 2-device configuration. Regular benchmarks as AES and Bezier improve throughput but degrade energy efficiency when compared to FPGA-only execution. Then NBody, HotSpot and Black-Scholes do not improve throughout neither energy efficiency when executed on 2-device configuration, therefore the FPGA-only execution represents the optimal choice for them.

6.4.2. Total Power

Figure 10 shows the energy efficiency when total power is considered. The x-axes represent the throughput.

The goal of these experiments, as stated before, is to explore situations in which we can turn off the devices once the application has been executed. Therefore we should select the device configuration that minimizes the total energy consumption. We see that there is a strong correlation between throughput and energy efficiency. The higher the throughput the better the energy efficiency. This is due to the significant contribution of the idle power that represents more than 80% of the total energy in any configuration. For instance, this idle power accounts for 86% - 82% in FPGA-only and FPGA+2CORES executions, respectively, in HotSpot, or even for 99% - 94% in AES. In any case, Figure 10 shows that FPGA+2CORES executions

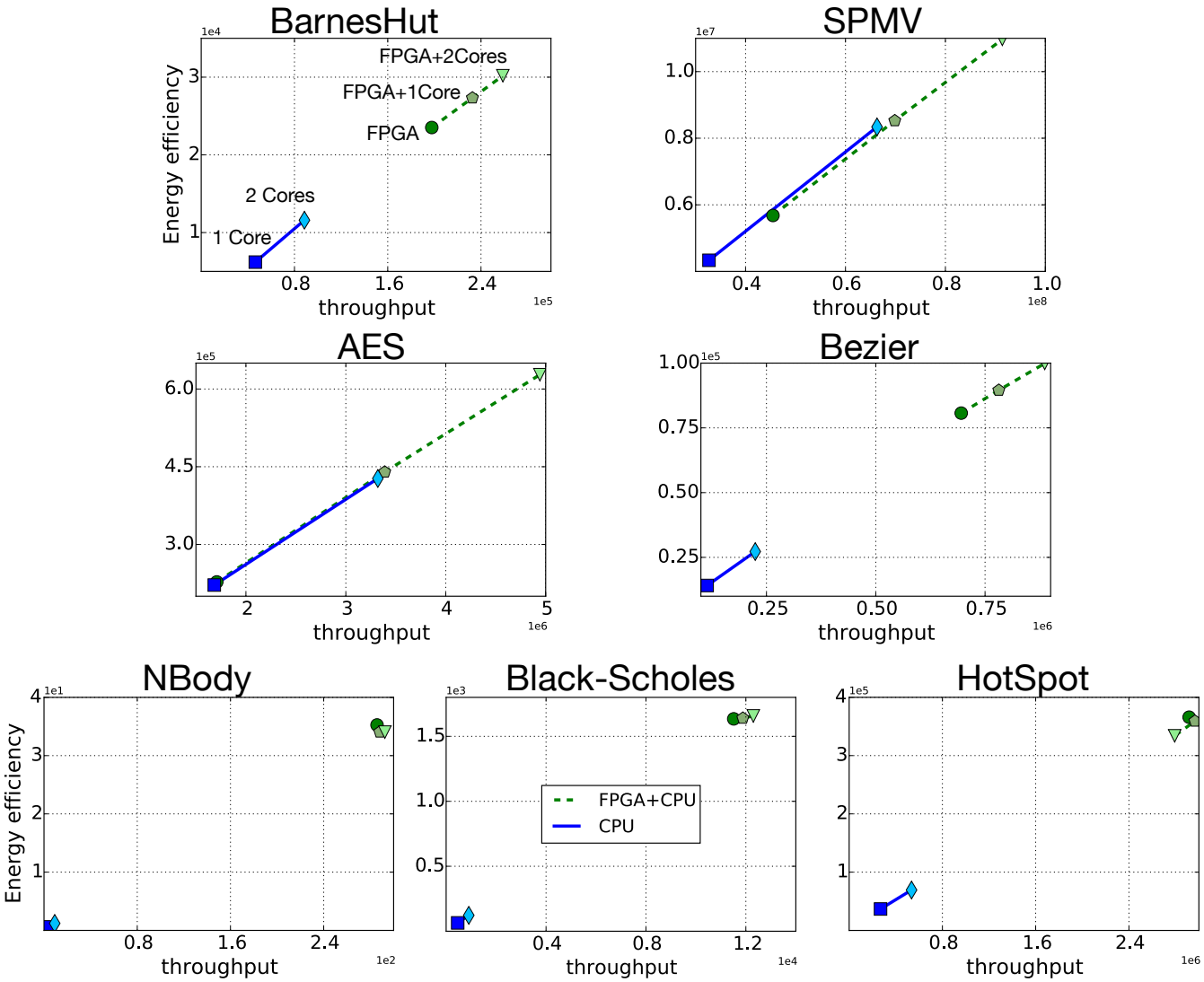


Figure 10: Efficiency-Performance plots for all benchmarks when HAP is invoked. Energy due to total power is accounted vs throughput. The higher the better in both dimensions. Throughput units are the same as in Figure 5. Energy efficiency is throughput per Watt.

are always more energy efficient than 2CORES executions: the range of improvement goes from 1.4x (SPMV) to 28x (Nbody). From Figure 10 we also notice:

- Irregular and regular benchmarks that map relatively well on FPGA like BarnesHut, SPMV, AES and Bezier get the optimal throughput and energy efficiency for 2-device configurations. For these benchmarks, the improvement of the energy efficiency for the 2-device executions range between 2.7x (AES) and 1.2x (Bezier), as we see in Figure 9b.
- Regular benchmarks that map very well on FPGA like NBody, Black-Scholes, and HotSpot obtain marginal throughput and energy efficient improvement when executed on 2-device configurations, being the FPGA-only the optimal configuration for these benchmarks.

7. Conclusions

This paper has extended and tuned a framework to efficiently map parallel iterations to the different compute units in a low-cost and low-power MPSoC, now enabling both CPU cores and an OpenCL-capable FPGA to work simultaneously on disjoint data regions. One key component of the framework is a scheduler able to dynamically and adaptively partition the work among the devices while achieving load balance. We have discovered that the chunk size of iterations offloaded to the FPGA must be carefully selected, an issue particularly key for irregular applications, but also profitable for some regular codes. To tackle this issue we provide the HAP scheduler that automatically finds a near-optimal FPGA chunk size at runtime. Using our framework we have studied the energy efficiency for different configurations by running multiple benchmarks from the HPC domain. Even for our memory bound applications we have found out that two-device configurations (FPGA+2CORES) usually improve performance (up to 2.9x) and energy efficiency (up to 2.7x) compared to the common approach of computing only on the accelerator device. The improvement is even higher when comparing two-device configurations to the implementation that only uses the multicore (2CORES): performance improves up to 32x and energy efficiency up to 28x. Interestingly, we have also learned that configurations that provide higher throughput do not necessarily achieve better energy efficiency when only the running power (i.e. excluding the idle power component) is considered. As future work we will extend the scheduler to find at runtime the device configuration that maximizes energy efficiency, either when considering running power or total power scenarios. Our scheduler will be also extended to target CPU+GPU+FPGA MPSoCs.

Acknowledgment

The authors would like to thank Altera for the generous donation of the DE1 boards through the Altera University Program. We also thanks Luís Piñuel for donating the Power Cape infrastructure that allowed us to measure energy consumption. This work was partially supported by the Spanish projects TIN 2016-80920-R, P11-TIC-08144, TIN2016-76635-C2-1-R, gaZ: T48 research group, UK EPSRC with the ENPOWER (EP/L00321X/1) and the ENEAC (EP/N002539/1) projects.

References

[1] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, J. P. Jue, All one needs to know about fog computing and related edge computing paradigms: A complete survey, Journal of Systems Architecturedoi:https://doi.org/10.1016/j.sysarc.2019.02.009.

[2] ECOSCALE: Energy-efficient heterogeneous computing at EXASCALE, http://www.ecoscale.eu/ ((2016)).

[3] EXTRA: Exploiting exascale technology with reconfigurable architectures, https://www.extrahpc.eu/ ((2015-2017)).

[4] Stratix 10 SoC, https://www.altera.com/products/soc/portfolio/stratix-10-soc/overview.html ((2017)).

[5] H.-E. Zahaf, A. E. H. Benyamina, R. Olejnik, G. Lipari, Energy-efficient scheduling for moldable real-time tasks on heterogeneous computing platforms, Journal of Systems Architecture 74 (2017) 46 – 60. doi:https://doi.org/10.1016/j.sysarc.2017.01.002.

[6] J. Reinders, Intel Threading Building Blocks: outfitting C++ for multi-core processor parallelism, O'Reilly, 2007.

[7] A. Navarro, F. Corbera, A. Rodriguez, A. Vilches, R. Asenjo, Heterogeneous parallel_for template for CPU–GPU chips, International Journal of Parallel Programming.

[8] M. Belviranli, L. Bhuyan, R. Gupta, A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures, ACM Trans. Archit. Code Optim. 9 (4).

[9] D. Bacon, R. Rabbah, S. Shukla, FPGA programming for the masses, Queue 11 (2) (2013) 40:40–40:52.

[10] G. Guidi, E. Reggiani, L. D. Tucci, G. Durelli, M. Blott, M. D. Santambrogio, On how to improve FPGA-based systems design productivity via SDAccel, in: Intl. Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2016.

[11] J. Auerbach, D. F. Bacon, P. Cheng, R. Rabbah, Lime: A java-compatible and synthesizable language for heterogeneous architectures, SIGPLAN Not. 45 (10) (2010) 89–108.

[12] J. Auerbach, D. F. Bacon, I. Burcea, P. Cheng, S. J. Fink, R. Rabbah, S. Shukla, A compiler and runtime for heterogeneous computing, in: Design Automation Conference, 2012, pp. 271–276.

[13] Q. Wu, Y. Ha, A. Kumar, S. Luo, A. Li, S. Mohamed, A heterogeneous platform with GPU and FPGA for power efficient high performance computing, in: Intl. Symp. on Integrated Circuits, 2014.

[14] P. Pandit, R. Govindarajan, Fluidic kernels: Cooperative execution of OpenCL programs on multiple heterogeneous devices, in: Int. Symp. on Code Generation and Optimization, 2014.

[15] A. Vilches, A. Navarro, R. Asenjo, F. Corbera, R. Gran, M. J. Garzarn, Mapping streaming applications on commodity multi-CPU and GPU on-chip processors, IEEE Transactions on Parallel and Distributed Systems 27 (4) (2016) 1099–1115.
[16] P. Meng, M. Jacobsen, R. Kastner, FPGA-GPU-CPU heterogenous architecture for real-time cardiac physiological optical mapping, in: Intl. Conf. on Field-Programmable Technology, FPT'12, 2012, pp. 37–42.
[17] S. Prongnuch, T. Wiangtong, Heterogeneous computing platform for data processing, in: Int. Symp. on Intelligent Signal Processing and Communication Systems, 2016, pp. 1–4.
[18] K. H. Tsoi, W. Luk, Axel: A heterogeneous cluster with FPGAs and GPUs, in: Intl. Symp. on Field Programmable Gate Arrays, FPGA '10, 2010, pp. 115–124.
[19] A. Vilches, R. Asenjo, A. Navarro, F. Corbera, R. Gran, M. J. Garzaran, Adaptive partitioning for irregular applications on heterogeneous cpu-gpu chips, Procedia Computater Science 51 (2015) 140–149.
[20] J. Nunez-Yanez, S. Amiri, M. Hosseinabady, A. Rodríguez, R. Asenjo, A. Navarro, D. Suarez, R. Gran, Simultaneous multi-processing in a software-defined heterogeneous fpga, The Journal of Supercomputingdoi:10.1007/s11227-018-2367-9. URL https://doi.org/10.1007/s11227-018-2367-9
[21] I. Corp., Intel FPGA SDK for OpenCL, best practices guide, https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf (2016).
[22] D. Rudolph, C. Polychronopoulos, An efficient message-passing scheduler based on guided self scheduling, in: 3rd Intl. Conf. on Supercomputing, ICS'89, 1989.
[23] E. Lederer, Cross-device NBody simulation sample, https://software.intel.com/en-us/articles/opencl-cross-devices-nbody-simulation-sample (2014).
[24] M. Kulkarni, M. Burtscher, C. Cascaval, K. Pingali, Lonestar: A suite of parallel irregular programs, in: Intl. Symp. on Performance Analysis of Systems and Software, 2009, pp. 65–76.
[25] N. Bell, M. Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors, in: Conference on High Performance Computing Networking, Storage and Analysis, SC '09, 2009.
[26] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, D. Kaeli, Hetero-mark, a benchmark suite for CPU-GPU collaborative computing, in: Intl. Symp. on Workload Characterization (IISWC), 2016, pp. 1–10.
[27] J. Gómez-Luna, I. El Hajj, V. Chang, Li-Wen Garcia-Flores, S. Garcia de Gonzalo, T. Jablin, A. J. Pena, W.-m. Hwu, Chai: Collaborative heterogeneous applications for integrated-architectures, in: Intl. Symp. on Performance Analysis of Systems and Software (ISPASS), 2017.
[28] I. Corporation, Monte carlo pricing of asian options on FPGAs using OpenCL, https://www.altera.com/support/support-resources/design-examples/design-software/opencl/black-scholes.html (2014).
[29] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, M. R. Stan, Hotspot: Acompact thermal modeling methodology for early-stage VLSI design, IEEE Trans. Very Large Scale Integr. Syst. 14 (5).
[30] Z. Wang, B. He, W. Zhang, S. Jiang, A performance analysis framework for optimizing OpenCL applications on FPGAs, in: Intl. Symp. on High Performance Computer Architecture (HPCA), 2016, pp. 114–125.
[31] K. Krommydas, R. Sasanka, W. c. Feng, Bridging the FPGA programmability-portability gap via automatic OpenCL code generation and tuning, in: Intl. Conference on Application-specific Systems, Architectures and Processors (ASAP), 2016, pp. 213–218.
[32] F. D. Igual, L. M. Jara, J. I. Gómez-Pérez, L. Piñuel, M. Prieto-Matías, A power measurement environment for PCIe accelerators, Computer Science - Research and Development 30 (2) (2015) 115–124.
[33] S. Barrachina, M. Barreda, S. Catalán, M. F. Dolz, G. Fabregat, R. Mayo, E. Quintana-Ortí, An integrated framework for power-performance analysis of parallel scientific workloads, Energy (2013) 114–119.
[34] K. Czechowski, V. W. Lee, E. Grochowski, R. Ronen, R. Singhal, R. Vuduc, P. Dubey, Improving the energy efficiency of big cores, in: Intl. Symp. on Computer Architecuture, ISCA '14, 2014.