Noname manuscript No.
(will be inserted by the editor)

Parallel Multiprocessing and Scheduling on the Heterogeneous Xeon+FPGA Platform

Andrés Rodríguez · Angeles Navarro · Rafael Asenjo · Francisco Corbera · Rubén Gran · Darío Suárez · Jose Nunez-Yanez

Received: date / Accepted: date

Abstract Heterogeneous computing that exploits simultaneous co-processing with different device types has been shown to be effective at both increasing performance and reducing energy consumption. In this paper we extend a scheduling framework encapsulated in a high level C++ template, and previously developed for heterogeneous chips comprising CPU and GPU cores, to new high-performance platforms for the data center, which include a cache coherent FPGA fabric and manycore CPU resources. Our goal is to evaluate the suitability of our framework with these new FPGA-based platforms, identifying performance benefits and limitations. We target the state-of-the-art HARP processor that includes 14 high-end Xeon-class tightly coupled to a FPGA device located in the same package. We select 8 benchmarks from the High Performance Computing domain that have been ported and optimized for this heterogeneous platform. The results show that a dynamic and adaptive scheduler that exploits simultaneous processing among the devices can improve performance up to a factor of 8x compared to the best alternative solutions that only use the CPU cores or the FPGA fabric. Moreover, our proposal achieves up to 15% and 37% of improvement compared to the best heterogeneous solutions found with a Dynamic and Static schedulers, respectively.

Keywords Heterogeneous architecture, FPGA, parallel_for template, heterogeneous scheduling, hybrid algorithm, adaptive chunk size

1 Motivation

To bring up to date Herb Sutter's quote "the free lunch is over", we would say that "lunch is becoming prohibitively expensive". Now it is not the multicore architecture the one being democratized, but the heterogeneous on-chip processor

A. Rodríguez, A. Navarro, R. Asenjo, F.Corbera
Dept. of Computer Architecture, Universidad de Málaga, Andalucía Tech, Spain. E-mail: {andres, angeles, asenjo, corbera}@ac.uma.es

R. Gran, D. Suárez
Computer Architecture Group. Universidad de Zaragoza, Spain. E-mail: {rgran,dario}@unizar.es

J. Nunez-Yanez
Dept. of Electrical & Electronic Engineering. University of Bristol, UK. E-mail: J.L.Nunez-Yanez@bristol.ac.uk

is the one getting momentum. Free lunch was over when developers were forced to face parallel programming. We are now facing heterogeneous programming that is certainly harder.

For instance, heterogeneous chips with integrated GPUs are commonplace in desktops (Intel Coffee Lake or AMD APUs –Accelerated Processing Unit–) and mobile devices (Qualcomm Snapdragon, Samsung Exynos and HiSilicon Kirin to name a few), usually called heterogeneous MultiProcessor System-on-Chip or MPSoC. In our quest to achieve the three Ps (Performance, Productivity and Portability) on those CPU+GPU chips, we recently proposed a heterogeneous library including a scheduling algorithm that dynamically distributes different chunks[1] of a parallel iteration space among CPU cores and a GPU [17]. Performance is achieved by keeping all the cores and the GPU collaborating at the same time. 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. Productivity is achieved because the library offers an easy-to-use API that eases the implementation of an heterogeneous parallel_for. Portability is achieved because the library is based on the Intel Threading Building Blocks, TBB, and OpenCL libraries, that are available for Intel, AMD, ARM and most integrated GPUs.

With the increased focus on energy-efficient acceleration, heterogeneous architectures integrating CPU and FPGA have become attractive platforms to deliver both high performance and low cost in the context of servers, deep learning and exascale, among others [26], [15], [30]. However, these systems are difficult to program posing a challenge to Productivity. A new trend is to couple the CPU and FPGA through cache coherent interconnect. Intel and IBM developed coherent memory interconnect technologies, as QPI, UPI and CAPI, that can provide coherent shared-memory access between the CPU and FPGA. This enables the FPGA to be a peer to the CPU from a memory access standpoint, eliminating the need to move data back and forth between both of them and allowing the offloading of specific tasks to the FPGA in a fine-grained manner.

In this paper, we evaluate the Intel Xeon+FPGA heterogeneous chip, commonly known as HARP system. This platform features 14 Xeon cores along with an Intel-Altera Arria10 FPGA on the same chip, so we will refer to it as Xeon+A10 from now on. Both devices are connected via a QuickPath Interconnect (QPI) bus. As in our CPU+GPU work [17], here we also demonstrate that Performance is increased when the FPGA and the 14 CPU cores share the computational burden. Although the FPGA can run OpenCL kernels, this is not enough to ensure Performance Portability. OpenCL kernels written for GPU execution have to be significantly tuned to make the most out of the FPGA capabilities. In fact, specific FPGA idioms have to be used when porting a kernel to the new accelerator. Also, a more careful chunk size selection which considers the alignment of the data for both the CPU and the FPGA caches, or the selection of the appropriate memory sharing mechanism for the data in global memory were needed in our heterogeneous implementations to efficiently utilize the CPU and the FPGA simultaneously.

All in all, this paper proposes the following novel contributions:

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

1. An extension of our CPU+GPU scheduling framework to consider the requirements of a CPU+FPGA platform. In particular, some restrictions to the size and alignment of the chunks were implemented to better exploit the communication bandwidth between the memory and the FPGA.
2. A novel implementation of the n-body problem based on the concept of an hybrid algorithm tailored to the CPU+FPGA particularities. The idea is to have both devices computing disjoint regions of the data, but each one running a different algorithm for the n-body computation: a regular brute-force NBody algorithm on the FPGA and an irregular CPU-optimized BarnesHut algorithm on the CPU.
3. An evaluation of our extended heterogeneous scheduler, called HAP (from Heterogeneous Adaptive Partitioner), on the Xeon+A10 chip. Our scheduler adaptively and continuously adjusts the size of the chunk of iterations offloaded to CPU cores and the FPGA. We also provide a thorough exploration of the performance of 8 applications in which the main kernels have been carefully tuned for the Arria10.

The rest of the paper is organized as follows. Next section introduces the problem as well as related works that tackle a similar problem. Section 3 briefly describes the proposed programming interface that eases the development of heterogeneous applications and a summary of the implementation details for the different scheduling strategies studied. The description of the testbed, the evaluated benchmarks and their optimization to target the Arria10 as well as the experimental evaluation of our heterogeneous scheduler is covered in Section 4. Finally, we wrap up with conclusions in Section 5.

2 Background and related work

The development of an easy programming framework and its runtime support for a broad class of applications in systems that integrate a multicore CPU and a FPGA is still an open problem [2]. 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. Traditionally, FPGAs are programmed with hardware description languages (HDLs) that are based on hardware-centered abstractions that enable flexibility but do not provide the expressiveness needed by software developers. Recent developments of high-level synthesis (HLS) focus on leveraging programming languages, such as C [29], Java [1], OpenCL [24], Scala [11], and others [20], [21], to raise the level of abstraction.

Most of these previous works mainly use the CPU of the heterogeneous platform for scheduling and pre-processing. So, when the accelerator is processing, the CPU cores become idle. Compared with the existing designs, our framework enables efficient CPU+FPGA simultaneous processing to fully utilize the computing resources of heterogeneous platforms, addressing the load balancing between the CPU cores and the accelerator. Related to our work, a previous research led by Belviranli et. al [4] also proposed a strategy to enable the simultaneous processing of work between the CPU cores and an accelerator (GPU or FPGA) while taking care of the load balancing between devices. However in that work the potential irregularity of the workload was not taken into account, a key consideration in our approach. In Section 3.4 we provide a more detailed discussion comparing

our proposal with [4]. A recent work on the Xeon+A10 platform has considered the co-processing of CPU and FPGA for graph analytics applications (BFS and SSSP) [30], although in this work the FPGA code is generated using HDL low level language, whereas our approach targets higher level hardware specifications. Another recent work has used a framework to orchestrate the cooperation of two discrete accelerators a GPU and a FPGA in addition to the CPU, however, its scheduler still depends on a previous setting of operating parameters [8].

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. To do so, we select a state-of-the-art high-level scheduler called LogFit [27,17] 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 [19], but in that case, instead of OpenCL, SDSoC and C were used to generate the FPGA computing units for regular applications. Irregular applications such as BarnesHut and SPMV were not considered for the Xilinx chip with SDSoC, whereas we implement and study these type of applications in this work.

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 section 3.4. Thus, the resulting HAP scheduler that we present in the paper, represents one of the novel contributions of this work.

The idea of designing hybrid algorithms based on different algorithmic paradigms tailored to the specificities of the devices in order to accelerate the simultaneous co-processing of parallel iterations on CPU+GPU [22] or CPU+FPGA [30] platforms has already been explored, in particular in the context of graph analytics. In this paper we propose a hybrid algorithm to accelerate n-body problems in CPU+FPGA systems, which is another novel contribution of this work.

3 Programming Environment

3.1 Programming Interface

This section introduces the Heterogeneous Building Blocks (**HBB**) library API. 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 that exploits heterogeneous CPU+GPU platforms [17] and that we have extended to also work on CPU+FPGA systems. HBB relies on TBB as the orchestrating task-parallel library and OpenCL as the FPGA back-end for the sake of availability and programmability features. That way, 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 the application instead of dealing with thread management and synchronization.

Figure 1 depicts a simplified scheme of how the *parallel_for()* has been built. The engine is implemented with a two-stage pipeline TBB template. Each token

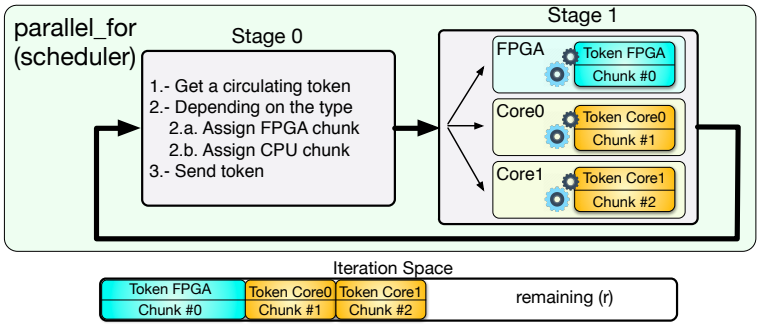


Fig. 1 Scheme of the `parallel_for()` implementation. Stage_0 performs the partitioning and scheduling, while Stage_1 computes the assigned chunks.

corresponds to a compute unit class, FPGA or CPU core in our system, and their sum represents the number of chunks of the iteration space that will be processed in parallel. 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_1, adaptively selects the chunk size for FPGA and CPU tokens, and extracts the corresponding chunk of iterations from the set of remaining iterations, r. The procedure to select the FPGA and CPU chunk sizes is covered in the next section. Then, when a token reaches the second stage, Stage_2, the chunk gets processed on the corresponding device. Stage_1 is a serial stage, so it can host just a token at a time, whereas Stage_2 is parallel, and several tokens can be in this stage simultaneously. We define the concept of Scheduling Interval, I, as each invocation of Stage_1 that results on a FPGA chunk, so that I_i identifies the i^{th} invocation of such Stage that generates the i^{th} chunk of iterations that will be processed on the FPGA. 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.

```
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 }
```

Fig. 2 Using the `parallel_for()` function template.

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

3.2 Function template: parallel_for()

Figure 2 shows a `main` function with all the required component initialization to make the `parallel_for()` function template works. 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 iteration space. The third parameter is the `Body` instance that has the implementations of the CPU and FPGA body loop. Previously, the user has created this `Body` instance (line 4).

The `Body` class must implement two methods: one that defines the code that a CPU core has to execute for an arbitrary chunk of iterations, and the same for the FPGA device using OpenCL. This method 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 `unmap` and `map` OpenCL operations [6] following zero-copy semantics. In any case, the user is responsible of: 1) writing and optimizing the OpenCL code for the FPGA; and 2) compiling 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.).

Program arguments, like the number of threads and scheduler configuration can be read from the command-line, as can be seen in line 6 of Figure 2. The benchmarks that we evaluate in Section 4 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 in how many CPU cores will be processing chunks of the iteration space. The second one can be set just to 0 or 1 to disable or not the FPGA as an additional compute unit. The last argument, `<sch_arg>`, depends on the particular implementation of the heterogeneous scheduler, as we will see in next section.

Orchestrating the body execution and handling the heterogeneous devices require a `Scheduler` class. The class 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 programmers to get rid of error prone low-level management chores, such as thread handling or synchronization operations. Currently, our `HBB` library provides three heterogeneous schedulers: Static, Dynamic and HAP. They can be instantiated as shown in Figure 2 at line 8. For the sake of space, next we cover the schedulers Dynamic and HAP.

3.3 Dynamic Scheduler

If the Dynamic scheduler is selected, only the FPGA chunk size, Ch_f is manually set by the user as a positive integer value ($Ch_f = b$), which is passed in `<sch_arg>`. 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. To that end, the model

described in [18] recommends that: *each time that a chunk is partitioned to be assigned to a compute device, its size should be selected such that it is proportional to the compute device's effective throughput.* Therefore, if n is the number of iterations of the `parallel_for()`, r is the number of remaining iterations (initially $r = n$) and $nCores$ is the number of cores (in our case, $nCores$ =`num_cpu_tokens`), the computation of the CPU chunk, Ch_c, follows equation 1:

$$Ch_c = min\Big(\frac{Ch_f}{f}, \frac{r}{f + nCores}\Big) \tag{1}$$

where f is the *relative speed* and represents how faster the FPGA is w.r.t. a CPU core. The ratio f is recomputed each time a chunk is processed, as explained in Section 3.1. In other words, Ch_c is either (Ch_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* [23]), when there are few remaining iterations, this is when $r/(f + nCores) < Ch_f/f$. The overhead of our Dynamic scheduler can be measured as the time consumed in Stage$_1$, and in our experiments this time always represents less than 0.2% of the total execution time. However, 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.

3.4 HAP Scheduler

To avoid the Dynamic scheduler limitations, HAP (Heterogeneous Adaptive Partitioner) dynamically estimates the FPGA chunk size at run-time. The scheduler is based on the LogFit scheduler [17] and is designed as a three-phase strategy consisting of: the Exploration Phase (EP), the Stable Phase (SP) and the Final Phase (FP).

In the Exploration Phase (EP) we look for an initial FPGA chunk size and a CPU chunk size that maximize the throughput in both devices. To that end, we carry out an exploration in which at each scheduling interval I_i, we increase the size of the FPGA chunk selected in the previous interval and sample the throughput $(\lambda_f(I_i))$ for the new FPGA chunk $(Ch_f(I_i))$. We keep increasing the size of the chunk and sampling its throughput until the throughput is stabilized. At this particular scheduling interval of stabilization, I_s, we perform a first logarithmic fitting to find an expression to model (estimate) the FPGA throughput: $\lambda_f = a_s \cdot ln(Ch_f) + b_s$. To compute this expression we apply the least squares logarithmic fitting to the samples $(Ch_f(I_i),\ \lambda_f(I_i))$, $i = 1..s$, i.e. the chunks and their corresponding measured throughputs that we have collected from all the previous intervals. Next, we compute the parameter $Ref_s = a_s/Ch_f(I_s)$, which gives us a ratio between the estimated FPGA throughput and the chunk size at the point of stabilization. This ratio is the tangent to our fitting curve at this point of stabilization, and it is a reference that represents the slope of the best straight-line approximation to the curve at that point. Also, during this phase, for each sampled FPGA chunk size, we compute a CPU chunk size that balances the load for both devices (see Ch_c in equation 2).

Then, in the Stable Phase (SP), for each new scheduling interval at time t, I_t, we continuously re-adjust the FPGA chunk size, $Ch_f(I_t)$, to cope with the application irregularities, as shown in equation 2. In this equation, a_t comes from function $\lambda_f = a_t \cdot ln(Ch_f) + b_t$, which now represents the logarithmic fitting to estimate the FPGA throughput at scheduling interval I_t. Here, we consider samples $(Ch_f(I_j), \lambda_f(I_j))$, $j = 1..t-1$. Please notice from equation 2 that we also use Ref_s, the reference from the initial fitting performed in the Exploration Phase. In other words, each time we recompute a new logarithmic curve in the Stable Phase and get a new a_t coefficient, we look for the point in the new curve that gets the same slope found in EP: $a_t/Ch_f(I_t) = Ref_s$. From this expression we obtain equation 2. Therefore, Ref_s is the criteria we set to guarantee that we select equivalent optimal points each time we re-adjust the logarithmic fitting curve. Again, at the same interval I_t, we re-compute a CPU chunk size, $Ch_c(I_t)$, that balances the load for both devices while ensuring optimal throughput as we also see in equation 2. Please note, that for computing this chunk size we take as reference the relative speed of the devices in the previous interval $(f_{t-1} = \lambda_f(I_{t-1})/\lambda_c(I_{t-1}))$, which comes from the throughputs just measured.

$$Ch_f(I_t) = \frac{a_t}{Ref_s} \qquad Ch_c(I_t) = \frac{Ch_f(I_t)}{f_{t-1}} \tag{2}$$

The Final Phase (FP) is activated when there are just a few remaining iterations and we have to pay particular attention to finding out the best possible partitioning. 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 ensuring that they finish at the same time. More details about each one of these phases can be found in [17].

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. [4]. However, in Belviranli's work, after the Exploration Phase, the estimations of the throughput for the accelerator and the CPU cores are fixed, and they are used for a guided self-scheduling strategy 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 because: i) for these applications the throughput may change during execution so the initial throughput estimation might not be valid as computation progresses; and ii) when the end of the iteration space is approaching then the guided self-scheduling usually selects a chunk size that is not big enough to achieve the predicted throughput, so the performance of the FPGA quickly degrades, which additionally leads to load imbalance with the CPU. This effect is particularly harmful in time-steps based applications such as the ones we study in this work, finding that for our irregular applications the performance degrades 21% on average when compared to our approach. By contrast, 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.

With respect to our previous scheduler, LogFit, described in [17], that targets CPU+GPU chips, we have extended the Exploration Phase significantly:

1. 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.
2. Since the EP explores now smaller chunk sizes, the execution time for these chunks is less reliable, which leads us to also modify the stabilization criteria. In HAP, 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%. When this criteria is met[4], then we have reached the interval of stabilization.
3. Another optimization 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 last FPGA chunk size as the initial 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.

Due to additional requirements of OpenCL for the FPGA, we also modified the chunk size computation in two aspects:

i. Some OpenCL kernels require the size of the chunk to be a multiple of a certain value. Since equation 2 does not initially guarantee this behavior, we updated it to return the closest multiple of the required size.
ii. Intel-Altera Best Practices guide [6] recommends 64-bytes aligned buffers to allow direct DMA transfers. We actually observed in our experiments that FPGA kernels are particularly susceptible to such data misalignment. To keep all the chunks 64-bytes aligned, we modified the scheduler to ensure that all chunk sizes (FPGA and CPU) are a multiple of 64 bytes which also provides cache alignment on the Xeon+A10 platform. This change reduced data transfer times from/to the FPGA. We studied the effect of selecting misaligned chunks (FPGA and CPU) finding that for our applications the throughput degrades up to 7% w.r.t. 64-bytes aligned chunks.

4 Experimental results

In this section, we present a comparative study of performance of our proposals. Firstly, in subsection 4.1, 4.2 and 4.3, we describe the test-bed, the benchmarks and the different optimizations that we apply to the FPGA kernels, respectively. Subsection 4.4 elaborates on the performance results of a hybrid proposal for the n-body problem that is based on combining a regular and an irregular implementation, tailored to the FPGA and the CPU, respectively. In subsection 4.5 we show the performance results of the Dynamic and HAP schedulers. We also compare HAP against a Static scheduler that partitions the workload once between the computing device taking into account the relative speed of each device.

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

[4] It is not guaranteed that the stabilization criteria is always met. If the number of iterations is not large enough to fully utilize the FPGA we may finish the computation without leaving the Exploration Phase.

4.1 Experimental Settings

We run our experiments on the Xeon+A10 (a.k.a. HARP) that tightly couples an Intel Broadwell 14-core Xeon CPU (2680v4) and an Intel Arria 10 FPGA in the same multi-chip socket. The communication channel between the CPU and the FPGA is Intel's Core Cache Interface (CCI-P), that indeed is an abstraction layer of other communication buses: one QPI and two PCI-Express. In total, CCI-P accumulates a bandwidth of 41.7 GB/s. Broadwell E5 2680v4 processor operates at 2.4Ghz (Turbo 3.3GHz) and features a shared L3 of 35 MB, L2 of 256KB and a TDP of 120W. The Arria 10 GTX1150 FPGA accounts with 1150K Logic Elements, 65 MB of internal memory (M20K + MLAB) and 1518 single-precision floating-point hardware multiplier/adder which delivers a peak performance of 1366 GFLOPS. The FPGA runs kernels compiled with Altera SDK for OpenCL, 64-Bit Offline Compiler version 16.0.2, supporting OpenCL 1.2. Note that this platform is using pre-production hardware and software and therefore the experimental results may not reflect the performance of production or future systems.

4.2 Benchmarks

Our benchmark collection comprises both integer and floating point applications from the High Performance Computing domain: linear algebra (Sparse Matrix Vector Multiplication-SPMV), cryptography (AES), gravitational dynamics (NBody-NB and BarnesHut-BH), physics simulation (HotSpot-HS), 3D graphics (Bezier-BZ), financial (Black Scholes-BS) and molecular dynamics (Bude-BD).

NB: NBody. We study two different implementations of a n-body problem: a regular approach called Nbody [14] 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 the other bodies. For our study an input set of 100,000 bodies and 15 time-steps were simulated.

BH: BarnesHut. An irregular implementation of the n-body problem that employs an octree data structure [13]. 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), resulting in an irregular application. It was used the same input as in NB.

SPMV: Sparse Matrix-Vector Multiplication. This code comes from Bell and Garland work [3], 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 GL7d17 sparse matrix from the University of Florida Sparse Matrix Collection that exhibits a triangular-like profile was selected as input.

AES: Advance Encryption Standard. This benchmark comes from the Hetero-Mark suite [25]. The program takes plain text as input and encrypts it using a given

encryption key. In our experiments we took an input text of 2 Gbytes and 256-bit encryption key length.

BZ: Bezier. This benchmark computes a 3D Bezier surface from a normal surface. The original algorithm is taken from the Chai benchmark [9] 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 4096 by 4096.

HS: HotSpot. An iterative solver that models the heat dispersion on a flat surface using differential equations [10], which comes from the Rodinia Benchmark Suite [5]. The benchmark has two input matrices, 8192x8192 each, that represent the actual temperature and the heating power corresponding to each point of the surface. These matrices are processed into 64-element width shift-registers.

BS: Black-Scholes. This benchmark consists on an Asian Option Pricing Algorithm based on the one provided by Intel-Altera in [7]. 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 models the stock price using 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 the total payoff of the option. We performed 100,000 simulations for each run of this benchmark.

BD: Bude. This benchmark runs a molecular docking algorithm developed at the Bristol University to perform virtual drug screening [16]. Organized in 3 nested loops (poses, ligands, and proteins), the kernel performs intensive floating point computations for 65536 poses.

Both BH and SPMV can be considered irregular benchmarks (due to their irregular memory accesses and their control divergences) while NB, HS, AES, BS, BZ and BD are representative of regular applications.

4.2.1 Hybrid Algorithm: BH-NB

As we said in the previous section, NB is a regular approach in which, for each time-step, each body has to compute the gravitational interaction with every other body. To that end, an array of bodies store the attributes (mass, position, velocity, acceleration) of each body. On the other hand, BH is an irregular variation of the n-body problem that assumes that far-away bodies can be summarized by their center of mass. To implement this alternative, a tree (oct-tree in a 3D space) is used to keep the information of the centers of mass and the bodies belonging to each sector of the space.

As we cover in Section 4.4, the NB implementation fits better on the FPGA whereas the HB algorithm is more suitable for the CPU, so we propose a hybrid BH-NB as an heterogenous implementation that targets the Xeon+A10 heterogenous hardware. Figure 3 sketches how a single data structure is used simultaneously by the FPGA and the CPU.

The main idea is to keep the leaves of the tree in an array of bodies that can be used by the FPGA when running the NB implementation. At the same time, the tree data structure is available for the CPU that executes the BH code. Each chunk of bodies is assigned to the FPGA or the CPU according to our HAP scheduler. For a chunk assigned to the FPGA, the NB computation will update the position,

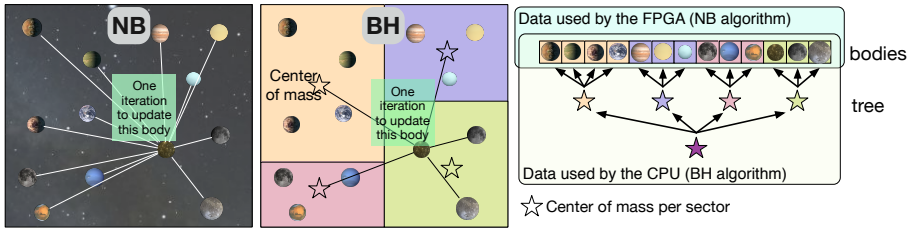


Fig. 3 Data structure used on the BH-NB implementation of the n-body problem.

velocity and acceleration of the bodies in this chunk and for that, all the other bodies will be traversed to compute their contribution to the total gravitational force. Simultaneously, other chunks of bodies will be assigned to the CPU cores and updated, now following the BH algorithm that saves some of the body-body gravitational interactions thanks to the tree data structure. As we confirm in Section 4.4, this heterogeneous algorithm that combines NB and BH outperforms a pure NB or a pure BH alternative.

4.3 Platform specific OpenCL optimizations

Optimizing OpenCL code for the FPGA requires using a plethora of techniques that have to be tailored per application [6], [28], [12]. 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 (second column) refers to single task kernels (also known as single work-item), in which the OpenCL kernel resembles a sequential C implementation. For these type of kernels the OpenCL NDRange[5] is set to (1, 1, 1), so a single thread is invoked on the accelerator. Intel-Altera [6] suggests to structure kernels this way, to foster loop pipelining and allow the overlapping of data transfers and computations between loop iterations. In our study, half of the benchmarks have benefited from single work-item kernels whereas the other four are NDRange kernels. Compute Unit Replication (third column) clones the kernel pipeline -when there is enough availability of resources in the FPGA- for improving throughput. However multiple compute units competing for global memory might lead to undesired memory access patterns that could degrade performance, so it must be used carefully. Due to restrictions in the version of the Altera SDK available in the Xeon+A10 platform, Compute Unit Replication can be applied only to NDRange kernels. Out of the four benchmarks that follow the NDRange approach, Compute Unit Replication is exploited only in BH and BD because for NB and SPMV a single compute unit already uses more than half of the FPGA resources. Loop Unrolling of inner loops (fourth column) allows increasing the number of parallel operations effectively improving the throughput. Loop Unrolling optimization is also limited by the availability of FPGA resources. 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

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

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.

	Task	Compute Unit Replication	Loop Unrolling	Locality
BH	✗	5	0	RP
NB	✗	1	48	RP
SPMV	✗	1	32	RP
AES	✓	1	16	CM, RP, SRP
BZ	✓	1	6	LM, CM
HS	✓	1	64	SRP, RP
BS	✓	1	64	LM, AC, RP
BD	✗	2	4	LM

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.

On the Xeon+A10 platform, it is possible to exploit the OpenCL1.2 zero-copy buffer optimization since the CPU and the FPGA can share the same memory region[6]. However, due to current driver limitations, CPU caching is disabled for the shared memory region, which severely hits CPU performance (up to 37% of degradation in our benchmarks). 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 their computation time, the traditional OpenCL host-to-device and device-to-host operations are used (BH, BZ, AES, NB, BS). These memory transfers represent less than 1.5% of the computing overhead. 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. That overhead is less than 2% in our codes. 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 slightly better performance figures can be expected.

Table 2 summarizes the resource usage for all benchmarks under test and the resultant FPGA frequency in MHz. The variability is large because resource utilization mainly depends on kernel organization and arithmetic intensity. For example, DSP utilization ranges between 81% and 2% for BZ and SPMV, whereas it is 0% for the integer benchmark AES. On top of that, there is no correlation between resource usage and performance as shown in the sequel.

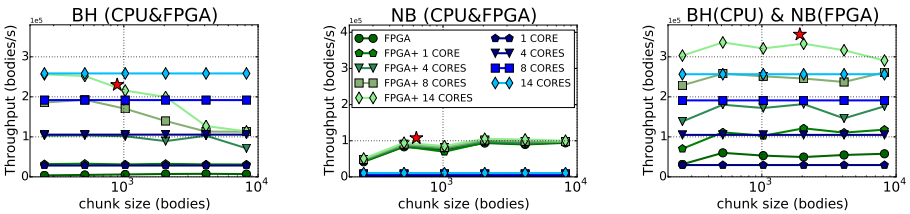
[6] This is achieved by allocating the region with clCreateBuffer(..., CL_MEM_ALLOC_HOST_PTR, size, ...) and then mapping this region to a CPU accessible pointer using clEnqueueMapBuffer().

	Logic (%)	Reg. (%)	Mem. (%)	DSP (%)	CLK (MHz)	II (cycles)
BH	55	66	87	10	178	NA
NB	51	76	76	70	200	NA
SPMV	42	68	83	2	184	NA
AES	27	24	24	0	276	1
BZ	50	10	21	81	197	2
HS	52	29	28	13	272	1
BS	63	37	35	33	232	4
BD	55	46	71	44	197	NA

Table 2 AOC Compiler Resource Usage Summary per category (% of total resources) compiling for Altera Arria 10. CLK and II Stands for kernel clock cycle and initiation interval. NA stands for "Not Applicable" since II is only reported by Altera SDK for Single Task kernels.

4.4 Hybrid algorithm case study: BH-NB

Figure 4 shows a study about how the size of the chunk of parallel iterations assigned to each device affects the throughput of the execution for our n-body implementations when different hardware configurations were selected. For the one-device configurations (1, 4, 8 and 14 CORES, 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+1, 4, 8 and 14 CORES) we apply the heterogeneous Dynamic and HAP schedulers presented in section 3.3. For the Dynamic scheduler we explore different sizes of the chunk offloaded to the FPGA (each size shown on the "x" axis), while the chunk assigned to a CPU core is computed adaptively by the scheduler. For HAP, considering that the FPGA chunk size is continuously re-adapted, we only show one point, depicted by a red five-point star (★), whose coordinates are (<average FPGA chunk size>, <average throughput>) for the FPGA+14 CORES configuration. This is, instead of showing the range of chunk sizes used throughout the execution, the x-axis value of the ★ represents the average of all the automatically estimated FPGA chunk sizes.



(a) BarnesHut (b) NBody (c) Hybrid BH-NB

Fig. 4 Throughput of single algorithm (NBody, BarnesHut) and hybrid algorithm (BH-NB) implementations of the n-body problem.

As we see in Figure 4(a), the irregular BH implementation achieves the highest throughput for CPU-only execution (14 CORES). Heterogeneous executions that include the FPGA (Dynamic and HAP) provide lower performance. In this code, the random memory accesses are highly irregular resulting in dramatically higher memory access latency. As a consequence, the accelerator stalls and the performance significantly deteriorates: the FPGA is 1/34x slower than the multicore CPU. Thus, including the FPGA will only add some imbalance what explains the degradation of heterogeneous executions (FPGA+4 CORES, FPGA+8 CORES, ...), in particular

when exploring big chunk sizes for the FPGA. On the other hand, Figure 4(b) shows that the regular NB implementation obtains the highest throughput for the HAP heterogeneous execution that includes the FPGA (red star), although the improvement with respect to FPGA-only execution is small. Now, NB traverses the bodies in a streaming fashion, making the FPGA an appropriate platform for acceleration. In fact, the FPGA is now 10x faster than the multicore CPU. Clearly, the FPGA benefits the most when a regular algorithm is implemented, obtaining a throughput of 9.48E+04 bps (bodies per second), while the multicore benefits from an irregular approach as BH that reduces the computational complexity of the n-body problem, achieving a throughput of 2.59E+05 bps in our case. This dramatic difference in performance motivated us to explore the trade-offs between BH and NB, and to propose a hybrid algorithm to accelerate the n-body problem when targeting a CPU+FPGA heterogeneous platform.

Figure 4(c) shows the throughput of the hybrid BH-NB algorithm for the one-device and the heterogeneous two-device configurations. Now, the heterogeneous implementation offloads chunks of iterations to the multicore CPU and computes them using BH, while simultaneously offloads chunks of iterations to the FPGA which are computed using NB. Now, the best heterogeneous execution of the hybrid BH-NB, obtained with our scheduler HAP, outperforms the highest throughput of BH (NB) by up to 1.4x (3.7x).

4.5 Evaluation of Heterogeneous Schedulers

Figure 5 summarizes the best speedups for the optimal chunk sizes of iterations that each of our heterogeneous schedulers, Dynamic (DYN) and HAP (HAP), obtained for our benchmarks. For both schedulers, the configuration that gets the highest throughput employs the FPGA + 14 CPU cores, simultaneously. The speedup is computed against the CPU only executions (CPU), which are shown as reference. The CPU results are obtained for 14 CPU cores. We also show the speedups for the FPGA only executions (FPGA), i.e. all iterations of the parallel loop are offloaded to the accelerator.

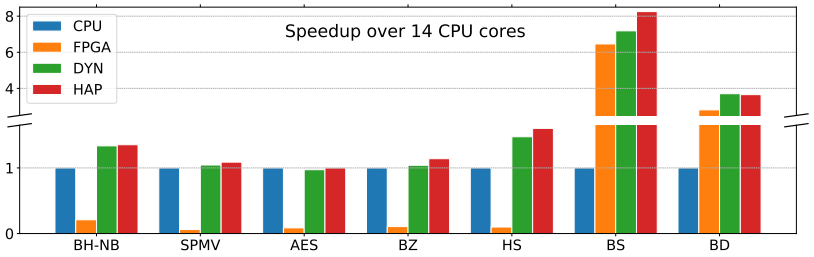


Fig. 5 Speedups of each benchmark w.r.t. CPU (14 CORES) execution when executing only on CPU (CPU), on FPGA (FPGA), on CPU + FPGA with Dynamic Scheduler (DYN) and on CPU + FPGA with HAP Scheduler (HAP).

As we see from the figure, the heterogeneous executions generally improve performance when compared to the homogeneous ones (CPU or FPGA). The accelerator clearly outperforms the multicore for BS and BD. In fact, for both codes,

the heterogeneous schedulers achieve up to 8x and 3.7x improvement, respectively, compared with the CPU-only. For the rest of the codes, the improvement of the heterogeneous schedulers is more modest when compared to the CPU-only: it goes between 1.6x (HS) and 1x (AES). For AES we also observe a slight degradation of throughput with the DYN scheduler (0.97x). For these latter codes, the FPGA implementation is not so competitive against the multicore, but it still contributes to the increment in throughput. The lack of improvement in AES with the DYN scheduler is due to the fact that the FPGA achieves the highest throughput with big chunks. Offloading big chunks to the accelerator creates a small load imbalance in the DYN heterogeneous execution, which is avoided with the adaptive FPGA chunk size selection in the HAP scheduler.

As a summary, Figure 5 shows that for all codes (except BD), HAP improves the throughput up to 15% compared to DYN. For BD the degradation of throughput is marginal: smaller than 1%. These results confirm that the overhead of the Exploration Phase is not significant, and that HAP saves the burden of manually tuning the FPGA chunk size while still obtaining some improvements.

	BH-NB	SPMV	AES	BZ	HS	BS	BD
Dyn (CS)	512	512	65536	1024	64	4096	8192
HAP (Avg. CS)	638	1792	29808	1036	788	5464	16024

Table 3 Optimal FPGA Chunk Size (CS) for each heterogeneous scheduler. For HAP we show average chunk sizes (the chunk size changes during execution).

In our evaluation, we performed also an exhaustive exploration of different chunk sizes for the Dynamic scheduler. We found that small chunks degraded performance due to underutilization of the deep pipeline FPGA implementations. On the other hand, very big chunks could create load imbalance among the CPU cores and the FPGA. In any case, this exploration is avoided with the HAP scheduler. Table 3 shows the optimal chunk sizes obtained for both schedulers. As we see, HAP tends to find bigger average chunk sizes. We also found that for the same program, the average chunk size that HAP gives for each configuration (FPGA only, FPGA + 1 CPU core, FPGA + 2 CPU cores, etc.) could differ. Let us note that the chunk sizes reported for HAP are average values. As HAP is an adaptive approach, we found that the chunk sizes selected in the Final Phase (that partitions the last remaining iterations among all the computing devices) differ depending on the configuration. For instance, the Final Phase computes smaller chunk sizes for configurations with higher number of CPUs, so the average chunk size reported for these configurations tends to be smaller. As an example, we found that for the the FPGA + 14 CPU configuration, average chunk sizes reduce between 1% and 18% when compared to FPGA only.

We also carried out a performance comparison between HAP and a baseline scheduler based on a Static partitioner that assigns one big chunk to the FPGA and the rest of the iterations to the CPU. The size of this single FPGA chunk is computed by a previous offline search phase that measures the average throughput of the application when running first on the FPGA and later on the CPU multicore (14 CPU cores in our experiments). Once these average throughputs are measured, they are used to compute the relative speed of each device, and next to calculate one block of iterations that are assigned to the FPGA while the rest are assigned to the CPU cores to ensure that both devices take the same time theoretically. The

goal of this study is to measure if a Static approach can outperforms our adaptive solution. For our benchmarks, Table 4 shows the ratio of the iteration space (RIS, normalized from 0 to 1) assigned to the FPGA with the Static scheduler and the % of improvement that HAP gets with respect to that Static scheduler (%IMP). From the results we see that HAP always improves performance, by speedups between 8% and 37% when compared to a Static solution. The improvement is more significant in those benchmarks for which a smaller ratio of iterations (RIS) is offloaded (because the CPU cores are much faster than the accelerator). For these cases a Static partition is clearly discouraged.

	BH-NB	SPMV	AES	BZ	HS	BS	BD
RIS	0.17	0.05	0.07	0.09	0.09	0.86	0.73
%IMP	37%	27%	26%	18%	13%	8%	8%

Table 4 HAP vs Static scheduler comparison: RIS = Ratio of Iteration Space assigned to the FPGA with Static; %IMP = Percentage of Improvement of HAP vs Static.

The improvement that HAP achieves when compared to Static and Dynamic approaches comes from the fact that our proposal continuously adapts the FPGA and CPU chunk sizes to: i) the code behavior (that is essential in irregular codes as SPMV and BH-NB), and ii) runtime variations (that is useful for designing robust and fluctuation-tolerant schedulers, even for regular applications, as our evaluation shows for BZ, HS, BS and BD).

As a final evaluation, we also computed an *Ideal Ratio*, IR, as the ratio of the throughput measured in HAP with respect to the ideal throughput which we estimate by assuming that both the FPGA and the CPU multicore can work seamlessly simultaneously. This ideal throughput is the sum of the throughputs for FPGA and for 14 CPU cores (one-device configurations). In other words, this IR ratio helps us to characterize if heterogeneous executions in which the two devices work simultaneously allow the aggregation of individual performances, or on the contrary, if the final performance might be degraded due to contention for shared resources (in this case, the QPI bandwidth is the critical shared resource). We found that the ideal ratio is close to 1. This means that the degradation from the ideal throughput is in general marginal: it goes from 0.2% (NB-BH) to 12% (AES). These results indicate that heterogeneous executions in our benchmarks are not constrained by the shared resources, so simultaneous co-processing that fully utilizes the computing resources of this heterogeneous platform is highly recommended.

5 Conclusions

In this paper we propose a scheduling framework, encapsulated in a high level C++ template, which enables simultaneous co-processing of parallel loop iterations in CPU+FPGA heterogeneous platforms. One key component of the scheduler is that is able to dynamically and adaptively partition the work among the available devices ensuring near-optimal throughput while achieving load balancing. Our evaluation teaches us that the size of the chunk of iterations offloaded to the FPGA must be carefully selected, taking into account that chunks must be aligned to 64 bytes. Moreover, the adaptability of the chunk size has proved to be profitable

for irregular applications, but also for regular codes. Experimental results for a diverse set of benchmarks from the HPC domain show that our heterogeneous HAP scheduler generally improves performance when compared to one-device executions, achieving up to 8x speedup with respect to CPU-only executions. HAP also achieves up to 15% of improvement compared with the best solution found with a Dynamic scheduler, or up to 37% of improvement compared to a Static approach (used as baseline).

We also propose a novel hybrid algorithm targeted to CPU+FPGA platforms to accelerate the n-body problem. Taking advantage of our heterogeneous scheduler that allow us to partition the iterations among the devices to process disjoint regions of data, our algorithm selects a regular brute-force NBody approach on the FPGA and an irregular BarnesHut approach on the CPU, achieving up to 3.7x (1.4x) throughput improvement compared with the best NB (BH) implementations.

As future work we will extend the scheduler to find at runtime the device configuration and work partition that maximizes energy efficiency, either when considering running power or total power scenarios. We also plan to extend our scheduler to target CPU+GPU+FPGA hardware platforms.

Acknowledgment

The authors would like to thank Intel-Altera for the opportunity to be part of the HARP program. This work was partially supported by the Spanish projects TIN 2016-80920-R, TIN2016-76635-C2-1-R, gaZ: T48 research group, UNIZAR JIUZ-2017-TEC-09, UK EPSRC with the ENPOWER (EP/L00321X/1) and the ENEAC (EP/N002539/1) projects.

References

1. Auerbach, J., Bacon, D.F., Cheng, P., Rabbah, R.: Lime: A java-compatible and synthesizable language for heterogeneous architectures. SIGPLAN Not. **45**(10), 89–108 (2010)
2. Bacon, D., Rabbah, R., Shukla, S.: FPGA programming for the masses. Queue **11**(2), 40:40–40:52 (2013). DOI 10.1145/2436696.2443836
3. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Conf. on High Perf. Computing Networking, Storage and Analysis, SC '09 (2009)
4. Belviranli, M., Bhuyan, L., Gupta, R.: A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. ACM Trans. Archit. Code Optim. **9**(4) (2013)
5. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: 2009 IEEE Intl. Symp. on Workload Characterization (IISWC), pp. 44–54 (2009)
6. Corp., I.: 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)
7. Corporation, I.: 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)
8. Dávila Guzmán, M.A., Nozal, R., Gran Tejero, R., Villarroya-Gaudó, M., Suárez Gracia, D., Bosque, J.L.: Cooperative CPU, GPU, and FPGA heterogeneous execution with EngineCL. J. of Supercomputing **75**(3), 1732–1746 (2019)
9. Gómez-Luna, J., El Hajj, I., Chang, L.W., Garcia-Flores, V., Garcia de Gonzalo, S., Jablin, T., Pena, A.J., Hwu, W.m.: Chai: Collaborative heterogeneous applications for integrated-architectures. In: Intl. Symp. on Perf. Analysis of Systems and Software (ISPASS) (2017)
10. Huang, W., Ghosh, S., Velusamy, S., Sankaranarayanan, K., Skadron, K., Stan, M.R.: Hotspot: A compact thermal modeling methodology for early-stage VLSI design. IEEE Trans. Very Large Scale Integr. Syst. **14**(5) (2006)

11. Koeplinger, D., Prabhakar, R., Zhang, Y., Delimitrou, C., Kozyrakis, C., Olukotun, K.: Automatic generation of efficient accelerators for reconfigurable hardware. In: Computer Architecture (ISCA), ACM/IEEE 43rd Intl. Symp. on, pp. 115–127 (2016)
12. Krommydas, K., Sasanka, R., c. Feng, W.: Bridging the FPGA programmability-portability gap via automatic OpenCL code generation and tuning. In: Intl. Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 213–218 (2016)
13. Kulkarni, M., Burtscher, M., Cascaval, C., Pingali, K.: Lonestar: A suite of parallel irregular programs. In: Intl. Symp. on Perf. Analysis of Systems and Software, pp. 65–76 (2009)
14. Lederer, E.: Cross-device NBody simulation sample. https://software.intel.com/en-us/articles/opencl-cross-devices-nbody-simulation-sample (2014)
15. Li, Z., Liu, L., Deng, Y., Yin, S., Wang, Y., Wei, S.: Aggressive pipelining of irregular applications on reconfigurable hardware. In: 2017 ACM/IEEE 44th Intl. Symp. on Computer Architecture (ISCA), pp. 575–586 (2017)
16. McIntosh-Smith, S., Price, J., Sessions, R.B., Ibarra, A.A.: High performance in silico virtual drug screening on many-core processors. Intl. J. of High Performance Computing Applications **29**(2), 119–134 (2015)
17. Navarro, A., Corbera, F., Rodriguez, A., Vilches, A., Asenjo, R.: Heterogeneous parallel_for template for CPU–GPU chips. Intl. J of Parallel Programming **47**, 213–233 (2019)
18. Navarro, A., Vilches, A., Corbera, F., Asenjo, R.: Strategies for maximizing utilization on multi-CPU and multi-GPU heterogeneous architectures. J. of Supercomputing **70**(2), 756–771 (2014)
19. Nunez-Yanez, J., Amiri, S., Hosseinabady, M., Rodríguez, A., Asenjo, R., Navarro, A., Suarez, D., Gran, R.: Simultaneous multiprocessing in a software-defined heterogeneous FPGA. The Journal of Supercomputing (2018). DOI 10.1007/s11227-018-2367-9
20. Oguntebi, T., Olukotun, K.: Graphops: A dataflow library for graph analytics acceleration. In: Intl. Symp. on Field-Programmable Gate Arrays, pp. 111–117. ACM (2016)
21. Prabhakar, R., Koeplinger, D., Brown, K.J., Lee, H., De Sa, C., Kozyrakis, C., Olukotun, K.: Generating configurable hardware from parallel patterns. SIGOPS Oper. Syst. Rev. **50**(2), 651–665 (2016). DOI 10.1145/2954680.2872415
22. Remis, L., Garzarán, M.J., Asenjo, R., Navarro, A.G.: Exploiting social network graph characteristics for efficient BFS on heterogeneous chips. J. Parallel Distrib. Comput. **120**, 282–294 (2018). DOI 10.1016/j.jpdc.2017.11.003
23. Rudolph, D., Polychronopoulos, C.: An efficient message-passing scheduler based on guided self scheduling. In: 3rd Intl. Conf. on Supercomputing, ICS'89 (1989)
24. Singh, D.: Implementing FPGA design with the OpenCL standard. Altera whitepaper (2011)
25. Sun, Y., Gong, X., Ziabari, A.K., Yu, L., Li, X., Mukherjee, S., Mccardwell, C., Villegas, A., Kaeli, D.: Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In: Intl. Symp. on Workload Characterization (IISWC), pp. 1–10 (2016)
26. Umuroglu, Y., Morrison, D., Jahre, M.: Hybrid breadth-first search on a single-chip FPGA-CPU heterogeneous platform. In: Intl. Conf. on Field Programmable Logic and Applications (FPL), pp. 1–8 (2015). DOI 10.1109/FPL.2015.7293939
27. Vilches, A., Asenjo, R., Navarro, A., Corbera, F., Gran, R., Garzaran, M.J.: Adaptive partitioning for irregular applications on heterogeneous CPU-GPU chips. Procedia Computer Science **51**, 140–149 (2015)
28. Wang, Z., He, B., Zhang, W., Jiang, S.: A performance analysis framework for optimizing OpenCL applications on FPGAs. In: Intl. Symp. on High Performance Computer Architecture (HPCA), pp. 114–125 (2016)
29. Windh, S., Ma, X., Halstead, R.J., Budhkar, P., Luna, Z., Hussaini, O., Najjar, W.A.: High-level language tools for reconfigurable computing. Proceedings of the IEEE **103**(3), 390–408 (2015). DOI 10.1109/JPROC.2015.2399275
30. Zhou, S., Prasanna, V.K.: Accelerating graph analytics on CPU-FPGA heterogeneous platform. In: Intl. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 137–144 (2017). DOI 10.1109/SBAC-PAD.2017.25