View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Explore Bristol Research

Nunez-Yanez, J., Hosseinabady, M., Rodríguez, A., Asenjo, R., Navarro, A., Gran-Tejero, R., & Suárez-Gracia, D. (2018). Simultaneous Multiprocessing on a FPGA+CPU Heterogeneous System-On-Chip. In *Parallel Computing is Everywhere* (pp. 677-686). (Advances in Parallel Computing; Vol. 32). IOS Press. https://doi.org/10.3233/978-1-61499-843-3-677

Peer reviewed version

Link to published version (if available):
10.3233/978-1-61499-843-3-677

Link to publication record in Explore Bristol Research
PDF-document

This is the author accepted manuscript (AAM). The final published version (version of record) is available online via IOS Press at http://ebooks.iospress.nl/publication/48666 . Please refer to any applicable terms of use of the publisher.

University of Bristol - Explore Bristol Research
General rights

This document is made available in accordance with publisher policies. Please cite only the published version using the reference above. Full terms of use are available: http://www.bristol.ac.uk/pure/about/ebr-terms

Simultaneous Multiprocessing on a FPGA+CPU Heterogeneous System-On-Chip

Jose NUNEZ-YANEZ [a] Mohammad HOSSEINABADY [a]
Andrés RODRÍGUEZ [b] Rafael ASENJO [b] Angeles NAVARRO [b]
Rubén GRAN-TEJERO [c] Darío SUÁREZ-GRACIA [c]
[a] *University of Bristol, United Kingdom*
[b] *Universidad de Málaga, Spain*
[c] *Universidad de Zaragoza, Spain*

Abstract. In this paper, we investigate how to enhance an existing software-defined framework to reduce overheads and enable the parallel utilization of all the programmable processing resources present in systems that include FPGA-based hardware accelerators. To remove overheads, a new hardware platform is created based on interrupts, which removes spin-locks and frees the processing resources. Additionally, instead of simply using the hardware accelerator to offload a task from the CPU, we propose a scheduler that dynamically distributes the tasks among all the resources to minimize load unbalance. The experimental evaluation shows that the interrupt-based heterogeneous platform increases performance by up 22% while reducing energy requirements by 15%. Additionally, we measure between 50% to 25% reduction in execution time when the CPU cores assist FPGA execution at the same level of energy requirements depending on hardware speed-ups.

Keywords. FPGAs, heterogeneous, interrupts, dynamic scheduler, performance improvement, energy reduction.

1. Introduction

Heterogeneity is seen as a path forward for computers to deliver the energy and performance computing improvements needed over the next decade. In heterogeneous architectures, specialized hardware units accelerate complex tasks. A good example of this trend is the introduction of GPUs (Graphics Processing Units) for general purpose computing combined with multicore CPUs. FPGAs (Field Programmable Gate Arrays) are an alternative high performance technology that offer bit-level parallel computing in contrast with the word-level parallelism deployed in GPUs and CPUs. In a typical configuration, the host CPU employs the FPGA accelerator to offload the work and then remains idle. In this research, we investigate a cooperative strategy applied to compute intensive applications in which both the CPU and FPGA perform the same task on different regions of the input data. The proposed scheduling algorithm dynamically distributes different

chunks of the iteration space between a dual-core ARM CPU and a FPGA fabric integrated in the same die. The objective is to measure if simultaneous computing among these devices could be more favourable from an energy and/or performance points of view compared with offloading to the FPGA and the CPU idling. The FPGA and CPUs are programmed with the same C/C++ language using the SDSoC (Software Defined SoC) framework which enables very high productivity and simplifies the development of drivers to interface the processor and logic parts.

The novelty of this work can be summarized as follows:

1. An enhanced platform for the SDSoC framework with a new interrupt-based framework which resolves the problem of CPU resources wasting energy and clock cycles while idle waiting for the parallel hardware accelerators to complete their tasks.
2. A productive programming interface based on an extension of `parallel_for` template of the TBB (Threading Building Blocks) task framework and its integration with the SDSoC framework to allow its exploitation on heterogeneous CPU-FPGA chip processors.
3. A scheduling algorithm that monitors the throughput of each computing device (CPU cores and FPGA) during the execution of the iteration space and uses this metric to adaptively resize the CPU chunks to optimize overall throughput and to prevent under-utilization and load unbalance.

The rest of this paper is organized as follows. Section II presents an overview of related work in the area of heterogeneous computing with FPGAs, GPUs and CPUs. Section III introduces the details of the novel SDSoC simultaneous multiprocessing platform and Section IV the scheduler and programming interface. Section V presents the considered benchmarks and the performance and energy results. Finally, section VI concludes the paper.

2. Background and related work

The idea of balancing the workload among devices has been explored previously in the literature mainly around systems that combine GPUs and CPUs. For example, a study with desktop CPUs and GPUs has been done in [4] where percentages of work to both devices are assigned before making a selection based on heuristics. With CPUs and GPUs, also energy aware decisions have been considered in [1], which requires proprietary code. Another related work in the context of streaming applications [8] considers performance and energy when looking for the optimal mapping of pipeline stages to CPU and on-chip GPU. The possibility of using GPU+CPU and FPGA simultaneously and collaboratively has also received attention in diverse application areas such as medical research [2]. 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 since the idea is to build a streaming pipeline with results moving processed data from one stage to the next. Data is captured and initially processed in the FPGA then moved with DMA engines to the CPU and GPU components. The heterogeneous solution achieves a 273× speed-up over a

multi-core CPU implementation. A study of the potential of FPGAs and GPUs to accelerate data center applications is done in [5]. 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 this work and the power measurements for the FPGA are based on worst case tool estimations and not direct measurements. In this research we explore a horizontal collaborative solution more closely related to the work done in [7]. That work focuses on a multiple device solution similar to our work and demonstrates how the N-body simulation can be implemented in a heterogeneous solution in which both FPGA and GPU work together to compute the same algorithm kernel on different portions of particles. While our approach uses a dynamic scheduling algorithm to compute the optimal split, in [7] the split is calculated manually with 2/3 of the workload given to FPGA and the rest to GPU; the collaborative implementation is 22.7× faster than the CPU only version. In summary, we can conclude that the available literature has largely focused on advancing the programming models to make the use of FPGAs in heterogeneous systems more productive, comparing the performance of GPGPUs, FPGAs and CPUs for different types of applications in large scale clusters, and creating systems that manually choose the optimal device for each part of the application and move data among them. In contrast, in this paper we select a state-of-the-art high-level design flow based on C/C++ for single-chip heterogeneous CPU+FPGA and extend it to support simultaneous computing performing dynamic workload balancing.

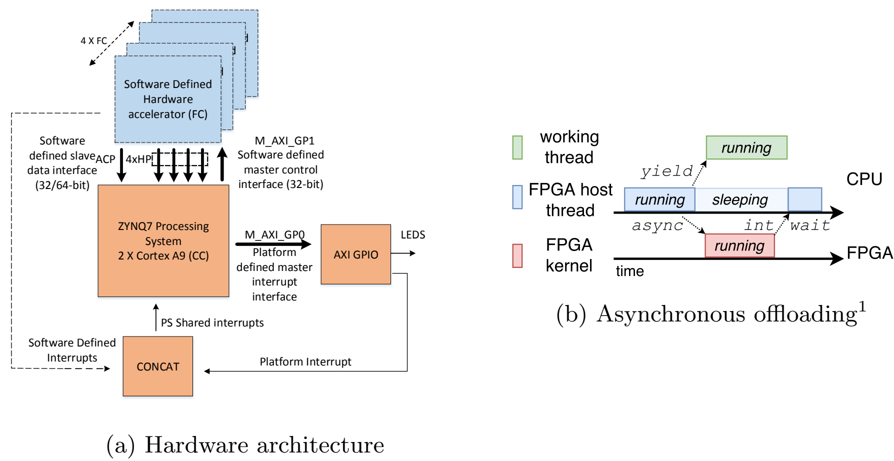
3. Simultaneous multiprocessing SDSoC platform. Hardware Description

The SDSoC environment is able to generate hardware acceleration blocks that run on the FPGA, that can integrate DMA engines to autonomously read input data and write output data without a host thread intervention. The host thread and the accelerator manage the communication status using memory mapped registers, which are accessed through a slave interface of the AXI_LITE communication bus. Additional master interfaces are typically implemented in the acceleration block to read the input and output data from and to main memory as it is processed.

SDSoC offers `async` and `wait` pragmas to enable asynchronous execution on the FPGA so that the hardware function returns control immediately to the host thread. Once the host thread reaches the `wait` pragma it enters into a spin-lock and it is continuously busy waiting for the hardware (FPGA) to complete. This is inefficient since the CPU core that allocates the host thread cannot perform any useful work and consumes energy with 100% utilization.

To address that problem, this work proposes extending the SDSoC framework with an interrupt mechanism. To do that, a dedicated AXI memory-mapped interface enables the acceleration hardware block to generate interrupts to the host thread once processing has completed on the FPGA. This removes the need for the host thread to constantly poll the status of the hardware registers for completion using the default slave interface and effectively frees one CPU core

for other tasks. That is, after asynchronously launching the hardware function, the host thread will yield voluntarily the CPU by changing to the sleeping state. After that, it will only wake up once the hardware accelerator has completed its task. During this sleep time, the dynamic scheduler proposed in Section 4 is able to allocate another working thread to the running state to perform useful work in the CPU core left idle (see Fig.1(b)). To enable this approach, a new hardware platform is proposed consisting of an additional IP block capable of generating shared interrupts to the main processor as illustrated in Fig. 1(a).



(a) Hardware architecture

(b) Asynchronous offloading[1]

Figure 1. SDSoC multiprocessing platform. CC and FC stand for “CPU core” and “FPGA Compute Unit”, respectively.

The proposed IP is based on an AXI GPIO block whose interrupt line connects through a concatenate block (CONCAT) to the processor interrupt inputs. The Linux kernel driver created by SDSoC for the GPIO is replaced with a custom kernel driver capable of putting the host thread to sleep when asked to do so by the user application. Once the hardware block completes its operation, it writes an AXI GPIO register that generates the interrupt causing the host thread to wake up.

```
1 //.h
2 void mmult(float * in_a, float * in_b, float * out_c, int begin,
      int end, int *scalar,int *status, int enable);
3 //.cpp
4  #pragma SDS async(1)
5   mmult(in_a, in_b, out_c, begin, end, scalar, status, enable);
6   ret_value = ioctl(file_desc, IOCTL_WAIT_INTERRUPT,0);
7  #pragma SDS wait(1)
```

Figure 2. Prototype and invocation for hardware function `mmult`

[1]For the sake of simplicity, the operating system activity has been excluded from the figure.

The code snippet of Fig. 2 shows the function declaration and calling style for a hardware function called `mmult`. The first three parameters of `mmul` hardware function are pointers identifying the input/output memory areas which the function uses to receive/send data, while `begin` and `end` will be used to define how much data in these memory areas must be processed. SDSoC constraints require that since the call to the hardware function uses the `async/wait` pragmas (lines 4 and 7), the function prototype cannot contain a return value. An additional value must be supplied in the function prototype, called `scalar` in this example, that will be modified by the hardware function. This `scalar` is implemented using AXI_LITE interface and read by the host thread to learn when hardware processing has completed. The slave interface type AXI_LITE means that this parameter cannot be used to generate the interrupt since it will only be read during the execution of `wait` pragma. The mechanism for interrupt generation itself needs a master interface that can write the GPIO register without the intervention of the host. To solve this problem an additional pointer is introduced in the function prototype called `status` that will be implemented as an AXIMM master interface. `status` points to the AXI GPIO register that controls interrupt generation and is written by the IP once the processing has completed. Then the AXI GPIO can generate an interrupt to the host processor hence waking up the host thread. In the code we can also see the Linux `ioctl` function (line 6) that will ask the kernel driver to put the host thread to sleep until the arrival of the interrupt. Finally, the `enable` parameter is used to control if the IP block is able to generate interrupts. If this parameter is passed with value 1, the IP can generate interrupts otherwise interrupt generation is disabled.

4. Programming Environment

4.1. Programming Interface

This section introduces the proposed Heterogeneous Building Blocks (**HBB**) library API. It is a C++ template library that takes advantage of heterogeneous processors and facilitates his usage and configuration. HBB aims to make easier the programming for heterogeneous processors by automatically partitioning and scheduling the workload among the CPU cores, and the accelerator. It builds on top of the SDS (Xilinx SDSoC library) and TBB libraries, and it offers a `parallel_for()` function template to run on heterogeneous CPU+FPGA systems. In Fig. 3 we depict an MPSoC with an integrated FPGA and two CPU cores (CC), as the one used in the experimental evaluation. The FPGA itself can contain a number of FPGA compute units (FC) depending on resource availability and accelerator configuration.

The left part of Fig. 3 shows the software stack that supports the user application. Our library (HBB) offers an abstraction layer that hides the initialization and management details of TBB and SDS constructs (contexts, command queues, device_ids, etc), thus the user can focus on his own application instead of dealing with thread management and synchronization. The right part of Fig. 3 shows that the internal engine that manages the `parallel_for()` function is a two-stage pipeline, Stage_1(S1) and Stage_2(S2), implemented with the TBB pipeline template.

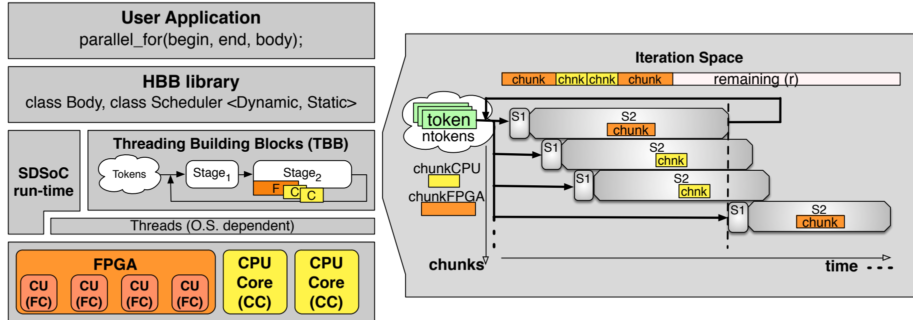


Figure 3. Heterogeneous Scheduler

At the top of this part we can see the iteration space with the chunks that have already been assigned to a processing resource (in orange for the FPGA and yellow for the two CPU cores) and the remaining iterations with the iterations that have not been assigned yet (in white). The right part of the figure shows an execution of the pipeline with 3 `tokens`. The tokens represent the number of items (chunks) that are processed in parallel. The time required for the computation of the chunk on the FPGA and on a CPU core is recorded. This time 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 as we will see in Section 4.2.

```
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    //Static  * hs = Static ::getInstance(&p);
9    Dynamic * hs = Dynamic::getInstance(&p);
10   ...
11   hs->parallel_for(begin, end, body);
12   ...
13 }
```

Figure 4. Using the `parallel_for()` function template

Fig. 4 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 user has to create a `Body` instance (line 4) that will later be passed to the `parallel_for()` function. Program arguments, like the number of threads and scheduler configuration can be read from the command-line, as can be seen in line 6. The benchmarks that we evaluate accept at least three command-line arguments: `<num_cpu_tokens>`, `<num_fpga_tokens>` and `<sch_arg>`. The first one sets the number of CPU tokens, which translates into how many CPU cores will be processing chunks of the iteration space. The second one can be set

just to 0 or 1 to disable or not the FPGA as an additional computing resource. The last argument, `<sch_arg>`, depends on the particular implementation of the heterogeneous scheduler, as we will see in Section 4.2.

```
1  class Body{
2
3  public:
4    void operatorCPU(int begin, int end) {
5       for(i=begin; i!=end; i++){
6          c[i] = a[i] * b[i]; }
7    }
8
9    void operatorFPGA() (int begin, int end){
10     mmult((float*)array_a,(float*)array_b,(float*)array_c, begin,
          end, scalar, status, enable);
11   }
12 };
13 ...
```

Figure 5. Definition of Class Body

Before using the `parallel_for()` function, the user must implement a `Body` class in order to define the body of the parallel loop, as we see in Fig. 5. 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. The `operatorCPU()` method (lines 4-7 in Fig. 5) defines the CPU code of the kernel, and the `operatorFPGA()` method (lines 9-11) calls a hardware function that has been already implemented in the FPGA using the SDSoC development flow.

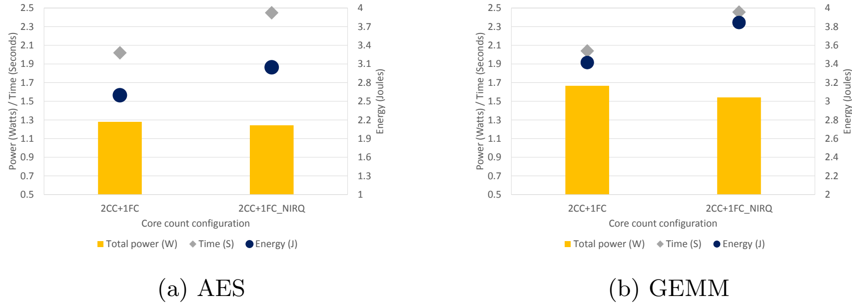
4.2. Scheduling strategies

This section covers the computation of the chunk size that will be executed by the CPU cores and the FPGA. We implement different scheduling policies, but in this work we focus in the dynamic scheduling strategy.

When the dynamic scheduling is selected (see line 9 in Fig. 4), then a FPGA chunk size must be manually set by the user. In this case the argument `<sch_arg>` is a positive integer that sets the size of the chunks of iterations that will be offloaded to the FPGA, (`<sch_arg>`$=S_f$), whereas the CPU chunk size is automatically computed by a heuristic described in [3]. Assuming that n is the number of iterations of the `parallel_for()`, $nCores$ the number of CPU cores, and r the number of remaining iterations (initially $r = n$), then the computation of the CPU chunk, Sc, follows the next expression:

$$S_c = min\Big(\frac{S_f}{f}, \frac{r}{f + nCores}\Big)$$

where f represents how much faster the FPGA is w.r.t. a CPU core, and it is recomputed each time a chunk is processed, as explained in Section 4.1. In other words, S_c is either (S_f/f) (the number of iterations that a CPU core must perform to consume the same time as the FPGA) when the number of remaining iterations, r, is sufficiently high, or $r/(f + nCores)$ (a *guided self-scheduling strategy* [6]), when there are few remaining iterations, this is when $r/(f + nCores) < S_f/f$.



(a) AES

(b) GEMM

Figure 6. Interrupt-based platform evaluation

5. Heterogeneous computing evaluation

The evaluation is based on two well-known benchmarks: AES (Advanced Encryption Standard) and GEMM (General Matrix Multiplication). Both benchmarks are written in C/C++ for both FPGA and CPU targets, and the FPGA functions are compiled using the high-level synthesis tools that are part of the SDSoC framework. The evaluation of the benchmarks is performed on a ZC702 board equipped with a Zynq 7020 device. This board contains a PMBUS (Power Manager BUS) power control and monitoring system that enables the reading of power and current values using the ARM CPUs and the infrastructure provided by Xilinx. For the power measurements the values of power corresponding to the processing system (CPU cores), programmable logic (FPGA) and memory have been measured and added together. For the energy computation we multiply this value for the execution time of the benchmark for different configurations. The GEMM already uses all the available FPGA resources with a single FPGA CU (1FC) but with AES a parallel configuration with 2 FPGA CU (2FC) is possible, so this configuration is also included in the experimentation. Note that the number of FPGA CUs is transparent to the user and to the scheduler, that will only see a potentially faster FPGA when the number of FC increases.

To validate the interrupt-based mechanism we perform an experiment with both benchmarks. We launch the heterogeneous platform with 2 CPU cores (2CC) and 1 FPGA CU (1FC) with and without interrupt capabilities. The results shown in Fig. 6 compare the performance of the original platform that uses the standard ASYNC/WAIT call (2CC+1FC_NIRQ) with the platform that implements the proposed interrupt-based mechanism (2CC+1FC). The configuration with the host thread waiting for the hardware accelerator to finish in 2CC+1FC_NIRQ results in lower power compared with the configuration in which the host thread sleeps and a working thread actively computes work on the available CPU core. However both execution time and energy increase when the interrupt mechanism is not enabled. The test results confirm that the interrupt-based platform increases performance by approximately 22%/18% and reduces energy requirements by 12%/15% for GEMM and AES, respectively. The interrupt-based platform is selected for the next experiments.

Figs. 7 and 8 show performance, power and energy consumption when we explore different chunk sizes for the FPGA (X axis) in our dynamic scheduling

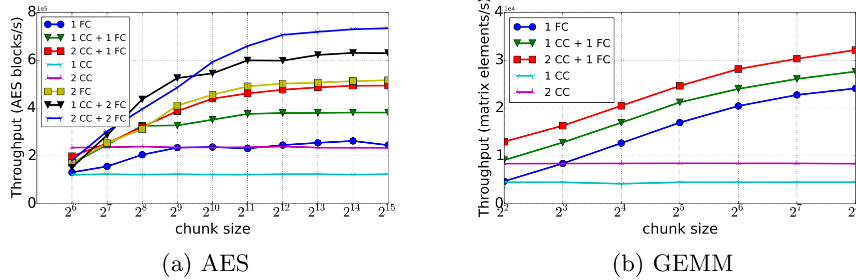


Figure 7. Benchmarks performance analysis

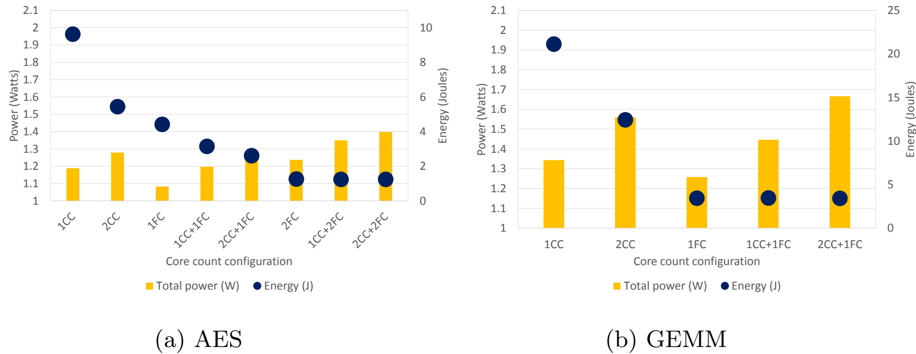
strategy. Note that the CPU chunk sizes are determined adaptively, as explained in Section 4.2. Different configurations are evaluated: 1/2CC (1/2 CPU cores), 1FC (1 FPGA CU), 1/2CC+1FC (1/2 CPU cores + 1 FPGA CU), and additionally for AES, 1/2CC+2FC (1/2 CPU cores + 2 FPGA CU). Fig. 7(a) shows the performance evaluation of AES. The heterogeneous configuration with 2 hardware CU and 2 CPU cores (2CC+2FC) is the fastest. At this configuration up to 82%[2] of the work is performed by the FPGA. In contrast, the hardware configuration with 1 FPGA CU and two CPU cores shown in red (2CC+1FC) offloads 53% of the workload to FPGA and is 2.3 times slower. AES shows a reduction in execution time of 50% when both CPU cores assist 1 FPGA CU and 33% when they assist 2 FPGA CU. The energy and power results shown in Fig. 8(a) show that the most energy efficient configurations use 2 FPGA CU. Adding CPU cores in parallel to the FPGA do not improve energy since the increase in power and reduction in time compensate each other. The lowest power is achieved with 1 FPGA CU and putting the CPU cores to sleep. The GEMM algorithm in Fig. 7(b) also shows the performance gains achieved by the heterogeneous execution and the benefits of assigning larger blocks for FPGA execution. The fastest configuration uses both CPU cores and 1 FPGA CU with a split of 75% of execution on the FPGA. GEMM shows a reduction in execution time of 25% when both CPU cores assist the FPGA CU. The power and energy results shown in Fig. 8(b) indicate that the energy costs are equivalent for all configurations that use the FPGA and lower than the CPU only configurations (1/2 CC). The lower power configuration uses the FPGA and puts the CPU cores to sleep.

6. Conclusion

This paper puts together an interrupt-based communication mechanism with a dynamic scheduler, and a friendly `parallel_for` programmer interface to effectively share work on a FPGA+CPU system-on-chip for improving performance at the same level of energy consumption.

The experiments show that even with a modest amount of CPU participation (only 25% in GEMM and 18% in AES) a noticeable performance gain can be

[2] From now on, results discussion corresponds to the best performing chunk size election for each case.



(a) AES (b) GEMM

Figure 8. Benchmarks power and energy

achieved with heterogeneous computing. Heterogeneous configurations that allow the CPU cores collaborate with the FPGA reduce execution times from 25% to 50%. If the objective is to minimize energy, then the heterogeneous versions tend to be energy neutral since the additional power required by the CPU cores is compensated by the reduction in execution time. Future work includes the generalization of the methodology with more benchmarks and different hardware.

References

[1] R. Dolbeau, F. Bodin, and G. C. de Verdire. One opencl to rule them all? In *2013 IEEE 6th International Workshop on Multi-/Many-core Computing Systems (MuCoCoS)*, pages 1–6, Sept 2013.

[2] Pingfan Meng, Matthew Jacobsen, and Ryan Kastner. Fpga-gpu-cpu heterogenous architecture for real-time cardiac physiological optical mapping. In *Intl. Conf. on Field-Programmable Technology, FPT'12*, pages 37–42, 2012.

[3] A. Navarro, A. Vilches, F. Corbera, and R. Asenjo. Strategies for maximizing utilization on multi-CPU and multi-GPU heterogeneous architectures. *The Journal of Supercomputing*, 2014.

[4] Prasanna Pandit and R. Govindarajan. Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 273:273–273:283, 2014.

[5] S. Prongnuch and T. Wiangtong. Heterogeneous computing platform for data processing. In *2016 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*, pages 1–4, Oct 2016.

[6] David C. Rudolph and Constantine D. Polychronopoulos. An efficient message-passing scheduler based on guided self scheduling. In *Proceedings of the 3rd international conference on Supercomputing*, ICS '89, pages 50–61, 1989.

[7] Kuen Hung Tsoi and Wayne Luk. Axel: A heterogeneous cluster with fpgas and gpus. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, pages 115–124, 2010.

[8] A. Vilches, A. Navarro, R. Asenjo, F. Corbera, R. Gran, and M. J. Garzarn. Mapping streaming applications on commodity multi-cpu and gpu on-chip processors. *IEEE Transactions on Parallel and Distributed Systems*, 27(4):1099–1115, April 2016.