A Programming Model and Runtime System for Approximation-Aware Heterogeneous Computing

Ioannis Parnassos, Nikolaos Bellas, Nikolaos Katsaros, Nikolaos Patsiatzis, Athanasios Gkaras,
Konstantinos Kanellis, Christos D. Antonopoulos, Michalis Spyrou and Manolis Maroudas
Electrical and Computer Engineering Department
University of Thessaly
Volos, Greece
Email: nbellas@inf.uth.gr

***Abstract*—Heterogeneous platforms that include diverse architectures such as multicore CPUs, FPGAs and GPUs are becoming very popular due to their superior performance and energy efficiency. Besides heterogeneity, a promising approach for minimizing energy consumption is through approximate computing which relaxes the requirement that all parts of a program are considered equally important to the output quality, thus, all should be executed at full accuracy.**

Our work extends a traditional OpenMP-like programming model and runtime system to support seamless execution on hybrid architectures with approximation semantics. Starting from a common application code, annotated with our programming model, the programmer can not only target heterogeneous architectures comprising CPU, FPGA and GPU components, but can also regulate the amount of approximation. We evaluate our framework on a number of large-scale applications and demonstrate that the combination of heterogeneous and approximate computing can provide a powerful dynamic interplay between performance and output quality.

I. Introduction

FPGA accelerators offer superior performance, power and cost characteristics compared to a homogeneous CPU-based platform, and are more energy efficient than GPU platforms. However, heterogeneity poses the problem of how to efficiently program a collection of those processing platforms without requiring heroic efforts on the part of the programmer. Besides heterogeneity, approximate computing has been proposed to increase energy efficiency and improve performance [5]. This is possible because, for some applications, not all computations and not all data are equally critical, requiring to be performed or maintained at 100% accuracy or correctness. For such applications, it may be possible to only approximate the final output (or part of it), rather than computing the exact results.

Data-parallel programming models such as OpenCL are being used to port applications to CPUs and GPUs and also as a High Level Synthesis (HLS) framework for FPGAs [6]. Still missing is a higher level programming abstraction that allows the developer to focus on inter-component parallelism in a heterogeneous environment, and an underlying software framework that lifts the burden of task execution and data movement between different address spaces. The contributions of this paper are the following:

(i) We propose a task-based, OpenMP-like, programming model which allows the programmer to simultaneously express parallelism and approximation semantics on heterogeneous architectures which include multicore CPUs, GPUs and FPGAs.

(ii) We design and implement a proof-of-concept runtime system that provides support for the programming model.

(iii) We present a rigorous evaluation of our software framework on a heterogeneous platform running realistic application workloads. We analyze the trade-offs between performance and accuracy for multiple possible platform configurations and we show drastic performance improvements without requiring considerable effort from the programmer.

The rest of the paper is structured as follows. Section II describes the hardware platform. Section III introduces the programming model and Section IV details the runtime system. Section V presents the experimental evaluation of our software framework. Finally, Section VI concludes the paper.

II. Heterogeneous Architecture

Our target platform comprises multicore CPUs, GPUs and FPGA devices interconnected via PCIe buses. GPUs and FPGA accelerators are used to execute OpenCL kernels. In each case, we assume error-free processing units so that approximation is due to algorithmic/software transformations and not due to approximate hardware (unlike for example in [3]).

We use RIFFA, an open-source framework, to provide an abstraction for software developers to access the FPGA as a PCIe-based accelerator [4]. The RIFFA hardware implements the PCIe Endpoint protocol and from the accelerator side, it provides a set of streaming channel interfaces that send and receive data between the CPU main memory and the customizable logic.

A System Manager unit in the FPGA is used a) to orchestrate data movement between the RIFFA channels, accelerators and memories, b) to schedule the operations in the FPGA conforming to data dependences between kernels, and c) to pull the profile data from the Profiler BRAM and send them back to the CPU memory through the RIFFA TX channel. In the current platform, the System Manager is implemented in hardware as an FSM.

The FPGA includes additional hardware mechanisms for cycle-accurate monitoring and profiling of events [7]. Events

that are monitored include accelerator and peripheral utilization, bus transactions with producer-consumer granularity, and data transfers through the RIFFA channels.

III. Approximation-Centric Programming Model

Our programming model adopts a task-based paradigm using directives to annotate dependencies and approximations. Tasks are implemented as OpenCL kernels, containing both the accurate and the approximate version of the code, if available. Listing 1 shows the `#pragma acl`[1] directives used for task processing (lines 1 to 6). The task body specifies the accurate implementation of the task, defined as a function call, which corresponds to an OpenCL kernel. The `approxfun()` clause provides the approximate implementation of the task. This is usually a simpler, faster and less accurate version of the accurate kernel.

```
1  #pragma acl task [approxfun( function )]
2  [significant( expr )]
3  [in( varlist )] [out( varlist )] [inout( varlist ) ]
4  [workers( int_expr_list )] [groups( int_expr_list )]
5  [bind( device_type )] [label( "name" )]
6  accurate_task_impl(...) ;
7
8  #pragma acl taskgroup label( "name" )
9
10 #pragma acl taskwait [label( "name" )]
```

Listing 1: Task and taskgroup specifications

The `significant()` clause specifies the relative significance of the computation implemented by the task for the quality of the output, with a value (or an expression) taking the values 0 or 1. If set to 1 or omitted, the runtime will always execute the task accurately. If set to 0, the runtime will always execute the task approximately, or discard it if an approximate version is not available. Future programming model extensions will have significance values in the range [0.0, 1.0] to allow the runtime system more leverage to decide, during execution, which kernel version, precise or approximate, to execute.

The programmer must also specify the input and output parameters of each task, using the `in()`, `out()` and `inout()` clauses. This information implicitly specifies data dependencies between tasks, and is exploited by the runtime system to perform data flow analysis and data management as explained in Section IV.

Since the tasks are implemented as OpenCL kernels, the programmer specifies the geometry for kernel execution (number of work-items and work-groups) via the `workers()` and `groups()` clauses, which follow the semantics of local and global work size of OpenCL, respectively. OpenCL kernels are able to run on every device on the heterogeneous system but sometimes the implementation is optimized for a specific device. For those reasons, the programmer can explicitly bind a task for execution on a specific device using the `bind()` clause. A possible usage would look like this: `bind(ACL FPGA)`, associating the OpenCL task to an FPGA.

The programmer can cluster multiple tasks in a task group using the `taskgroup` directive and the `label()` clause (line 8). Lastly, the `taskwait` directive specifies an explicit synchronization point, acting as an execution and memory barrier. By default, `taskwait` waits on all issued tasks created so far, unless the `label()` clause is present, which limits the explicit barrier only to tasks of the specific task group.

[1] **A**pproximate Open**CL**

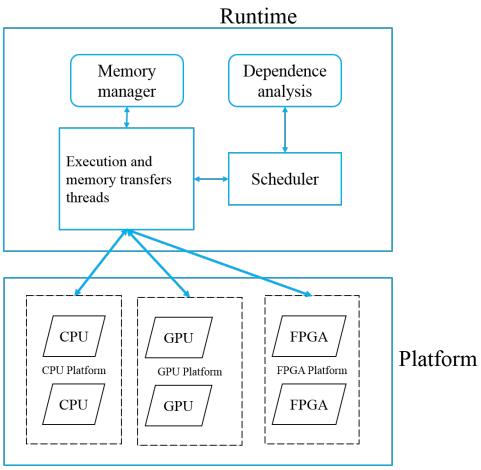

Fig. 1: Outline of the runtime system

We have implemented a source-to-source compiler to translate the directives of the programming model to runtime system API calls. The compiler is implemented as an LLVM/Clang pass.

IV. Runtime System

Figure 1 depicts the architecture of the runtime system used to support the functionality offered by the programming model. It is organized as a coordinator/worker work-sharing scheduler. An important aspect of the runtime system, is the automatic data flow analysis at the granularity of tasks. The runtime system exploits the information provided by the programmer via the `in/out` clauses and keeps track of the memory ranges read and written by each task. This knowledge is used for a) detecting data dependencies between tasks and b) automating memory transfers among different address spaces of the heterogeneous system.

For each device on the system, the runtime creates two threads in the Host CPU: (a) a memory transfer thread, responsible for transparent data transfers between the CPU host and the device, and (b) a task issue thread, responsible for issuing tasks (implementing the OpenCL kernels), for execution to the corresponding device. A task has both the accurate and approximate OpenCL kernels pre-compiled and stored in a fat-binary, but we also support Just In Time (JIT) compilation if necessary. For CPUs and GPUs, our runtime reuses the underlying vendor OpenCL implementation for each device for data transfers, code execution, as well as to identify system configuration.

We use the Vivado toolset of Xilinx to transform the C code to Verilog and generate the bitstream for the FPGA device. The runtime system calls the RIFFA drivers to communicate data between the device and the CPU memory and to trigger the invocation of the hardware accelerator. We have also extended the RIFFA API to be able to configure and access hardware profiling and monitoring information as explained in Section II.

The runtime system also undertakes the responsibility for moving memory objects across address spaces and retaining data coherence across the system. When the execution thread selects a task for execution on a device, the thread responsible for memory management checks the availability of the task input data in the memory of that device and makes the necessary arrangements so that the input data are present in the memory of the device before task execution.

V. Experimental Evaluation

A. Methodology and Benchmarks

We evaluated our framework in a platform comprising (i) an Intel i7-4820K, eight-core CPU running at 3.70 GHz with 32 GB main memory, (ii) an Nvidia GeForce GTX-770 GPU with 1536 streaming processors (SPs) running at 1.1 GHz, and (iii) a VC707 board with a Virtex-7 FPGA and 1 GB external DRAM. The GPU and the FPGA boards are connected to the CPU via a PCIe 2.0 bus. All codes for each platform, both accurate and approximate versions wherever available, are precompiled and stored in fat-binary files. Application binaries and bitstreams have been created with the Intel OpenCL SDK 2016 and Nvidia *clcc* compilers for the x86 and GPU, respectively, and with the Vivado and Vivado HLS 2016.2 toolset for the FPGA. The FPGA target frequency is 250 MHz in all cases.

TABLE I: Benchmark suite

Benchmark	Domain	Input Set	Error Metric
Histogram of Oriented Gradients (HOG)	Computer Vision	30 images from INRIA dataset	Perc. of undetected pedestrians
Molecular Dynamics (MD)	Physics Simulation	Bounding Box side = 300 Å, dt=0.02secs, dur=1sec, 32768 particles	Energy Relative Error
JPEG	Video/Image compression	One 4096x4096 image	100/PSNR
Fisheye	Multimedia	One 2059x1944 fisheye image	100/PSNR

The **Histogram of Oriented Gradients (HOG)** [2] is a computer vision application for object detection. HOG receives an image as input and returns the locations of detected human pedestrians. The input image is processed iteratively in different scales to capture the different sizes of pedestrians appearing in the scene.

We use two independent approximate techniques in HOG, targeting computations with a noticeable effect on performance. First, we periodically eliminate some of the scaled images during the iterative processing. In our experiments, we drop one scaled image every five (processing 25 out of 31 scalings per image) or every ten scaled images (processing 28 out of 31 scalings per image). The second technique is to reduce the accuracy of the histogram which is the most computationally expensive kernel of HOG.

Molecular Dynamics (MD) is an N-body method widely used to simulate particle systems in a wide range of scales from solids and liquids to the motion of stars and galaxies. We approximate the simulation by setting a cut-off distance so that interactions are computed only among particles within the region.

The **JPEG** benchmark consists of three kernels which are applied on 8x8 pixel blocks: (i) the Discrete Cosine Transform (DCT) transforms 2D pixel images to spatial frequency coefficients, (ii) the Quantization / inverse Quantization (Q / iQ) compress the range of coefficient values to a narrower range of values and uncompress it back (lossy compression), and (iii) the inverse DCT (iDCT) reconstructs a lossy version of the initial image using the output of the Q / iQ kernel.

The computation of low frequency coefficients (located at the 2x4 upper left part of an 8x8 block) is more important than the computation of the remaining coefficients. The approximate version of DCT only computes those and sets the remaining 56 coefficients to zero. The approximate iDCT only uses the non-zero coefficients to reconstruct the block of pixels.

Finally, **Fisheye** lens distortion correction is an image processing application which transforms distorted wide angle (fisheye) images back to the natural-looking perspective space [1]. The most computationally demanding phase, bicubic interpolation, is used to approximate intermediate points of a continuous event given the interpolation nodes, or, in this case, pixel points. The method requires the use of the 16 pixel values of a 4x4 window around the interpolated point.

The approximate version of Fisheye relaxes the requirement of bicubic interpolation to use a 4x4 window of pixels, and, instead, it uses only the neighboring 2x2 window.

B. Evaluation

Figure 2 shows the experimental results for a range of runtime decisions for task executions and quality levels. For *HOG*, we started with the fastHOG CUDA implementation [8] which is optimized for GPU execution, and we converted all kernels to OpenCL. In the CPU execution, the histogram kernel accounts for 45% of execution time and is a prime target for GPU and FPGA acceleration. We benchmarked five configurations as follows: (i) CPU-only execution, (ii) GPU-only execution, (iii) GPU running the histogram kernel (H), (iv) FPGA running the histogram kernel, and, finally, (v) FPGA running the histogram kernel and GPU running the SVM kernel. In the last three cases, the Host CPU executes the remaining code. Performance analysis shows that mapping all kernels to the GPU consistently outperforms any other configuration.

HOG is rather sensitive to fluctuations in approximation, a testament of the lack of algorithmic redundancies of this application. Even a 10% reduction of the number of scalings (i.e. processing 28 out of 31 scales per image) results into a 70% pedestrian detection, down from the 84% detection when the code executes accurately. On the other hand, skipping histogram interpolation across neighboring bins is mainly harmful to accuracy when combined with scaling reduction.

MD performs better when the single OpenCL kernel is mapped on the GPU or the FPGA. Approximate execution significantly improves performance for CPU and somewhat for GPU and FPGA execution, without noticeable quality degradation (0.2% relative error), which is typical of most N-body applications. thereby to its energy.

JPEG performance profits from approximation by 1.75, 1.23 and 6.63 times when executing on the CPU, GPU and

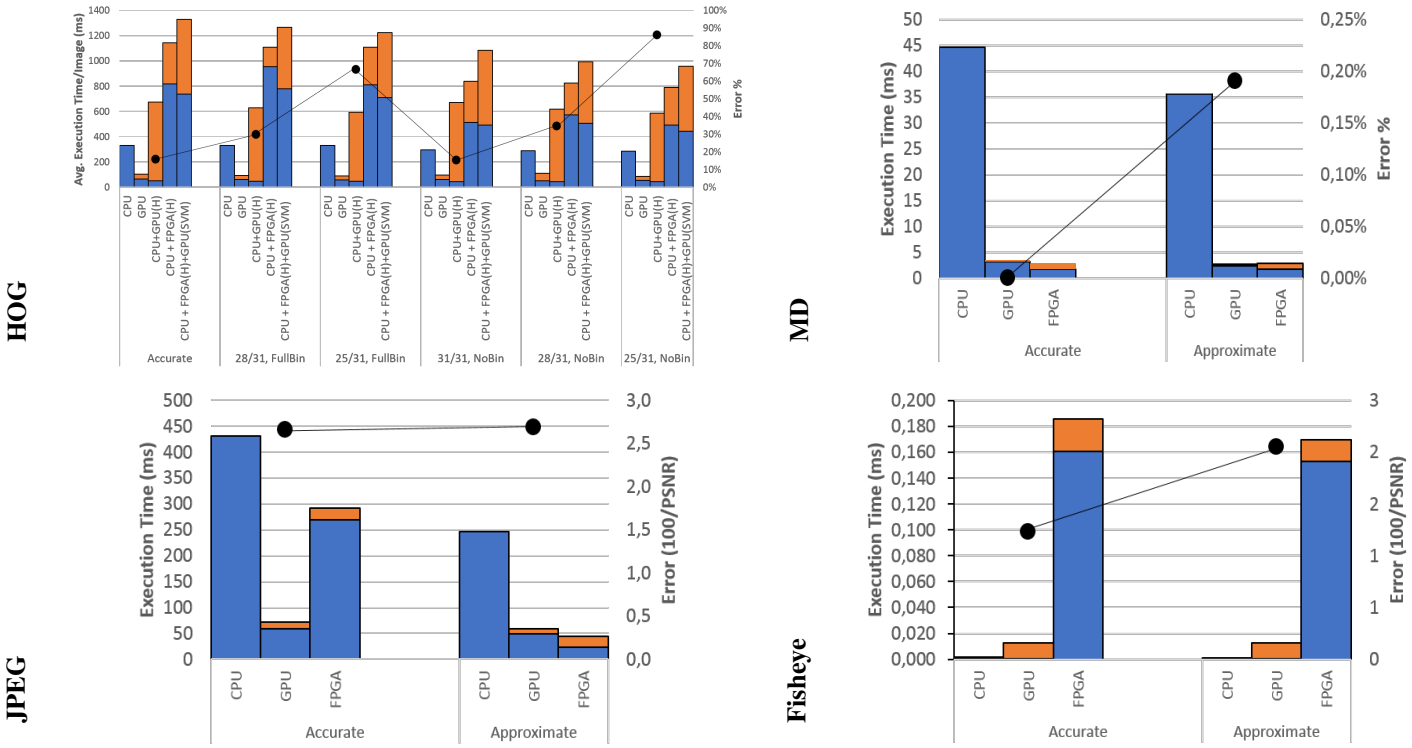

Fig. 2: Experimental results for the benchmarks of Section V-A. Execution time is shown in bars (left vertical axis), and error due to approximation is shown in black dots (right vertical axis) for a range of architectural configurations and degrees of approximation (horizontal axis). Execution time includes both processing (blue bars) and transfer time between the Host CPU and the device (orange bars). Approximation error is independent of the specific architectural configuration. For JPEG and Fisheye we use the error metric $100/PSNR$ (instead of simply $PSNR$) to abide by the concept that in this figure lower is better.

FPGA, respectively with a negligible drop of image quality (PSNR dropping from 37.6dB to 37.2dB). DCT and iDCT kernels have well-defined significance semantics due to the insensitivity of the human eye to high visual frequencies thereby exhibiting a well-behaved performance vs. accuracy curve.

Fisheye shows a steep drop in output quality when we relax the requirement for bicubic interpolation (PSNR dropping from 81dB to 49dB). FPGA execution fares poorly mainly because the memory access pattern for each 4x4 window of pixels is irregular and cannot be bursted out of the external DRAM.

VI. Conclusion

We introduce a programming model and runtime system to support approximate computing on heterogeneous systems. We allow the user to express expertise on the importance of different computations for the quality of the end result, to provide performance-efficient approximate implementations of computations and to control the quality / energy efficiency trade-off at execution time. Also, our runtime eliminates technical concerns when programming a heterogeneous system, such as computation scheduling and data management, which are often a huge programming burden that limit productivity. We evaluated our implementation with realistic applications and found that exploiting the concept of significance at the application level enables measurable performance gains through a combination of approximations and heterogeneity, while the programmer maintains control of the quality of the output.

References

[1] N. Bellas, S. M. Chai, M. Dwyer, and D. Linzmeier. Real-time fisheye lens distortion correction using automatically generated streaming accelerators. In *17th IEEE Symposium on Field Programmable Custom Computing Machines*, FCCM '09, pages 149 – 156, April 2009.

[2] N. Dalal and B. Triggs. Histograms of Oriented Gradients for Human Detection. In *IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '05, pages 886–893, June 2005.

[3] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture Support for Disciplined Approximate Programming. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 301–312, March 2012.

[4] M. Jacobsen and R. Kastner. RIFFA 2.0: A reusable integration framework for FPGA accelerators. In *23rd International Conference on Field programmable Logic and Applications*, FPL '13, pages 1–8, September 2013.

[5] S. Mittal. A Survey of Techniques for Approximate Computing. *ACM Comput. Surv.*, 48(4):62:1–62:33, Mar. 2016.

[6] M. Owaida, N. Bellas, C. D. Antonopoulos, K. Daloukas, and C. Antoniadis. Massively parallel programming models used as hardware description languages: The OpenCL case. In *2011 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2011, San Jose, CA, November 7-10*, pages 326–333.

[7] I. Parnassos, P. Skrimponis, G. Zindros, and N. Bellas. SoCLog: A real-time, automatically generated logging and profiling mechanism for FPGA-based Systems On Chip. In *26th International Conference on Field Programmable Logic and Applications, FPL 2016, Lausanne, Switzerland, August 29 - September 2, 2016*, pages 1–4.

[8] V. Prisacariu and I. Reid. fastHOG - a real-time GPU implementation of HOG. Technical Report 2310/09, Department of Engineering Science, Oxford University, 2009.