Heterogeneous exascale supercomputing: the role of CAD in the exaFPGA project

M. Rabozzi, G. Natale, E. Del Sozzo, A. Scolari, L. Stornaiuolo, M. D. Santambrogio
Politecnico di Milano, DEIB, Italy
{marco.rabozzi, giuseppe.natale, emanuele.delsozzo, alberto.scolari, marco.santambrogio}@polimi.it
luca.stornaiuolo@mail.polimi.it

***Abstract*—Since the end of Moore's law is limiting the growth of general purpose processors, High Performance Processing (HPC) systems are considering FPGA-based accelerators as a promising solution for several application fields. However, their employment poses challenges the research is still tackling, and existing tools and workflows do not naturally adapt to the scale and complexity of HPC domains. To help researchers and practitioners, this paper proposes CAOS, a platform that implements an FPGA development workflow tailored to HPC systems while being open to external contributions. Indeed, researchers and developers can plug into CAOS to experiment and compare their solutions at each step of the design flow. This paper describes the CAOS workflow and validates it against several case studies to assess its generality and highlight possible research contributions.**

I. Introduction

As we are approaching the end of Dennard scaling and Moore's law [1], High Performance Computing (HPC) systems are failing to meet the ever-increasing demand for performance and energy efficiency. This has resulted in a growing need for embracing heterogeneity and leveraging hardware accelerators – such as Graphic Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) – along with Central Processing Units (CPUs). Among the available solutions, FPGAs, thanks to their advancements, currently represent the most promising candidate, offering a compelling trade-off between efficiency and flexibility that is arguably the most beneficial. Indeed, albeit ASICs show the best performance and energy efficiency figure, they are not cost-effective solutions due to the diverse and ever-evolving HPC workloads and the high complexity of their development and deployment, especially for HPC. Programmability is thus an essential feature that accelerators deployed in datacenters have to enjoy, the two most common representative being GPUs and FPGAs. Physical restrictions in terms of power consumption, space and heat dissipation limit GPUs adoption in large-scale systems, even if they were shown to deliver better performance and energy efficiency than CPUs. Conversely, FPGAs can meet these restrictions while providing high performance, energy efficiency and programmability thanks to hardware specialization.

Although the benefits of employing High Performance Reconfigurable Computer (HPRC) systems, based on integrating conventional CPUs and FPGAs, have been already shown [2], the current difficulty in programming FPGAs is a barrier to the widespread adoption of HPRC systems. As of now, highly specialized designers are needed to develop and deploy large-scale HPRC systems and applications, effectively discouraging domain experts or pure software developers. Therefore, it is crucial that all the steps to be performed during development, from the profiling of the application to identify acceleration candidates, to the choice of the suitable *architectural template* to program the FPGA, to the design of the accelerator itself, to the runtime management and the evaluation of the solution obtained, be completely automated when possible or made accessible by heavily assisting the application designer.

As part of the exaFPGA project[1], this paper presents the design flow and the peculiarities of *CAOS (CAD as an Adaptive OpenPlatform Services)*, a framework that aims at providing a fully integrated platform for automating and assisting all the steps for accelerating applications on HPRC systems. This platform also provides practical interfaces to enable extensions and enhancements of its functionalities and promote contribution from the entire community, ultimately pushing the adoption of reconfigurable hardware in HPC.

The rest of the paper is organized as follows. Section II provides the necessary background and motivations of this work. Section III presents the architecture of the CAOS framework and details its components, while Section IV describes the analyzed case studies and their development flow experience within CAOS. Finally, Section V draws the conclusions.

II. Background and motivation

This section reviews the background material at the base of CAOS, highlighting limitations and open challenges that motivate this work. Since HPC systems comprise multiple nodes, the description of nodes' interconnection and resources can be formalized through the concept of *template*, which describes the system organization at a certain granularity. We can distinguish three main templates: a *system template* describes how the nodes are interconnected, a *node template* describes the computational resources each node contains and an *architectural template* describes the computational kernels deployed on the FPGA. Throughout this section we will use these three categories to review the bulk of work around CAOS.

A. Existing workflows for FPGA development

Due to the diversity of domains and vendors, different development practices and tool suites have established to aid developers, yet at the cost of a strong lock-in. As an example, Xilinx provides different tool sets and flows like the Vivado Suite, SDAccel and SDSoC [4], while Altera provides "Quartus Prime", "FPGA SDK for OpenCL" and others for the same domains. For instance, the Vivado Suite is used for custom FPGA development and leaves the user in control of the whole

[1]The exaFPGA project is part of the EXTRA European effort [3]

design process, allowing free choice of the templates (system, node, architectural) but exposing most of the complexity of this process. To help System-on-Chip (SoC) designers, Xilinx's MPSoC simplifies profiling, High Level Synthesis (HLS) and hardware/software interfacing, greatly reducing engineering time and effort; yet, it is applicable only to SoC development and assumes fixed templates at all system levels. Finally, SDAccel allows developing FPGA-based applications at a higher level of abstraction than Vivado, targeting PCIe-attached accelerators. SDAccel eases acceleration by assisting the user during the HLS process and automatizing the subsequent phases, including the generation of hardware/software interfaces and runtime management, but leaves no control over the templates and the system parameters. Another workflow is Dyplo by Topic [5], which targets embedded applications and focuses on simplifying the development of partially reconfigurable solutions, assuming a single-node system with CPU+FPGA; Dyplo uses a fixed architectural template based on a central ring bus for communication, which simplifies runtime reconfiguration. Based on the dataflow model [6], Maxeler Technologies [7] provides integrated solutions for distributed applications with cluster-shared accelerators or tightly coupled CPU+FPGA nodes. The Maxeler tools are based on a custom java-based language (named MaxJ), which automates the synthesis phase and hides many implementation details, fixing the node and system templates on the basis of the workflow initially chosen.

On the academic side, several works also attempt to establish design methodologies, but fall short of the freedom of templates choice and of flexibility. A large body of literature is devoted to enforcing design practices for reconfigurable systems [8, 9] while leaving designers the freedom to choose the architectural template, but usually assume a single-FPGA + CPU node template. On the opposite side, more recent efforts like Darkroom [10] propose a workflow for image and video processing kernels that is based on a Domain Specific Language (DSL), in order to automatize the port of such kernels to FPGA. However, a DSL is too restrictive for the plurality of HPC applications, which have much more complex algorithmic patterns.

B. *Unsolved issues and motivations behind CAOS*

As the research is still struggling with the complexity of the HPC domain and the quest for automated solutions, most HPC practitioners still resort to full-custom designs in order to achieve high performance, and the learning curve remains very steep. The exploration of solutions, practices and architectural paradigms for FPGAs in the HPC domain is likely to continue, and possibly broaden, in the coming years [11, 12, 13], rising two main needs. The first need is to devise a general enough workflow for FPGA acceleration in HPC taking into account the complexity of HPC systems, whose organization is typically hierarchical. This workflow should allow the designer to *choose the organization of the system* at the various levels, corresponding to the three templates introduced at the beginning of this section. The second need is to have a common platform for implementing the solutions the research proposes over time, ensuring *"pluggability" of components*. For example, users willing to showcase profiling techniques or resource estimation algorithms should easily plug into the system and can fairly compare their work

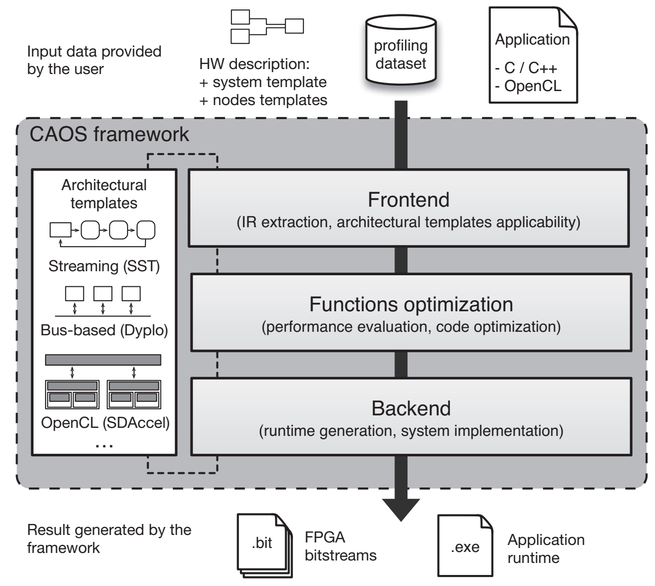


Fig. 1. High level overview of the CAOS framework in terms of: the framework main components, the input data provided by the designer and the final output generated by the framework.

with others. Similarly, contributions to the floorplanning phase should integrate in the proper step of the workflow and receive the information they need from CAOS.

III. Proposed solution and design flow

This section presents the overall architecture of the proposed CAOS framework, starting from a high level description towards a more detailed characterization of its main components. The CAOS framework has been designed around the three key points:

- **Modularity**: ensure separation of concerns among the components of the platform and provide a set of well defined Application Programming Interfaces (APIs) to allow seamless integration of different versions of the analysis tools, design exploration algorithms and performance models leveraged within the framework.
- **Usability**: allow users with low expertise on reconfigurable heterogeneous systems to quickly optimize their applications, analyze the potential performance gain and deploy the final design on the target architecture.
- **Interactivity**: guide the users towards the optimization flow, provide suggestion and error reports at each stage of the process and allow the user to specify or modify the solutions provided by the framework.

The main components of the platform, the inputs provided by the user and the final generated artifacts are presented in Figure 1. The platform allows the user to specify the application logic either in C, C++ or OpenCL. Along with the application code, the platform also requires a HW description, that defines the system template and the nodes templates of the target architecture, and a profiling dataset, used to identify the compute-intensive functions of the application. The flow is organized in three main *components*: the *frontend*, the *functions optimization* component and the *backend*. All the components

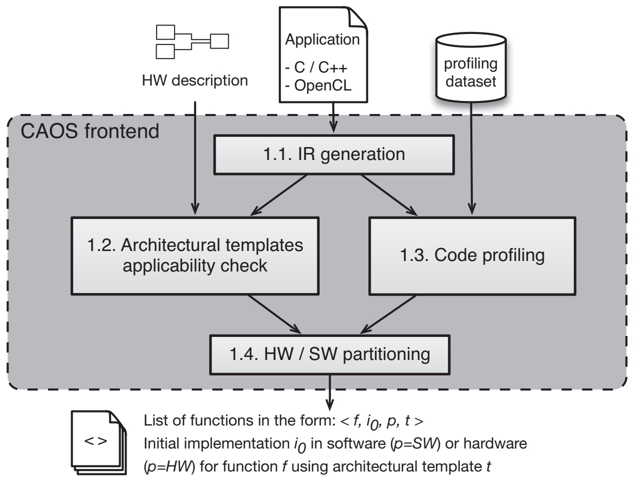

HW description
Application
- C / C++
- OpenCL
profiling dataset
CAOS frontend
1.1. IR generation
1.2. Architectural templates applicability check
1.3. Code profiling
1.4. HW / SW partitioning
< >
List of functions in the form: < f, i0, p, t >
Initial implementation i0 in software (p=SW) or hardware (p=HW) for function f using architectural template t

Fig. 2. Presentation of the CAOS framework frontend flow that starting from the user inputs produces an intermediate description of the application functions. The frontend consists in the following phases: 1.1. IR generation, 1.2. architectural templates applicability check, 1.3. code profiling and 1.4. HW / SW partitioning.

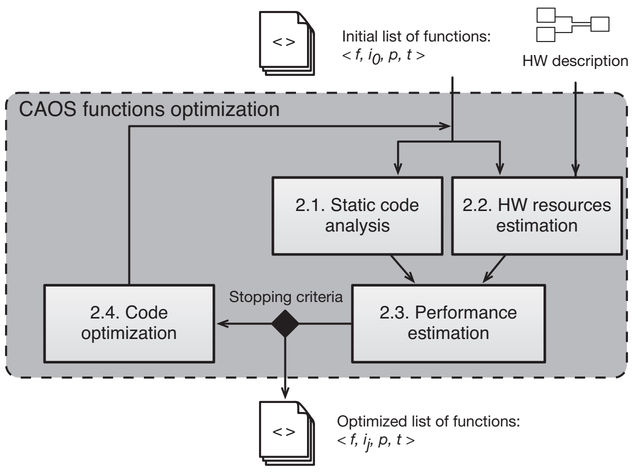

< >
Initial list of functions: < f, i0, p, t >
HW description
CAOS functions optimization
2.1. Static code analysis
2.2. HW resources estimation
2.4. Code optimization
Stopping criteria
2.3. Performance estimation
< >
Optimized list of functions: < f, ij, p, t >

Fig. 3. Representation of the CAOS functions optimization cycle. The list of functions produced by the frontend are analyzed and optimized by means of the following phases: 2.1 static code analysis, 2.2 HW resources estimation, 2.3 performance estimation and 2.4 code optimization.

operate on one or more node architectural templates that specify how the reconfigurable nodes of the target system should be configured. An architectural template specifies both the computation model within the reconfigurable hardware and the specific technology used to implement the model. Within this paper, and specifically in Section IV, we refer to the following architectural templates: Streaming Stencil Time-step (SST) model [14], bus-based model implemented with TOPIC Dyplo interconnect technology [5] and the OpenCL computation model implemented via Xilinx SDAccel [15]. The architectural templates are part of the the platform and cannot be defined directly by the application designer, but additional architectural templates can be integrated within the framework.

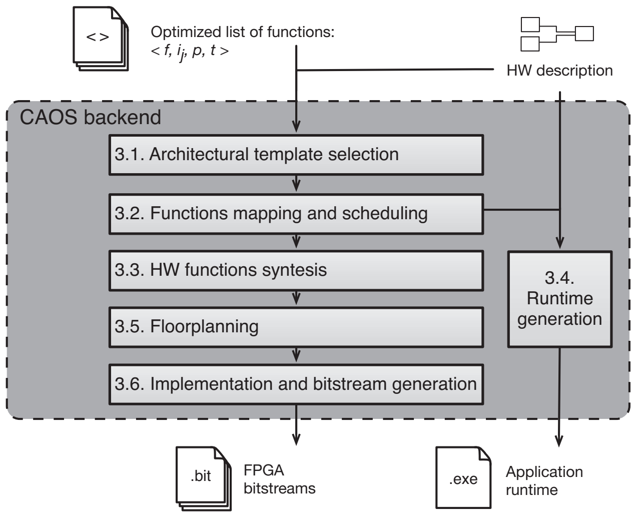
The frontend analyzes the user's code and provides suggestions on the architectural templates that best fit the application, allowing also the user to consider multiple templates for the subsequent phases. The frontend returns a list of functions to be optimized within the functions optimization component. The optimized list of functions and the HW description of the system are fed to the backend component, which produces the final bitstreams for FPGA configuration and the application runtime.

A. *CAOS frontend*

The main phases of the frontend are depicted in Figure 2. Phase 1.1 pre-processes the user's code and produces an IR of the application, which identifies each function in the source code and associates the function identifier to the initial software implementation. The IR and the HW description are then given in input to the architectural templates applicability check (phase 1.2), where the framework analyzes the code to verify which functions fit a specific architectural template. As an example, CAOS checks whether the SST architectural template can be applied to a function by first verifying that the function meets the requirements of polyhedral codes [14]. As for most of the framework phases, this phase is not meant to be fully automatic, and the user can manually select the templates or filter the solutions proposed by the platform. In parallel with phase 1.2, the code is profiled by using the provided dataset or multiple ones (phase 1.3). This phase aims at identifying the most compute-intensive functions of the application, so that phase 1.4, which also knows the applicability of the architectural templates, can identify the candidates for hardware acceleration on specific architectural templates. As for phase 1.2, phase 1.4 allows user to manually specify the HW/SW partitioning, possibly considering the profiling data from phase 1.3. The frontend outputs a list of decorated functions defined by the tuple $< f, i_0, p, t >$, where f is the function identifier, i_0 is its initial implementation, p specifies the implementation type (either HW or SW) and t identifies the target architectural template. For each function f in the original application, multiple tuples with different target architectural templates t can be generated.

B. *CAOS functions optimization*

The function list generated by the frontend is given as input to the CAOS functions optimization component detailed in figure 3. This component guides the designer through the optimization of the function list identified in the frontend. Phase 2.1 performs a static analysis of the code to derive a set of performance metrics (such as the operational intensity [16]), while the parallel phase 2.2 estimates the resource requirements on the reconfigurable hardware, in case this is targeted by the function, using, for example, HLS tools such as Xilinx Vivado HLS. The data from phases 2.1 and 2.2 are used to estimate the final performance of the function and determine the set of optimizations that can be applied to the code (phase 2.3). This phase is not bound to a specific performance model and allows different approaches depending on the architectural template: for example, the Roofline model [16] can be used for the OpenCL and bus-based architectural templates, while the SST template, which has more constraints, allows simpler analytical models along with the Roofline model. Finally, phase 2.4 presents to the user the potential optimizations that can be applied to the code. Depending on the complexity of



<>
Optimized list of functions:
< f, i_j, p, t >
HW description
CAOS backend
3.1. Architectural template selection
3.2. Functions mapping and scheduling
3.3. HW functions syntesis
3.4. Runtime generation
3.5. Floorplanning
3.6. Implementation and bitstream generation
.bit
FPGA bitstreams
.exe
Application runtime

Fig. 4. Description of the CAOS backend that produces the final system implementation and the application runtime. The backend consists in the following phases: 3.1. architectural template selection, 3.2. functions mapping and scheduling, 3.3. HW functions synthesis, 3.4. runtime generation, 3.5. floorplanning and 3.6. implementation and bitstream generation.

the optimization, the framework can either automatically apply it or guide the user towards manual modifications. After phase 2.4, the initial implementation i_0 of a function is updated with its optimized implementation i_j. The optimization cycle then repeats from phases 2.1 and 2.2 until no more optimizations are identified or the required goal (e.g., performance or area) is achieved.

C. CAOS backend

The flow of the CAOS backend is presented in Figure 4. The backend receives the functions list generated by the functions optimization component and the user's HW description. In case multiple architectural templates are considered, the first step of the backend guides the selection of the architectural template for the final system implementation (phase 3.1). Then, phase 3.2 maps and schedules the execution of the HW functions on the reconfigurable logic, possibly consisting of multiple FPGAs. Depending on the selected architectural template, the mapping and scheduling algorithm might also need to take into account partial dynamic reconfiguration [17]. The results of phase 3.2 and the HW description are then used in phase 3.4 to generate the runtime for the system. Simultaneously, the hardware functions are further processed to obtain the final FGPA bitstreams. Phase 3.5 performs HLS and hardware synthesis to generate the functions' netlists. Phase 3.5 provides more accurate estimations of the HW resources and, if partial dynamic reconfiguration is required, a floorplanning algorithm is run to place the reconfigurable regions on the FPGA [18] (phase 3.6). Finally, phase 3.7 performs the place and route of the HW functions and generates the FPGA bitstreams.

IV. CASE STUDIES

This section presents four different case studies that validate the workflow CAOS proposes. In particular, we analyze how each case study fits within the framework components and within the various phases of each component.

A. Smith Waterman

Smith-Waterman performs local sequence alignment between two strings or genetic data sequences [19], and we used CAOS for porting its C implementation to FPGA.

The target system is an Intel 64 bits workstation, and we used both a Xilinx Virtex 7 and a Xilinx Kintex Ultrascale FPGA for accelerating Smith-Waterman; both FPGAs were attached to the host via PCIe.

In the frontend, we selected the OpenCL template implemented through SDAccel. After the IR generation and the analysis of the functions, we provided a dataset of strings in order to profile the input application, and the HW/SW partitioning phase suggested to assign the most compute-intensive stage to hardware implementation; this stage is the update of the algorithm data structures, while the traceback stage still runs on the CPU.

Within the functions optimization, the static code analysis counted the relevant operations like indexing, arithmetic operations and comparisons from the profiling data. At the same time, CAOS estimated the HW resources required by hardware-assigned function. This analysis reported some bandwidth limits and used the Roofline model to evaluate the performance, which started from the operational intensity computed statically to check the memory boundedness of the function for both FPGAs.

To solve the memory bound the tool indicated, we decided to manually compress the inputs and change the code, as the possible DNA nucleotides are only 4.

The HW resources estimation suggested to use the entire area on the FPGA by duplicating the instances of the hardware-assigned function, but, since the limited bandwidth prevented this, CAOS moved to the backend component.

The selected template required SDAccel to perform the last phases. SDAccel allows the user to set parameters such as which part of the global memory is used to store the buffers, which ports send and receive the data and the number of compute units to implement, an finally generates the bitstream and the runtime support.

Table I shows the performance and the energy efficiency of the application, comparing the results of the two FPGAs to the CPU. On Virtex 7 FPGA, the porting achieved a 3.68x performance speedup compared to CPU and a 11.8x improvement in energy efficiency., while the Kintex Ultrascale achieved higher results (14.15x and 45x, respectively) thanks to the higher bandwidth of its PCIe interface.

B. Protein Folding

We used CAOS to generate an optimized hardware implementation of a Protein Folding application, which encodes the physical process that shapes the linear amino-acid sequence of a protein. Our implementation relies on an implementation of the *Ab Initio* Protein Folding Algorithm, based on Monte Carlo simulation [20].

The inputs to the frontend are the C++ application, a set of relevant amino-acid sequences for profiling and the system description, which is a 64 bits machine with a Xilinx Virtex 7 FPGA connected via PCIe.

As in Smith-Waterman case study, we chose the OpenCL template implemented through SDAccel technology. The profiling phase identified the function that computes the pairing energy as the most compute intensive. Hence, this function was the candidate for HW acceleration while the rest of the code remained in software. The template applicability check showed unsupported C++ functionalities that we manually replaced. In the functions optimization component, the static analysis phase counted the number of arithmetical, comparison and indexing operations of the pairing energy function for each input and computed the expected memory traffic between the DRAM and the FPGA as an input to the Roofline model. This model identified the function as compute bound and CAOS proposed to apply loop pipelining and to rearrange data in memory in order to achieve longer memory bursts. Finally, SDAccel completed the backend phases by performing HW synthesis, generating the runtime and the FPGA bitstream. Table I reports the FPGA speedup with respect to the CPU in terms of execution time (1.61x), as well as the energy efficiency improvement (15.29x).

C. Pearson Correlation Coefficient

The following case study is a Brain Network image analysis application that applies Pearson Correlation Coefficient (PCC) [21] (a statistic coefficient that measures the degree of linear correlation between two variables) to infer interconnections between neurons.

In this case, we had already identified the computational kernel of the application, and two different implementations were available for it (one in OpenCL and one in C), as well as two different system descriptions. The former description is equal to Smith-Waterman case study, the latter requires the application to run on a MPSoC. In particular, the target architecture is a *Miami-Florida* board [22].

In terms of architectural templates, this case study could suit both OpenCL and Bus-based templates, according to both the input code and the system descriptions. Since we already explored the phases through the proposed framework for the OpenCL template, we now focus on the Bus-based one.

We skipped most of the frontend phases; indeed, we already knew the most compute intensive function and candidate for hardware acceleration, which was the function in charge of computing the PCC. Given the Bus-based template, we could place multiple instances of this function on the FPGA, and communicate with them by means of TOPIC Dyplo interconnection technology. Differently from other case studies, PCC function instances could not use all the available resources on the FPGA, but they had to fit one or more reconfigurable regions on the FPGA. As a consequence, the functions optimization component had to take into account the resources available on each reconfigurable region, in order to properly tune the code optimizations and not exceed the resources available in the selected regions.

Once functions optimization component was done, CAOS backend was in charge of mapping PCC function instances on the reconfigurable regions, and performing all the steps to the generation of the bitstream. In particular, here CAOS framework produced a static bitstream and a set of partial bitstream, one for each reconfigurable regions and function mapped on them. Finally, the runtime generation phase relied on TOPIC Dyplo APIs in order to manage communication and partial dynamic reconfiguration features.

Table I reports the improvements (in terms of speedup and energy efficiency) of this case study only on Virtex 7 FPGA. Indeed, the goal of PCC implementation using Bus-based template was to take advantage of partial dynamic reconfiguration, therefore, in terms of performance, it was not possible to apply all the available optimizations due to resource constraints. On the other hand, PCC implementation on Virtex 7 FPGA remarkably leveraged CAOS functions optimization phase, hence its performance are more meaningful. We compared such implementation with a pure software implementation parallelized by means of OpenMP libraries. The experimental results report a 3.1x speedup in terms of execution time, as well as a 2.2x improvement in energy efficiency.

D. Iterative Stencil Loops

Iterative Stencil Loops (ISLs) are the core computational kernels of a class of algorithms that can be found in a lot of industrial and scientific applications, from numerical methods to solve partial differential equations, to cellular automata and image and video processing [23]. The computational pattern of ISLs consists in the iterative update of each element of a multi-dimensional array with weighted contributions from a subset of its neighbors in both time and space [23].

In this case study the applications are written in C, while the system description requires the code to be accelerated on a Xilinx Virtex 7 FPGA connected via PCIe to the x64 host processor. Since the computational kernels – i.e. the ISLs – within the applications are already identified, profiling in the frontend phase is skipped. The phase demanded to perform the architectural templates applicability check extracts the code parts from the functions identified by the application designer and tries to derive the corresponding polyhedral representation [14], while also checking for them to be ISLs. As in this case study these steps are successfully performed, the architectural template selected is then the streaming SST [14, 23].

In the functions optimization component, static code analysis is in charge of retrieving the total number of time-steps for each identified ISL, while HW resource estimation is produced by means of Vivado HLS reports for a single SST. The performance estimation is then performed employing an analytical model [14] to derive the maximum lenght of SSTs that can be enqueued in the resulting accelerator, taking into account total amount of available resources.

The backend component then produce the scheduling and the runtime support to load the ISLs accelerators when needed during the execution of the application, and the PCIe interface needed to perform the data exchange. HW synthesis and bitstream generation are then performed by means of Vivado.

Table I provides performance and power efficiency improvements of the best implementation – i.e. the longest queue of SSTs that it was possible to synthesize – for all the ISLs analyzed, with respect to the CPU-only system, equipped with an Intel Xeon E5 processor, where the ISLs have been compiled using Pluto with *diamond tiling*[24], state of the art optimization for the implementation of ISLs on CPU. In both cases, overall power consumption was measured at the wall, which means that the whole system power drain was taken into account. Notice how we yield better power efficiency in all cases (up to 12.9X), while raw performance shows an improvement only in the case of *jacobi-2d*.

TABLE I. EXPERIMENTAL RESULTS

Case Study	Board	Improvement wrt CPU	
		Performance	Energy Efficiency
IV-A	Virtex 7	3.68x	11.8x
IV-A	Kintex	14.15x	45x
IV-B	Virtex 7	1.61x	15.29x
IV-C	Virtex 7	3.1x	2.2x
IV-D jacobi-2d	Virtex 7	1.09x	12.9x
IV-D heat-3d	Virtex 7	0.22x	2.46x

V. CONCLUSIONS

In this paper we presented the design flow of CAOS (CAD as an Adaptive OpenPlatform Service), a platform designed to provide a fully integrated development experience when accelerating an application on reconfigurable hardware. The framework has been conceived to automate or heavily assist all the steps involved in the development flow, being able to provide usability, modularity and interactivity. We then validated the proposed design flow by analyzing four different case studies, and showing how each of them flows through the different framework phases. The final objective of this work is to ultimately push the adoption of reconfigurable hardware in HPC.

ACKNOWLEDGMENTS

The work is part of the EXTRA project. The EXTRA project runs from September 2015 till August 2018 and receives funding from the EU Horizon 2020 research and innovation program under grant No 671653.

REFERENCES

[1] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE, 2011, pp. 365–376.

[2] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell, "The promise of high-performance reconfigurable computing," *Computer*, no. 2, pp. 69–76, 2008.

[3] D. Stroobandt, A. L. Varbanescu, C. B. Ciobanu, M. Al Kadi, A. Brokalakis, G. Charitopoulos, T. Todman, X. Niu, D. Pnevmatikatos, A. Kulkarni *et al.*, "Extra: Towards the exploitation of exascale technology for reconfigurable architectures," in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2016 11th International Symposium on*. IEEE, 2016, pp. 1–7.

[4] Xilinx Inc., "Xilinx developer zone." [Online]. Available: https://www.xilinx.com/products/design-tools.html

[5] Topic Embedded Products, "Dynamic process loader - dyplo." [Online]. Available: https://topicembeddedproducts.com/products/dyplo/

[6] J. B. Dennis, "Data flow supercomputers," *Computer*, vol. 13, no. 11, pp. 48–56, Nov. 1980. [Online]. Available: http://dx.doi.org/10.1109/MC.1980.1653418

[7] Maxeler Technologies, "Maxeler technologies website." [Online]. Available: http://maxeler.com/#/

[8] A. Panella, M. D. Santambrogio, F. Redaelli, F. Cancare, and D. Sciuto, "A design workflow for dynamically reconfigurable multi-fpga systems," in *2010 18th IEEE/IFIP International Conference on VLSI and System-on-Chip*, Sept 2010, pp. 414–419.

[9] M. D. Santambrogio, D. Pnevmatikatos, K. Papadimitriou, C. Pilato, G. Gaydadjiev, D. Stroobandt, T. Davidson, T. Becker, T. Todman, W. Luk, A. Bonetto, A. Cazzaniga, G. C. Durelli, and D. Sciuto, "Smart technologies for effective reconfiguration: The faster approach," in *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, July 2012, pp. 1–7.

[10] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: Compiling high-level image processing code into hardware pipelines," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 144:1–144:11, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2601097.2601174

[11] M. Araya-Polo, J. Cabezas, M. Hanzich, M. Pericas, F. Rubio, I. Gelado, M. Shafiq, E. Morancho, N. Navarro, E. Ayguade, J. M. Cela, and M. Valero, "Assessing accelerator-based hpc reverse time migration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 147–162, Jan 2011.

[12] C. Tomas, L. Cazzola, D. Oriato, O. Pell, D. Theis, G. Satta, and E. Bonomi, "Acceleration of the anisotropic pspi imaging algorithm with dataflow engines," in *Proceedings of 82nd Annual Meeting and International Exposition of the Society of Exploration Geophysics-SEG, Las Vegas - Nevada*, 2012. [Online]. Available: http://publications.crs4.it/pubdocs/2012/TCOPTSB12

[13] P. Sundararajan, "High performance computing using fpgas," *Xilinx White Paper: FPGAs*, pp. 1–15, 2010.

[14] G. Natale, G. Stramondo, P. Bressana, R. Cattaneo, D. Sciuto, and M. D. Santambrogio, "A polyhedral model-based framework for dataflow implementation on fpga devices of iterative stencil loops," in *Proceedings of the 35th International Conference on Computer-Aided Design*. ACM, 2016, p. 77.

[15] L. Wirbel, "Xilinx sdaccel: a unified development environment for tomorrows data center," Technical Report, The Linley Group Inc, Tech. Rep., 2014.

[16] B. da Silva, A. Braeken, E. H. D'Hollander, and A. Touhafi, "Performance modeling for fpgas: extending the roofline model with high-level synthesis tools," *International Journal of Reconfigurable Computing*, vol. 2013, p. 7, 2013.

[17] E. A. Deiana, M. Rabozzi, R. Cattaneo, and M. D. Santambrogio, "A multiobjective reconfiguration-aware scheduler for fpga-based heterogeneous architectures," in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2015, pp. 1–6.

[18] M. Rabozzi, G. C. Durelli, A. Miele, J. Lillis, and M. D. Santambrogio, "Floorplanning automation for partial-reconfigurable fpgas via feasible placements generation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. PP, no. 99, pp. 1–14, 2016.

[19] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[20] R. A. Abagyan and M. Totrov, "Ab initio folding of peptides by the optimal-bias monte carlo minimization procedure," *Journal of Computational Physics*, vol. 151, no. 1, pp. 402–421, 1999.

[21] K. Pearson, "Note on regression and inheritance in the case of two parents," *Proceedings of the Royal Society of London*, vol. 58, pp. 240–242, 1895. [Online]. Available: http://www.jstor.org/stable/115794

[22] "Topic: Miami-florida board." [Online]. Available: https://topicembeddedproducts.com/products/boards-kits/

[23] R. Cattaneo, G. Natale, C. Sicignano, D. Sciuto, and M. D. Santambrogio, "On how to accelerate iterative stencil loops: a scalable streaming-based approach," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, p. 53, 2016.

[24] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 40.