# Chapter 8
# CAOS: CAD as an Adaptive Open-Platform Service for High Performance Reconfigurable Systems

**Marco Rabozzi**

**Abstract** The increasing demand for computing power in fields such as genomics, image processing and machine learning is pushing towards hardware specialization and heterogeneous systems in order to keep up with the required performance level at sustainable power consumption. Among the available solutions, Field Programmable Gate Arrays (FPGAs), thanks to their advancements, currently represent a very promising candidate, offering a compelling trade-off between efficiency and flexibility. Despite the potential benefits of reconfigurable hardware, one of the main limiting factor to the widespread adoption of FPGAs is complexity in programmability, as well as the effort required to port software solutions to efficient hardware-software implementations. In this chapter, we present CAD as an Adaptive Open-platform Service (CAOS), a platform to guide the application developer in the implementation of efficient hardware-software solutions for high performance reconfigurable systems. The platform assists the designer from the high-level analysis of the code, towards the optimization and implementation of the functionalities to be accelerated on the reconfigurable nodes. Finally, CAOS is designed to facilitate the integration of external contributions and to foster research on Computer Aided Design (CAD) tools for accelerating software applications on FPGA-based systems.

## 8.1 Introduction

Over the last 40 years, software performance has benefited from the exponential improvement of General Purpose Processors (GPPs) that resulted from a combination of architectural and technological enhancements. Despite such achievements, the performance measured on standard benchmarks in the last 3 years only improved

M. Rabozzi (✉)
Dipartimento di elettronica, informazione e bioingegneria, Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133 Milano, Italy
e-mail: marco.rabozzi@polimi.it

B. Pernici (ed.), *Special Topics in Information Technology*, PoliMI SpringerBriefs,
https://doi.org/10.1007/978-3-030-32094-2_8

at a rate of about 3% per year [11]. Indeed, after the failure of Dennard scaling [21], the current diminishing performance improvements of GPP reside in the difficulty to efficiently extract more fine-grained and coarse-grained parallelism from software. Considering the shortcomings of GPP, in current years we are assisting at a new era of computer architectures in which the need for energy-efficiency is pushing towards hardware specialization and the adoption of heterogeneous systems. This trend is also reflected in the High Performance Computing (HPC) domain that, in order to sustain the ever-increasing demand for performance and energy efficiency, started to embrace heterogeneity and to consider hardware accelerators such as Graphics Processing Units (GPUs), FPGAs and dedicated Application-Specific Integrated Circuits (ASICs) along with standard CPU. 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. Among the available solutions, FPGAs, thanks to their advancements, currently represent the most promising candidate, offering a compelling trade-off between efficiency and flexibility. Indeed, FPGAs are becoming a valid HPC alternative to GPUs, as they provide very high computational performance with superior energy efficiency by employing customized datapaths and thanks to hardware specialization. FPGA devices have also attained renewed interests in recent years as hardware accelerators within the cloud domain. The possibility to access FPGAs as on-demand resources is a key step towards the democratization of the technology and to expose them to a wide range of potential domains [2, 6, 24].

Despite the benefits of embracing reconfigurable hardware in both the HPC and cloud contexts, we notice that one of the main limiting factor to the widespread adoption of FPGAs is complexity in programmability as well as the effort required to port a pure software solution to an efficient hardware-software implementation targeting reconfigurable heterogeneous systems [1]. During the past decade, we have seen significant progress in High-Level Synthesis (HLS) tools which partially mitigate this issue by allowing to translate functions written in a high-level language such as C/C++ to a hardware description language suitable for hardware synthesis. Nevertheless, current tools still require experienced users in order to achieve efficient implementations. In most cases indeed, the proposed workflows require the user to learn the usage of specific optimization directives [22], code rewriting techniques and, in other cases, to master domain specific languages [10, 13]. In addition to this, most of the available solutions [9, 10, 13] focus on the acceleration of specific kernel functions and leave to the user the responsibility to explore hardware/software partitioning as well as to identify the most time-consuming functions which might benefit the most from hardware acceleration. To tackle these challenges, we propose the CAOS platform bringing the following contributions:

- A comprehensive design flow guiding the designer from the initial software to the final implementation to a high performance FPGA-based system.
- Well-defined Application Programming Interfaces (APIs) and an infrastructure allowing researchers to integrate and test their own modules within CAOS.

- A general method for translating high-level functions into FPGA-accelerated kernels by matching software functions to appropriate architectural templates.
- Support for three different architectural templates allowing to target software functions with different characteristics within CAOS.

Section 8.2 describes the overall CAOS platform, its design flow and infrastructure, while Sect. 8.3 presents an overview of the supported architectural. Section 8.4 discuss the experimental results on case studies targeting different architectural templates. Finally, Sect. 8.5 draws the conclusions.

## 8.2 The CAOS Platform

The CAOS platform has been developed in the context of the Exploiting eXascale Technology with Reconfigurable Architectures (EXTRA) project and shares with it the same vision [19]. CAOS targets both application developers and researches while its design has been conceived focusing on three key principles: usability, interactivity and modularity. From a usability perspective, the platform supports application designers with low expertise on reconfigurable heterogeneous systems in quickly optimizing their code, analyzing the potential performance gain and deploying the resulting application on the target reconfigurable architecture. Nevertheless, the platform does not aim to perform the analysis and optimizations fully automatically, but instead interactively guides the users towards the design flow, providing suggestion and error reports at each stage of the process. Finally, CAOS is composed of a set of independent modules accessed by the CAOS flow manager that orchestrates the execution of the modules according to the current stage of the design flow. Each module is required to implement a set of well-defined APIs so that external researchers can easily integrate their implementations and compare them against the ones already offered by CAOS.

### *8.2.1 CAOS Design Flow*

The platform expects the application designer to provide the application code written in a high-level language such as C/C++, one or multiple datasets to be used for code profiling and a description of the target reconfigurable system. In order to narrow down and simplify the set of possible optimizations and analysis that can be performed on a specific algorithm, CAOS allows the user to accelerate its application using one of the available architectural templates. An *architectural template* is a characterization of the accelerator both in terms of its computational model and the communication with the off-chip memory. As a consequence, an architectural template constrains the architecture to be implemented on the reconfigurable hardware and poses restrictions on the application code that can be accelerated, so that the

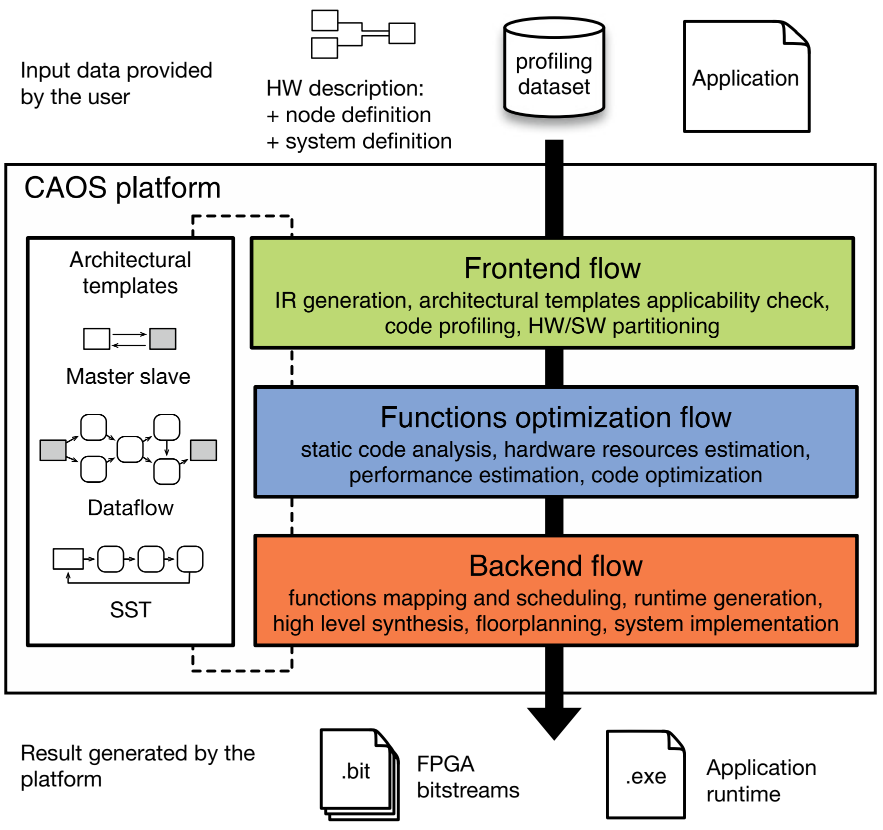
Input data provided by the user
HW description:
+ node definition
+ system definition
profiling dataset
Application
CAOS platform
Architectural templates
Master slave
Dataflow
SST
Frontend flow
IR generation, architectural templates applicability check, code profiling, HW/SW partitioning
Functions optimization flow
static code analysis, hardware resources estimation, performance estimation, code optimization
Backend flow
functions mapping and scheduling, runtime generation, high level synthesis, floorplanning, system implementation
Result generated by the platform
.bit
FPGA bitstreams
.exe
Application runtime

**Fig. 8.1** High-level overview of the CAOS platform. The overall design flow can be divided in three main parts: the *frontend*, the *functions optimization* and the *backend* flow. The application code, datasets to profile the application and an HW description constitute the input data provided by the designer. The final outputs generated by the platform are the bitstreams, that the user can use to configure the FPGAs, and the application runtime, needed to run the optimized version of the application

number and types of optimizations available can be tailored for a specific type of implementation. Furthermore, CAOS is meant to be orthogonal and build on top of tools that perform High-Level Synthesis (HLS), place and route and bitstream generation. Code transformations and optimizations are performed at the source code level while each architectural template has its own requirements in terms of High-Level Synthesis (HLS) and hardware synthesis tools to use.

As shown in Fig. 8.1, the overall CAOS design flow is subdivided into three main flows: the frontend flow, the function optimization flow and the backend flow. The main goal of the frontend is to analyze the application provided by the user, match the application against one or more architectural templates available within the platform, profile the user application against the user specified datasets and, finally, guide

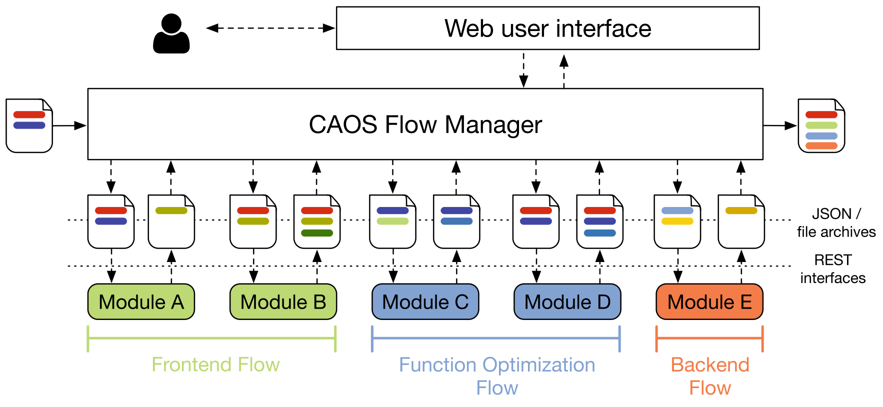

**Fig. 8.2** CAOS infrastructure in terms of its main components and their interaction

the user through the hardware/software partitioning of the application to define the functions of the application that should be implemented on the reconfigurable hardware. The function optimization flow performs static analysis and hardware resource estimation of the functionalities to be accelerated on the FPGA. Such analyses are dependent upon the considered architectural template and the derived information are used to estimate the performance of the hardware functions and to derive the optimizations to apply (such as loop pipelining, loop tiling and loop unrolling). After one or more iterations of the function optimization flow, the resulting functions are given to the backend flow in which the desired architectural template for implementing the system is selected and the required High-Level Synthesis (HLS) and hardware synthesis tools are leveraged to generate the final FPGA bitstreams. Within the backend, CAOS takes care of generating the host code for running the FPGA accelerators and optionally guides the place and route tools by floorplanning the system components.

## 8.2.2 *CAOS Infrastructure*

In order to simplify the adoption of the platform while being open to contributions, the CAOS infrastructure is organized as a microservices architecture which can be accessed by application designers through a web user interface. The infrastructure, shown in Fig. 8.2, leverages on Docker [8] application containers to isolate the modules and to provide scalability. Each module is deployed in a single container, and several implementations of the same module can coexist to provide different functionalities to different users. Moreover, each module's implementation can be replicated to scale horizontally depending on system load. The modules are connected together and driven by the *CAOS Flow manager* which serves the User Interface (UI) and provides the glue logic that routes each request to the proper module.

The interaction between the flow manager and the CAOS modules is performed by means of data transfer objects defined with a JSON Domain Specific Language (DSL). The user can specify at each phase the desired module implementation and the platform will take care of routing the request to the proper module automatically. Moreover, the platform supports modules deployed remotely by simply specifying their IP address. Another advantage of the proposed infrastructure is that, thanks to Docker containers, it can also be easily deployed on cloud instances, possibly featuring FPGA boards, such as the Amazon EC2 F1 instances. This allows a complete design process in the cloud in which the user can optimize the application through a web UI, while the final result of the CAOS design flow can be directly tested and run on the same cloud instance. CAOS supports the integration of new implementations of the modules described in Sect. 8.2.1. Researchers are free to adopt the preferred tools and programming languages that fit their needs, as long as the module provides the implementation of the REST APIs prescribed by the CAOS flow manager.

## 8.3 Architectural Templates

The core idea for devising efficient FPGA-based implementations in CAOS revolves around matching software functions to an architectural template suitable for its acceleration. CAOS currently supports three architectural templates: Master/Slave, Dataflow and Streaming architectural templates. In the next sections, we provide an overview of the templates, describing the supported software functions and hardware platforms, the proposed optimizations and the tools on which the templates rely.

### *8.3.1 Master/Slave Architectural Template*

The Master/Slave architectural template [7] targets systems with a shared Double Data Rate (DDR) memory that can be accessed both by the host running the software portion of the application and by the FPGA devices on which we implement the accelerated functionalities (also referred as kernel). The template also requires that the communication between the accelerator and the DDR memory is performed via memory mapped interfaces. Such requirements allow to standardize the data transfer as well as to support random memory accesses to pointer arguments of the target C/C++ function. Currently, the template supports two target systems: Amazon F1 instances in the cloud and Xilinx Zynq System-on-Chips (SoCs).

The generality of the communication model of the Master/Slave architectural template allows supporting a wide range of C/C++ functions. In particular, the function has to abide to quite general constraints for High-Level Synthesis (HLS) such as no dynamic creation of objects/arrays, no syscalls and no recursive function calls. The Master/Slave architectural template currently leverages on Vivado HLS [23] for the High-Level Synthesis (HLS) of C/C++ code to Hardware Description Language (HDL). Hence, in the CAOS frontend, we verify the applicability of the template to a given function by running Vivado HLS on it and verify that no errors are generated. In addition to the Vivado HLS constraints, we also require the size of the function arguments to be known. This is needed by CAOS to properly estimate the kernel performance throughout the function optimizations flow. During the CAOS functions optimization flow, the template performs static code analysis by leveraging on custom Low-Level Virtual Machine (LLVM) [12] passes. In particular, it identifies loop nests with their trip counts, local arrays and information on the input and output function arguments. Furthermore, the template collects hardware and performance estimations of the current version of the target function directly from Vivado HLS. Such information is then used to identify the next candidate optimizations among loop pipelining, loop unrolling, on-chip caching and memory partitioning. The user can then either select the suggested optimization or conclude the optimization flow if he/she is satisfied with the estimated performance. After having optimized the kernel, the design proceeds to the CAOS backend flow. Here the optimized C/C++ function is translated to HDL and, according to the target system, the template leverages either on the Xilinx SDAccel toolchain [22] or Xilinx Vivado [23] for the implementation and bitstream generation. In both cases, CAOS takes care of modifying the original application and inserts the necessary code and APIs calls to offload the computation of the original software function to the generated FPGA accelerator.

### *8.3.2 Dataflow Architectural Template*

The dataflow architectural template trades off the generality of codes supported by the Master/Slave architectural template in order to achieve higher performance. In a dataflow computing model, the data is streamed from the memory directly to the chip containing an array of Processing Elements (PEs) each of which is responsible for a single operation of the algorithm. The data flow from a PE to the next one in a statically defined directed graph, without the need for any kind of control mechanism. In such a model, each PE performs its operation as soon as the input data is available and forwards the result to the next element in the network as soon as it is computed.

The target system for the dataflow architectural template consists in a host CPU and the dataflow accelerator deployed on a FPGA connected via PCIe to the host. Both the host CPU and the FPGA logic have access to the host DDR memory containing the input/output data. The FPGA accelerator is organized internally as a Globally Asynchronous Locally Synchronous (GALS) architecture divided into the actual accelerated kernel function and a manager. The manager handles the asynchronous communication between the host and the accelerator, whereas the kernel is internally organized as a set of synchronous PEs that perform the computation in parallel.

The architectural template leverages on the OXiGen toolchain [18] and its extension [17] to translate C/C++ functions into optimized dataflow kernels defined with the MaxJ language. In order to efficiently perform the translation from sequential C/C++ code to a dataflow representation, the target function has to satisfy certain requirements detailed in [18]. An exemplary code supported by the template is shown in Listing 8.1. The code requirements are validated in the CAOS frontend flow in order to identify functions that can be optimized with the dataflow architectural template. Within the CAOS function optimization flow, OXiGen performs the dataflow graph construction directly from the LLVM Intermediate Representation (IR) of the target function. Nevertheless, the initial dataflow design might either exceed or underutilize the available FPGA resources. Hence, in order to identify an optimal implementation that fits within the FPGA resources and available data transfer bandwidth, OXiGen supports loop rerolling, to reduce resource consumption, and vectorization, to replicate the processing elements in order to fully utilize the available bandwidth and compute resources. In order to derive the best implementation, OXiGen relies on resource and performance models and runs a design space exploration using an approach based on Mixed-Integer Linear Programming (MILP). Once having generated an optimized kernel, the CAOS backend runs MaxCompiler [13] to synthesize the MaxJ code generated by OXiGen to a Maxeler DFE (Dataflow Engine) that can be accessed by the host system.

**Listing 8.1** An exemplary code supported by the dataflow template. The function takes as input a combination of array types and scalar types. The outer loops iterate over the outer dimension of the array types which are translated as streams. Accesses to the streams are linear with constant offsets. The function can have a combination of nesting levels iterating over the inputs or local variables.

```
void foo(type_1* in_1, type_1* in_2, type_2* out_1, int iter) {
    type_1 tmp_vect [15];
    for(int i = const_1; i < iter - const_2; i++) {

        ...statements...

        for(int j = const_3; j < 15; j++)
            tmp_vect[j] = ...expression... ;

        type_1 tmp_scalar = ...expression... ;

        for(int j = const_3; j < 15; j++)
            tmp_scalar = tmp_scalar + tmp_vect[j];
    }
}
```

## 8.3.3 Streaming Architectural Template

Iterative Stencil Loops (ISLs) represent a class of algorithms that are highly recurrent in many HPC applications such as differential equation solving or scientific simulations. The basic structure of ISLs is depicted in Algorithm 8.1; the outer loop iterates for a given number of times, so-called *time-steps*, while, at each time-step, the inner

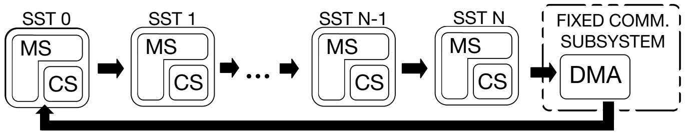

**Fig. 8.3** Architecture of an SST-based accelerator for ISL

loop updates each value of the $n$-dimensional input vector by means of the *stencil* function, computing a weighted sum of the neighbor values in the vector.

**Algorithm 8.1** Generic ISL Algorithm.

```
for $t \leq TimeSteps$ do
    for all points $p$ in matrix $M$ do
        $p \leftarrow stencil(p)$
```

The Streaming architectural template specifically targets stencil codes written in C and leverages the SST architecture proposed by [3] for its implementation. The architectural template of the SST-based accelerator [14] is depicted in Fig. 8.3. The basic SST module performs the computation of a single time-step and is conceptually separated in a *memory system*, responsible for storage and data movement, and a *computing system*, that performs the actual computation. The SST module is designed in order to operate in a streaming mode on the various elements of the input vector; this internal structure is derived by means of the *polyhedral analysis* that allows refactoring the algorithm to optimize on-chip memory resource consumption and implement a dataflow model of computation. Then, the complete SST-based architecture is obtained by replicating $N$ times the basic module to implement a pipeline, where each module computes in streaming a single time-step of the outer loop of the algorithm. Such a pipeline is finally connected with a fixed communication subsystem interfacing with the host machine. Within this context, CAOS offers a design exploration algorithm [20] that jointly maximizes the number of SST processors that can be instantiated on the target FPGA and identifies a floorplan of the design that minimizes the inter-component wire-length in order to allow implementing the system at high frequency. In the frontend, CAOS identifies those functions having the ISL structure shown in Algorithm 8.1, while the CAOS function optimization flow runs the design space exploration algorithm detailed in [20] on the function to accelerate. The approach starts by generating an initial version of the system consisting of a single SST [14] to obtain an initial resource estimate. Subsequently, it solves a maximum independent set problem formulated as an Integer Linear Programming (ILP) to identify the maximum number of SST as well as their floorplan and solves a Traveling Salesman Problem (TSP) to identify the best routing among SSTs. Finally, the CAOS backend generates the accelerator bitstream through Xilinx Vivado while enforcing the identified floorplanning constraints.

## 8.4 Experimental Results

Within this section we discuss the results achieved by CAOS on different case studies targeting the discussed architectural templates. The first case study we consider is the N-Body simulation, which is a well known problem in physics having applications in a wide range of fields. In particular, we focused on the *all-pair* method: the algorithm alternates a computationally intensive force computation step, in which the pairwise forces between each pair of bodies are computed and a position update step, that updates the velocities and positions of the bodies according to the current resulting forces. CAOS, after profiling and analyzing the application in the frontend, properly identified the force computation as the target function to accelerate and matched it to the Master/Slave architectural template. As we can see from Table 8.1, the CAOS implementation targeting an Amazon F1 instance greatly outperforms, both in terms of performance and energy efficiency, the software implementation from [5] running in parallel on 40 threads on an Intel Xeon E5-2680 v2 and the implementation from [4] on a Xilinx VC707 board. Nevertheless, the bespoke implementation from [5] targeting the same hardware provides 11% higher performance than the CAOS one. However, the CAOS design was achieved semi-automatically in approximately a day of work, while the design from [5] required several weeks of manual effort.

As a second test case, we consider the Curran approximation algorithm [16] for calculating the pricing of Asian options. More specifically, we tested two flavors of the algorithm using 30 and 780 averaging points. Within CAOS, we started from a C implementation and we targeted a host machine featuring an Intel(R) Core(TM) i7-6700 CPU @ 3.40 GHz connected via PCIe gen1 x8 to a MAX4 Galava board equipped with an Altera Stratix V FPGA. Since the algorithms operate on subsequent independent data items, the overall computations are easily expressed in C using the structure in Listing 8.1. Hence CAOS identified and optimized the computations leveraging on the dataflow architectural template. As the initial designs did not fit within the device, CAOS applied the rerolling optimization for both cases. As shown in Table 8.2, both CAOS implementations achieve speedups over 100x against the single thread execution on the host system only. Moreover, we compared

**Table 8.1** Performance and energy efficiency of the all-pairs N-Body algorithm accelerated via CAOS and the results achieved by bespoke designs proposed in [4, 5]

| Reference | Platform | Type | Frequency (MHz) | Performance (MPairs/s) | Performance/Power (MPairs/s/W) |
|---|---|---|---|---|---|
| [5] | Intel Xeon E5-2680 v2 | CPU | – | 2,642 | 22.98 |
| [4] | Xilinx VC707 | FPGA | 100 | 2,327 | 116.36 |
| [5] | Xilinx VU9P | FPGA | 154 | 13,441 | 672.06 |
| CAOS | Xilinx VU9P | FPGA | 126 | 12,072 | 603.61 |

**Table 8.2** Results achieved by CAOS on the Curran approximation algorithm with 30 and 780 averaging points compared against CPU and the bespoke FPGA-based implementations from [15]

| Averaging points | Rerolling factor | Speedup w.r.t. CPU | Speedup w.r.t. [15] | Input bandwidth (MByte/s) | Output bandwidth (MByte/s) |
|---|---|---|---|---|---|
| 30 | 4 | 118.4x | 1.23x | 1,767.64 | 196.40 |
| 780 | 98 | 101.0x | 0.5x | 75.11 | 8.35 |

**Table 8.3** Results achieved by the CAOS streaming architectural template compared to [14]

| Algorithm | #SSTs | | Design frequency (MHz) | | Performance improvement w.r.t. [14] (%) | Design time reduction w.r.t. [14] |
|---|---|---|---|---|---|---|
| | CAOS | [14] | CAOS | [14] | | |
| Jacobi2D | 90 | 88 | 228 | 206 | 13.20 | 15.84x |
| Heat3D | 25 | 25 | 228 | 206 | 10.68 | 1.69x |
| Seidel2D | 19 | 19 | 183 | 183 | 0 | 1.08x |

the results against the DFE execution times reported from the designs in [15]. For the version with 30 averaging points, we achieved a speedup of 1.23x. For the version with 780 averaging points, our implementation shows a speed down of about 0.5x. Nevertheless, it was obtained in less than a day of work.

As a final test case, we evaluated the streaming architectural template on three representative ISL computations (Jacobi2D, Heat3D and Seidel2D) targeting a Xilinx Virtex XC7VX485T device [20]. Table 8.3 reports the performance improvement and the design time reduction compared to the methodology in [14]. Thanks to FPGA floorplanning we are able to increase the target frequency for the Jacobi2D and Heat3D algorithms of approximately 11%. Additionally, for the Jacobi2D case, the floorplanner is also able to allocate two additional SSTs improving the performance up to 13%. Nevertheless, the Seidel2D algorithm does not provide the same improvement figure. Indeed, since the total number of SSTs that can be placed into the design is small, the floorplanning reduces its impact on the overall design by leaving more room to the place and route algorithm. Regarding the design time, our approach allows to greatly reduce the number of trial synthesis required, thus leading to an execution time saving of 15.84x for Jacobi2D.

## 8.5 Conclusions

In this chapter we presented CAOS, a platform whose main objective is to improve productivity and simplify the design of FPGA-based accelerated systems, starting from pure high-level software implementations. Currently, the slowing rate at which general purpose processors improve performance is strongly pushing towards specialized hardware. We expect FPGAs to have a more prominent role in the upcoming years as a technology to achieve efficient and high performance solutions both in the HPC and cloud domains. Hence, by embracing this idea, we designed CAOS in a modular fashion, providing well-defined APIs that allow external researchers to integrate extensions or different implementations of the modules within the platform. Indeed, the second, yet not less important, objective of CAOS, is to foster research on tools and methods for accelerating software on FPGA-based architectures.

## References

1. Bacon DF, Rabbah R, Shukla S (2013) FPGA programming for the masses. Commun ACM 56(4):56–63
2. Cardamone S, Kimmitt JR, Burton HG, Thom AJ (2018) Field-programmable gate arrays and quantum Monte Carlo: power efficient co-processing for scalable high-performance computing. arXiv:1808.02402
3. Cattaneo R, Natale G, Sicignano C, Sciuto D, Santambrogio MD (2016) On how to accelerate iterative stencil loops: a scalable streaming-based approach. ACM Trans Archit Code Optim (TACO) 12(4):53
4. Del Sozzo E, Di Tucci L, Santambrogio MD (2017) A highly scalable and efficient parallel design of n-body simulation on FPGA. In: 2017 IEEE international parallel and distributed processing symposium workshops (IPDPSW), pp 241–246. IEEE
5. Del Sozzo E, Rabozzi M, Di Tucci L, Sciuto D, Santambrogio MD (2018) A scalable FPGA design for cloud n-body simulation. In: 2018 IEEE 29th international conference on application-specific systems, architectures and processors (ASAP), pp 1–8. IEEE
6. Di Tucci L, O'Brien K, Blott M, Santambrogio MD (2017) Architectural optimizations for high performance and energy efficient Smith-Waterman implementation on FPGAs using OpenCL. In: 2017 design, automation and test in Europe conference and exhibition (DATE), pp 716–721. IEEE
7. Di Tucci L, Rabozzi M, Stornaiuolo L, Santambrogio MD (2017) The role of CAD frameworks in heterogeneous FPGA-based cloud systems. In: 2017 IEEE international conference on computer design (ICCD), pp 423–426. IEEE
8. Docker. https://www.docker.com
9. Fort B, Canis A, Choi J, Calagar N, Lian R, Hadjis S, Chen YT, Hall M, Syrowik B, Czajkowski T et al (2014) Automating the design of processor/accelerator embedded systems with legup high-level synthesis. In: 2014 12th IEEE international conference on embedded and ubiquitous computing (EUC), pp 120–129. IEEE
10. Hegarty J, Brunhaver J, DeVito Z, Ragan-Kelley J, Cohen N, Bell S, Vasilyev A, Horowitz M, Hanrahan P (2014) Darkroom: compiling high-level image processing code into hardware pipelines. ACM Trans Graph 33(4):144:1–144:11. https://doi.org/10.1145/2601097.2601174
11. Hennessy JL, Patterson DA (2017) Computer architecture: a quantitative approach. Elsevier, Amsterdam
12. Lattner C (2008) LLVM and Clang: next generation compiler technology. In: The BSD conference, pp 1–2
13. Maxeler Technologies: MaxCompiler. https://www.maxeler.com
14. Natale G, Stramondo G, Bressana P, Cattaneo R, Sciuto D, Santambrogio MD (2016) 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, p 77. ACM

15. Nestorov AM, Reggiani E, Palikareva H, Burovskiy P, Becker T, Santambrogio MD (2017) A scalable dataflow implementation of Curran's approximation algorithm. In: 2017 IEEE international parallel and distributed processing symposium workshops (IPDPSW), pp 150–157. IEEE
16. Novikov A, Alexander S, Kordzakhia N, Ling T (2016) Pricing of Asian-type and basket options via upper and lower bounds. arXiv:1612.08767
17. Peverelli F, Rabozzi M, Cardamone S, Del Sozzo E, Thom AJ, Santambrogio MD, Di Tucci L (2019) Automated acceleration of dataflow-oriented c applications on FPGA-based systems. In: 2019 IEEE 27th annual international symposium on field-programmable custom computing machines (FCCM), pp 313–313. IEEE
18. Peverelli F, Rabozzi M, Del Sozzo E, Santambrogio MD (2018) OXiGen: a tool for automatic acceleration of c functions into dataflow FPGA-based kernels. In: 2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW), pp 91–98. IEEE
19. Rabozzi M, Brondolin R, Natale G, Del Sozzo E, Huebner M, Brokalakis A, Ciobanu C, Stroobandt D, Santambrogio MD (2017) A CAD open platform for high performance reconfigurable systems in the extra project. In: 2017 IEEE computer society annual symposium on VLSI (ISVLSI), pp 368–373. IEEE
20. Rabozzi M, Natale G, Festa B, Miele A, Santambrogio MD (2017) Optimizing streaming stencil time-step designs via FPGA floorplanning. In: 2017 27th international conference on field programmable logic and applications (FPL), pp 1–4. IEEE
21. Taylor MB (2013) A landscape of the new dark silicon design regime. IEEE Micro 33(5):8–19
22. Wirbel L (2014) Xilinx SDAccel: a unified development environment for tomorrow's data center. Technical report, The Linley Group Inc.
23. Xilinx Inc.: Vivado design suite. http://www.xilinx.com/products/design-tools/vivado.html
24. Zhang C, Li P, Sun G, Guan Y, Xiao B, Cong J (2015) Optimizing FPGA-based accelerator design for deep convolutional neural networks. In: Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays, pp 161–170. ACM