Deliverable 6.3

Guidelines for Efficient Programming of Heterogeneous HPC Systems

2023-11-30

Work package:	WP6	
Author(s):	Albert Kahira	FZJ
	Manuel Arenaz	APP
	Gino Perna	ES
Reviewer #1	Matthias Balzer	FRAUN
Reviewer #2	Dimitris Theodoropoulos	ICCS
Dissemination Level	Public	
Nature	Report	

Date	Author(s)	Comments	Version	Status
2023-09-28	Giovanni Isotton	Draft M3E contribution	0.1	Draft
2023-10-31	Olivier Michel	CYB contribution	1.0	Final
2023-11-06	Manuel Arenaz	Codee contribution	1.1	
2023-11-15	Gino Perna	EnginSoft contribution		
2023-11-29	Manuel Arenaz	Codee contribution		

This project has received funding from the European High-Performance Computing Joint Undertaking Joint Undertaking (JU) under grant agreement No 955739. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Greece, Germany, Italy, Netherlands,Spain,Switzerland.

Executive Summary

The OPTIMA project demonstrated the viability of diverse, FPGA-based HPC systems for various applications, highlighting the ease of application development and porting akin to conventional GPU-based HPC systems, thanks to new tools and runtimes. This deliverable summarizes vital lessons learnt on efficient FPGA system programming, informed by the experiences of application developers and hardware providers. We provide a set of practical guidelines for programming future heterogeneous HPC systems, derived from a comprehensive analysis of successes and challenges faced during the development process. These guidelines are intended to aid those developing applications for next-generation HPC systems.

Table of Contents

Executive Summary 2
Terminology/Glossary 4
1. Introduction 5
2. Experiences of Applications 5
2.1. Cyberbotics 5
2.2 Underground Analysis 6
2.3 MESHFREE 8
2.4 Lattice Boltzmann 9
3. Experiences of System and Tools Providers 10
3.1 ExaNeSt and Alveo Prototype 10
3.2 JUMAX 10
3.3 Codee 11
4. Guidelines for using the OPTIMA toolflow 13
4.1 Supported platforms 13
4.2 Online repository import 13
4.3 How to use the OOPS library kernels 14
4.4 Buffer allocation 14
4.5 Tips on application debugging 15
4.6 Enabling multiple CUs 16
5. Guidelines for using DFE 16
5.1 DataFlow Engines (DFEs) 16
5.2 DFE Architecture 17
5.3 Programming a DFE application 18
5.4 Getting started on Jumax 18
6. Guidelines for Coding Patterns suitable for DFE Offload 18
6.1 Detailed description 21
6.1.1 PWR055 coding pattern in MaxJ 21
6.1.2 PWR056 coding pattern in MaxJ 22
6.1.3 PWR057 coding pattern in MaxJ 23
6.2 Suggestions for new coding patterns 24
Proposal: Loop Unrolling 24
6.3 Summary 25
7. Conclusions 26

Terminology/Glossary

CUDA	Compute Unified Device Architecture
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
OOPS	OPTIMA OPen Source
OpenCL	Open Computing Language
PCG	Preconditioned Conjugate Gradient

1. Introduction

The main goal of the OPTIMA project is to prove that there are several HPC applications that can take advantage of future highly-heterogeneous, FPGA-populated HPC systems. In addition, by using newly introduced tools and runtimes, application porting/development can be almost as simple as developing software for conventional HPC systems incorporating GPUs. The final versions of all the applications implemented on the OPTIMA hardware were tested and evaluated, while final fine-tuning of the various parameters of the applications was performed. The results obtained when executing these applications on the OPTIMA prototypes were compared with those obtained when the same applications are executed on commercial CPU and GPU-based HPC platforms.

Throughout the duration of the project, several invaluable lessons were learnt on how to efficiently program such FPGA-accelerated systems. In this deliverable we highlight the experiences of the entire process from the application developers and OPTIMA hardware providers. We conclude with preparing and publishing a set of guidelines on how to efficiently program FPGA-based HPC systems. These guidelines will be useful for anyone developing applications for the future heterogeneous HPC systems. We first highlight the experiences from the application developers. We focus on both what worked and what didn't work. Using these experiences, we develop the following sections which focus on the actual guidelines and lessons learnt.

2. Experiences of Applications

2.1. Cyberbotics

Cyberbotics developed the open-source Webots robot simulation software which usually runs on GPU-enabled workstations, clusters and cloud computing infrastructures. In the framework of the OPTIMA project, Cyberbotics wanted to explore the possibility to run a Webots simulation using deep learning in its robot controllers on FPGA-based systems. The goal is to measure the performance on a FPGA-based system compared to CPU-only or CPU+GPU architectures.

The first contribution consists in creating a framework which allows to create simple neural networks: multilayers perceptrons. The details of its implementation in C++ is given here: Creation of a Deep Learning Framework in C++. Implementations are explained here: MLP Forward Propagation on CPU: Tests and Results and MLP Forward Propagation on DFE: Structure.

The second contribution is about autonomous car simulation. The different autonomous car robot controllers are the following:

- *training_track_driving*: this controller is used to follow the train track using the trajectory planning data.
- *CNN_autonomous_car_cpu_float*: this controller contains the naive version of the neural network with floating point representation to drive the car.
- *CNN_autonomous_car_cpu_fixed*: this controller is the optimized CPU version of the neural network with fixed point representation and multithreading to drive the car.

- *CNN_autonomous_car_fpga*: this controller uses the cnn_dfe library to run the neural network on a DFE. This controller has 3 different loop optimization modes.
- *CNN_autonomous_car_fpga_optimized*: this controller uses the *cnn_dfe* library to run the neural network on a DFE. This controller implements only one step loop of *CNN_autonomous_car_fpga*. It is the final, most optimized FPGA version of the controller, described in the Deliverable 2 results.

The DFE optimization of the neural network is compiled in a shared library: *CNN-autonomous-car/libraries/cnn_dfe*. The corresponding DFE kernels and manager are in: *CNN-autonomous-car/libraries/cnn_dfe/src*.

You can select the controller to drive the car by editing the controller field in CNN-autonomous-car/worlds/autonomous_car_test.wbt and then run Webots on Jumax.

A GPU is now available on *jumaxbuild1*. It is recommended to run Webots on *jumaxbuild1* and the controller as extern on *jumax-cpu*. Detailed instructions can be found on the following page: How to run the most optimized car simulation.

2.2 Underground Analysis

M3E developed two hybrid CPU+FPGA applications within the OPTIMA project focusing on the Preconditioned Conjugate Gradient (PCG) method: the first is a distributed-memory implementation of the PCG for large size models while the second is a PCG verticalization for small/medium size models; refer to deliverable D5.2 for further details. In both of these applications, the hybrid system resulted from an interface between M3E's application, originally developed to work entirely on CPUs, and the OOPS library developed by ICCS in the OPTIMA project. In particular, the hybrid PCG implementation requires the utilization of 5 BLAS-L1 kernels and the SpMV kernel from the OOPS library.

In the first application, the OOPS kernels interface directly with the host. From a programming point of view, the interfacing procedure to the high-level functions provided by OOPS is entirely comparable to the use of GPU accelerators via the CUDA language. Specifically, the procedure consists of five steps:

1. Buffer allocation on the device
2. Copy information from the CPU to the device
3. Call the calculation kernel on the device
4. Copy information from the device to the CPU
5. Buffer deletion on the device

Regarding the third step, this was greatly facilitated because in the first part of the OPTIMA project, M3E collaborated with the ICCS on the definition of APIs for several functions of the OOPS library with the dual intent of generalizing and conforming the OOPS library to other existing linear algebra libraries. All these steps are very easily handled at a high level through the OpenCL library.

In the second application, the OOPS kernels interface was built in the hardware and a new top-level function was created in order to utilize it within the PCG kernel. The most

important design characteristics that should be taken into account when designing such an algorithm that involves the incorporation of multiple kernels, are the following:

- The top-level function should utilize all the available bandwidth offered by the device. In the OPTIMA FPGA-based system, it offers 32 banks with 512 bits each. The busses should be 512 bits from memory to all the kernels incorporated within the PCG kernel, in order to efficiently utilize the available memory bandwidth.
- The amount of ports utilized by the top-level kernel should be minimized in order for multiple kernels to be able to be fitted within the FPGA. In the case of the PCG, 4 ports to the memory are utilized per kernel, which allows 8 Compute Units (CUs) to be placed within each ALVEO U55C FPGA. Unfortunately, due to routing issues, only 7 cores were able to be routed in the FPGA by the tools.
- When combining multiple kernels, the interfaces of the kernels and the dependencies between the calls have to be considered. The HLS tool may take into account dependencies that do not exist, while ignoring dependencies that have actual effect on the final results. These have to be considered and the tool has to be guided using the DEPENDENCE directive in order to resolve any issues, which may affect performance and correct functionality.
- Some kernels may require modifications within the PCG kernel in order to optimize their performance. More specifically, for the *ddot* kernel, two different top level functions had to be implemented, due to the fact that the algorithm called *ddot* with the same input in both its arguments. This leads the tool to an incorrect implementation in those cases. We created a new top level function with a single input which is duplicated within the kernel for the calculations.
- Finally, when moving from simple and small kernels, to such a complicated kernel that requires significantly more resources, the tools may have difficulty in routing such design. Special directives had to be provided to the Vivado implementation tool in order for it to route 7 CUs within the FPGA. The default directives allowed the implementation of up to 5 CUs per FPGA. The vivado directives utilized for the implementation are presented below.

[vivado]

prop=run.impl_1.STEPS.OPT_DESIGN.IS_ENABLED=true

prop=run.impl_1.STEPS.OPT_DESIGN.ARGS.DIRECTIVE=Explore

prop=run.impl_1.STEPS.PLACE_DESIGN.ARGS.DIRECTIVE=AltSpreadLogic_high

prop=run.impl_1.STEPS.PLACE_DESIGN.ARGS.DIRECTIVE=Explore

prop=run.impl_1.STEPS.PHYS_OPT_DESIGN.IS_ENABLED=true

prop=run.impl_1.STEPS.PHYS_OPT_DESIGN.ARGS.DIRECTIVE=Explore

prop=run.impl_1.STEPS.ROUTE_DESIGN.ARGS.DIRECTIVE=AggressiveExplore

prop=run.impl_1.STEPS.POST_ROUTE_PHYS_OPT_DESIGN.IS_ENABLED=true

prop=run.impl_1.STEPS.POST_ROUTE_PHYS_OPT_DESIGN.ARGS.DIRECTIVE=AggressiveExplore

2.3 MESHFREE

During the OPTIMA project two different algorithms from MESHFREE were ported to a FPGA version which resulted in an CPU+FPGA hybrid version of MESHFREE for the final iteration of the project. The focus of those two routines were part of the so called “pointcloud” organization inside MESHFREE, which seemed like a good candidate in the beginning. Knowing that FPGAs excel at highly parallelizable tasks, such as doing a single calculation on every point of the pointcloud, which often consists of 1 million points or more, could be a prime example for FPGA acceleration.

During the development of the code we came across multiple issues that we had not predicted nor anticipated. An interface from C to Fortran or vice versa is generally easy to do but gets very complicated once your code does not only consist of simple integers or doubles. For example, dealing with structs in C vs types in Fortran is difficult since every compiler has different alignments in Fortran and the treatment of strings or any other complex matrix or array structure differs in both languages. Another issue was having to port our entire code base from *ifort*, the Intel Fortran compiler, to *gfortran*, the GNU Fortran compiler, since JUMAX did not provide *ifort* and Fortran is not standardized in such a way that you can simply use any compiler. Huge differences exist between those two compilers, including Fortran intrinsics, behavior of allocation and, in general, the handling of modules and headers. So the first guideline we figured out during the lifespan of OPTIMA was to spend enough time before the project to learn about what you want to do. In the past neither of both FPGA platform providers in the project had worked with Fortran code and consequently on both sides there were no past experiences. So if a code consists of either legacy code or worse, a non-standardized language, such as Fortran, those parts can become huge issues for a novice programmer.

Furthermore, just because a part of the code has no data dependency and is highly parallelizable that does not necessarily mean that it is good for FPGA porting. Our part of the code for example consisted of 4 independent loops, but the result of the last loop was required for the next iteration of the first loop therefore not allowing for 5 independent loops. Our expectation here was that we could properly save our whole pointcloud on the FPGA and would not need to transfer it again during the next cycle. This does not suit the idea of a FPGA though, since there is only some memory for constant variables and nothing to store a large data set for the next iteration. Furthermore our code depends on a few Fortran intrinsics in those loops of which some are available in the C *math.h* library, but are not available on the FPGA - some of those are sine, cosine and random. Consequently, going back to CPU was mandatory, which created a much bigger I/O bottleneck than initially expected by us. The guideline resulting from this part was again, to keep raw modern code and do not depend on external libraries in the code part that will be offloaded to an FPGA.

2.4 Lattice Boltzmann

Enginsoft ported a complete new implementation of the lattice D2Q9 in the Palabos Library for the second phase of the project. The process used the MAXELER suite available on JUMAX.

These were key aspects to consider in order to achieve good results based on the experience on the first phase:

- If the application is using MPI, the ported part of the code must be well inside MPI directives, in order to avoid bottlenecks before undergoing the FPGA porting;
- A complete timing of the application should be obtained in order to check the possibilities and maximum gain before starting;
- Then, once analyzed all the above, the porting should be limited to the relevant part of the code where the much time is spent, in order to check possible bottlenecks due the necessary buffering between the memory and the FPGA (see below);
- From a programming point of view the usage of the tools were very similar to those in use with GPU computing with CUDA libraries and the procedure consists of:
    - Allocating buffering vectors;
    - Copying memory from the data structures to buffers;
    - Call the kernel on the FPGA;
    - Copy back data from the buffers;
    - Deallocation (if needed) depending on the structure of data blocks.
- Some of the advantages in using the MAXELLER environment were:
    - The language for generating the computational part on FPGA is very similar to JAVA and generation of the DFE is straightforward;
    - The load distribution of single MPI tasks over different FPGAs is straightforward and was very effective;
- The difficult parts are:
    - Getting started with the interfaces between the languages (in our case the original coding language was C++ and the MAXELLER environment uses a JAVA like sintax).
    - Recode the computational intensive part to this new language.
    - Recode the original program in order to do the buffering, which, in our case, was difficult due to the high Object Hierarchy. In particular we had to overload the original call and recode the specific part entirely going deeply inside the object oriented structure locally.
    - The debugging phase could be critical if something goes wrong (in our case there were few locks of the FPGA system) that needed a reset on a multi user system. Errors from the FPGAs are not easy to understand for a scientific code programmer, not specialized in FPGAs.
    - Timing was not very reliable on JUMAX and we had to run many times (up to 20) to get a reasonable average runtime, even if the system was completely free.

3. Experiences of System and Tools Providers

3.1 ExaNeSt and Alveo Prototype

The ExaNeSt prototype was updated in order for the newest Xilinx tools to be used for creating and using accelerators within the FPGAs. The hardware and firmware were updated and the XRT (Xilinx Runtime) was installed on the platform. The difficulty of programming the specific platform comes with the physical design, as 4 FPGAs are incorporated on each board with each requiring a different hardware design. Although the implementation process was simplified with the use of the Vitis tools, 3 different platforms are used, one for each FPGA as we did not manage to have a working platform project for the network FPGA. The network FPGA's hardware and firmware could not be updated due to the complexity of the hardware design along with the custom software responsible for the softcore network. As a result, for an application to be executed on the Exanest platform, three separate projects have to be used, with each requiring slightly different implementation strategies. This implementation complexity along with the poor performance (Deliverable 6.1) of the ExaNeSt platform lead to the implementation of the new ALVEO prototype platform.

The new ALVEO prototype platform incorporates two servers, each equipped with two ALVEO U55C accelerator cards. The latest Xilinx tools are utilized for their programming with Vitis being used for implementation and XRT for the execution on the devices. The XRT with the OpenCL API simplifies the software - hardware communication. MPI can be used for communication between the servers for the utilization of all 4 accelerator cards. Below there are few notes on using the ALVEO U55C cards:

- The specific accelerator cards come with HBM memory which offers bandwidth of up to 460 GB/s. This bandwidth is provided to applications from 32 banks with 512 bit width ports. The applications targeting this device should try and utilize as much of the available bandwidth as possible.
- When applications use all 32 banks along with significant resources of the FPGA, routing tools may face difficulties on mapping the complete design, and meeting timing constraints. As such, compromises may have to be made with respect to the amount of resources in order for the tool to be able to route the design.
- Developers can also explore the possibility of merging ports, i.e. reusing the same port to access more than one input parameter, and / or try to use the same port for reading parameters and writing results. This strategy can enable the addition of more compute units within the FPGA, which is especially useful for applications that are not I/O bound.

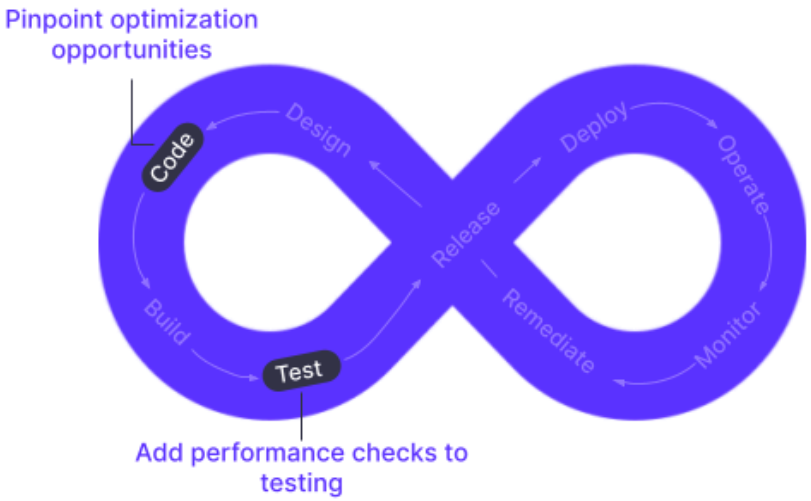
3.2 JUMAX

After initial experiments with JUMAX by Cyberbotics, it became obvious that even though FPGAs could accelerate the inference of Convolutional Neural Networks (CNN), the overall

simulation speeds were still slow due to low rendering speeds from CPU. For this reason, we added a GPU in the JUMAX system to significantly speed up rendering of images required for the simulation. This GPU addition led to more than 5x speedup in overall simulation, which demonstrates the importance of heterogeneous systems especially with FPGAs as accelerators. .

3.3 Codee

Codee is a software development platform that provides tools to help improve the performance of C/C++/Fortran applications targeting multicore CPUs and GPUs. Codee Static Code Analyzer provides a systematic predictable approach to enforce C/C++/Fortran performance optimization best practices for the target environment: hardware, compiler and operating system. It also provides innovative Coding Assistant capabilities to enable semi-automatic source code rewriting through the software development lifecycle. Codee provides integrations with IDEs and CI/CD frameworks to make it possible to 'Shift Left Performance', that is addressing performance issues early in the software development process including the coding and testing stages.



As its first contribution to OPTIMA, Appentra developed an extension of Codee with experimental FPGA support that helps programmers enforce the OPTIMA guidelines. A tangible outcome of the OPTIMA project is the delivery of a free-of-charge Codee software package specialized in FPGA programming. The package includes a command-line interface tool, which lists the coding rules of the 'OPTIMA catalog' that apply to a given C/C++ code. These coding rules of OPTIMA are focused on optimizing code for FPGAs, considering the following programming environments: Maxeler and Xilinx. The tool demonstrates that the benefits of the 'Shift Left Performance' approach is also beneficial in the scope of FPGA programming. It also establishes the baseline for further development of C/C++/Fortran source code checkers that automate the coding rules of the OPTIMA guidelines for FPGA-based HPC systems.

As its second contribution to OPTIMA, Appentra leveraged the Open Catalog of Best Practices for Performance (www.codee.com/catalog/) for FPGA programming, proposing

new FPGA coding rules extracted from the OPTIMA guidelines in cooperation with Maxeler and ICCS, targeting the Maxeler and Vitis programming environments, respectively. The organization and documentation of the catalog mimics existing coding standards for security and compliance with regulations (e.g. SEI CERT C[1], CWE[2]) supported by static code analysis tools from the major vendors. A tangible outcome of the OPTIMA project is the publication of the 'OPTIMA catalog' at optima-hpc.eu/catalog/ with the following rules:

OPTIMA coding rule		Cooperation between OPTIMA partners	Codee coding rule
Identifier	Title		Identifier
OPTIMA001	Consider applying offloading parallelism to forall loop using Maxeler	Maxeler-Codee	PWR055
OPTIMA002	Consider applying offloading parallelism to scalar reduction loop using Maxeler	Maxeler-Codee	PWR056
OPTIMA003	Consider applying offloading parallelism to sparse reduction loop using Maxeler	Maxeler-Codee	PWR057
OPTIMA004	Consider applying loop unrolling to scalar reduction with non-associative operator using Maxeler	Maxeler-Codee	n/a
OPTIMA005	Consider applying offloading parallelism to forall loop using Xilinx	ICCS-Codee	PWR055
OPTIMA006	Consider applying offloading parallelism to scalar reduction loop using Xilinx	ICCS-Codee	PWR056

The 'OPTIMA catalog' demonstrates that the benefits of the Open Catalog of Best Practices for Performance are also beneficial in the scope of FPGA programming, beyond multicore CPU and GPU programming. It constitutes a valuable resource to leverage the knowledge gained in the OPTIMA project to the community of FPGA developers targeting FPGA-based HPC systems.

As its third contribution to OPTIMA, Appentra has gained significant learnings about the development of C/C++/Fortran applications for FPGA-based HPC systems used in OPTIMA:

1. Best practices for performance targeting multicore CPUs and GPUs are also applicable to FPGAs.

[1] https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-c-coding-standard-2016-v01.pdf
[2] https://cwe.mitre.org/data/definitions/699.html

2. Best practices for performance targeting FPGAs can be described as coding rules that leverage the knowledge of FPGA experts to the wider community of FPGA developers.
3. The coding rules used by FPGA experts can be automated through the Codee tools, enabling significant development time savings in the software development process.
4. The Codee tools can interoperate with FPGA software development tools, enabling the identification of opportunities in the C/C++/Fortran code to be offloaded to FPGAs.
5. The automation in Codee of static code analysis features for FPGAs can leverage the detection of C/C++/Fortran features for CPU and GPUs, as demonstrated by the existing rules directly applicable to the OPTIMA environments.
6. The automation in Codee of coding assistant features for FPGAs is feasible and requires new source code rewriting capabilities tailored for FPGA APIs that might be developed driven by a business use case, establishing the foundation of potential future cooperations with Maxeler and Xilinx, now part of AMD.

Overall, the objectives of Appentra in the scope of the OPTIMA project have been successfully accomplished and establish the basis for potential future cooperation between the OPTIMA partners in the future.

4. Guidelines for using the OPTIMA toolflow

These guidelines aim to assist software developers to quickly start using the OOPS library, and easily run their applications to the OPTIMA HPC platform.

4.1 Supported platforms

The OOPs library supports HPC systems that host Alveo u55c FPGA cards. Moreover, its current version is built with the Vitis 2022.1 IDE version.

4.2 Online repository import

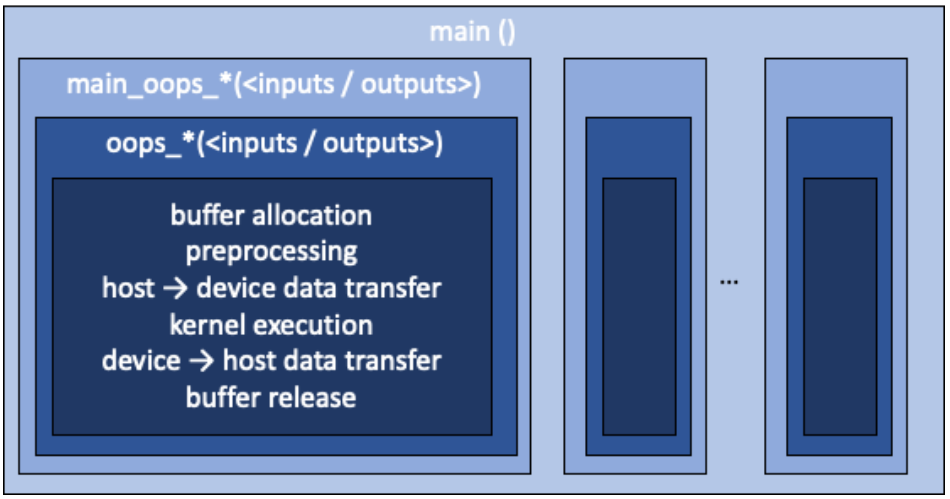
The Vitis IDE GUI allows quickly importing past projects to their workspace. This option does not require developers to go through the process of selecting hardware parts of development kits, since this information is already included. As such, to create a new workspace with the OOPS library, developers can do the following steps:

1. Go to the OOPS online repository and download the latest zip-based project that is available.
2. Start the Vitis IDE GUI and then go to File ⇒ Import.
3. Select the zip-based project.Although the OOPS library is developed using the 2022.1 IDE version, newer releases should be able to import it as well.

4.3 How to use the OOPS library kernels

Software-hardware codesign is an aspect that requires special knowledge from both domains. Application developers, though, need a quick way to port legacy code to new HPC platforms and easily deploy tasks to available hardware accelerators. As such, the OOPS library has been designed in a way to require as few code changes as possible.

For example, all BLAS kernels retain the exact interface than the original BLAS functions. The only difference is the hardware kernel function name, which requires preceding the "oops_" text before the actual name. For example, to offload the BLAS L1 "*dot*" function onto the FPGA, the developers simply need to replace the "*dot*" name with the "*oops_dot*" name. As mentioned in Deliverable D5.2, the Underground Analysis application resembles a solid example of easily porting software for FPGA-supported execution on the OPTIMA HPC platform, by simply renaming its BLAS L1 functions as described above.



Moreover, the Vitis project of the OOPS library allows a quick transition for legacy-code based applications. As shown in the above figure, each hardware kernel is wrapped by a software function that hides FPGA buffer allocation, data transfers, and OpenCL-based runtime interactions, and exposes only the function data input / output (and configuration if applicable) parameters. In addition, each kernel is accompanied by a full "main" program that demonstrates an end-to-end usage (i.e. memory allocation on the host, data transfers, results acquisition), which can be used as a quick "jump-start" template for software developers. As such, developers can call a kernel's main function to run the example code either on software/hardware emulation or directly on the FPGA (after of course running the full synthesis-translate-place-route-bitstream flow).

4.4 Buffer allocation

The FPGA DMA engines require that all buffers used by an application are aligned to 4 KiB pages boundaries. In case a buffer is not so aligned, then the XRT will perform an extra copy to ensure its alignment, increasing the memory overhead. For this reason, the OOPS library provides the void* OOPS_malloc(size_t alloc_bytes) function; OOPS_malloc accepts as input the buffer size in bytes, and returns a void pointer that is aligned at a 4KiB

debug flow	build time	logic validation	pragmas validation	execution time
software emulation	low	software-level	no	fast
hardware emulation	medium	hardware-level	yes	low
hardware implementation	high	real hardware	yes	real time

boundary. As such, developers are strongly advised to use this function to avoid any extra data copies from the XRT during the application execution. The standard free(void* ptr) function can be used to free allocated buffers.

4.5 Tips on application debugging

The Vitis IDE suite allows developers to test applications using either software/hardware emulation or the actual FPGA. The table above summarizes the benefits and limitations of each debug flow. Software emulation enables testing the application functionality, validating software/hardware algorithms, and checking FIFO-related bottlenecks (e.g. full or leftover data after kernel completion). However, it does not take into account any hardware-related aspects, such as latency. On the other hand, hardware emulation is based on the Qemu platform[3] to provide detailed timing of the application when executed on the FPGA, without requiring that the latter is actually installed.

For this reason, hardware emulation requires long running times, hence developers are advised to use small data sets during hardware emulation. An important benefit of hardware emulation versus running the application on the FPGA is that the Vitis flow requires considerably less time to generate the final executables. For example, the hardware emulation flow can take less than 10 minutes to create all executables for a BLAS L1 kernel, whereas hardware execution would require up to 1 hour.

This fact is crucial for hardware debugging that is not available during software emulation. For example, the DATAFLOW directive has no effect during software emulation; on the other hand, the hardware emulation will reveal any hardware-related issues during its flow (e.g. a DATAFLOW directive is not correctly applied, and Vitis HLS cannot generate hardware), and application emulation (e.g. FIFOs get full and / or empty due to the dataflow hardware structure, revealing imbalances between HLS functions). As such, towards reducing compilation time overheads, developers are encouraged to go through software / hardware emulation before running the application to the FPGA.

4.6 Enabling multiple CUs

All OOPS kernels can utilize multiple CUs to improve performance. BLAS L1, L3 and the Jacobi preconditioner kernels can be configured from the software API in terms of the

[3] https://www.qemu.org/

number of CUs to be used. To change the number of CUs in BLAS L2 / L3, LU decomposition kernels, developers need to instantiate or remove the corresponding high-level functions within the HLS kernel code. The reason that these compute-demanding kernels can not be configured in terms of CUs from the software API is because “hardwired” CUs exhibit less latency during invocation, compared to multiple CUs that are enabled using the OpenCL runtime.

Overall, this option allows developers to balance between area utilization and performance, which is very useful for cases where many functions will be offloaded to hardware kernels. For example, if an application is based on many different vector-vector operations, developers can instantiate most (if not all) kernels to the FPGA using a different number of CUs for each function, based on the workload requirements.

To enable multiple CUs from the Vitis IDE GUI for the BLAS L1 and the Jacobi preconditioner kernels, developers can go to the Assistant ⇒ expand the oops_library_system_hw_link ⇒ Hardware ⇒ and double-click on the binary_container_1. From there, they can configure the number of CUs, along with the HBM channels to be used by each one of them. As a final step, developers need to configure the corresponding parameter that designates the number of CUs to be used during the application execution.

It should be noted that a set of updated instructions and examples will be available at the OOPS online repository.

5. Guidelines for using DFE

Cyberbotics has published a series of guidelines to access the JUMAX machine, use the DFE, install the robot simulation on the JUMAX machine, develop multi-layer perceptrons and convolutional neural networks for autonomous car simulations. The complete guidelines are available online from the public wiki pages of the CYB optima project at https://github.com/cyberbotics/optima/wiki. We report here only the page entitled *Basics of DFE Applications*.

5.1 DataFlow Engines (DFEs)

DataFlow Engines (DFEs) are processing systems developed by Maxeler and allow accelerated execution of CPU programs. They are based on FPGAs (Field Programmable Gate Arrays) and allow Dataflow Computing. In contrast to CPU control flow computing, where instructions of the program are more or less sequentially executed, dataflow computing allows parallelization of operations on big data. DFEs are composed of thousands of dataflow cores that can be re-programmed for each application to allow the biggest optimization possible. DFEs cores are very simple and each of them executes a very simple operation, but programming an application in space instead of in control is also really more complex. Fortunately, Maxeler provides a set of tools to make it really easier.

As can be understood, DFEs are essentially FPGAs with additional connections and memory added by Maxeler. In addition to the on-chip memory blocks, DFEs provide a large amount of external memory which can be accessed slower. Also fast connections, named MaxRing, allow communication between engines. On Jumax, DFEs are connected to the

CPU with InfiniBand, a computer networking communications standard used in high-performance computing.

Jumax is equipped with a MPC-X card, which has 8 DFEs of MAX5 generation, each containing a Virtex UltraScale VU9P FPGA from Xilinx. This chip contains 2,586,000 logic cells, 6,840 Digital Signal Processors for multiplications and 340Mb of memory blocks. External slow memory has a capacity of 48Gb. Head to this page for a more detailed configuration of Jumax: JuMax DFEs (MAX5 gen).

5.2 DFE Architecture

In general, FPGAs are programmed using HDLs (Hardware Description Languages). But these languages may not be intuitive to everybody, so Maxeler developed MaxCompiler to ease the process.

Thanks to MaxCompiler, DFEs can be programmed using .maxj files (a JAVA based language). MaxCompiler compiles this dataflow program into a java executable which produces the corresponding HDL files, in function of the vendor of the FPGA in the DFE. The synthesis step produces a set of logic operations from the HDL files. Placing step takes care of choosing the logic blocks to implement the set of logic operations on the chip. Finally, the routing step defines the wires that interconnect the blocks. One of the main tasks for the compiler is to find a route and placement which satisfies the frequency of the clock. If the frequency is too high or the design too complex, the compiler can return a timing failure error. For a more illustrated view of the process you can go to: From Graphs to Hardware and to: Substrate Agnostic Compilation.

MaxCompiler generates a .max file which contains the configuration of the DFEs for the given application. At runtime, the CPU program loads them to the DFEs, using MaxelerOS. MaxelerOS is running within Linux and DFEs and is responsible for making the link between CPU and DFE by loading, executing and unloading compiled .max files.

Finally, CPU programs can call functions running on DFEs defined in .max files. To do that, .max files must be linked to the CPU application using standard GCC linker. The dataflow implementations can then be called using the SLiC (Simple Live CPU) interface.

A synthesized illustration of what is going on is accessible here: MaxCompiler Architecture.

It is also possible to call DFE functions from programs coded in a language other than C. Sliccompile is a SLiC tool bundled with MaxCompiler allowing to execute DFE functions from any supported language by generating SLiC Skins. They consist of all the class, function or script files needed to call the kernel functions from Matlab, Python, and other CPU applications. A *_simutils_* folder is created to make the link to related .max files.

A more detailed overview of the DFE architecture can be found in this presentation: Using, Understanding and Programming Data Flow Engines.

5.3 Programming a DFE application

DFE applications are easily programmed to Maxeler's hardware platform; they are made of kernels (.maxj files), a manager (.maxj file) and a CPU application code (in C for

example).The kernels define functions that will run on DFEs. Programming kernels are very close to java but they use DFE specific variables. In kernels, basic arithmetic functions can be used, as well as loops or conditionals. For an example of a kernel, head to this page: A Basic Kernel.

The Manager configures the kernels, connects them together and links the stream from and to the CPU. For an example of a manager, head to this page: Configuring a Manager. The CPU program can directly call a dataflow implementation in case the corresponding .max files are linked. For an example of a CPU application in C, go to this page: Simple CPU application.

A more detailed overview of the programming process can be found in this MaxCompiler tutorial.

5.4 Getting started on Jumax

To start creating DFE applications, you can start MaxIDE on Jumax using this page: Start MaxIDE.

6. Guidelines for Coding Patterns suitable for DFE Offload

In this section we describe how the Codee tool can be used to identify code patterns for Maxeler DFEs. Maxeler uses a dataflow-oriented programming model for FPGA-based Dataflow Engine (DFE). In a dataflow computing model, operations are naturally concurrent: data flows from memory or the host CPU into a chip where arithmetic operations are carried out by chains of functional units, which are statically interconnected in a topology corresponding to the implemented functionality. Data streams from one functional unit directly to the next one without the need of complex control mechanisms. Data arrives just in time when it is needed, and the final results flow back into memory. This dataflow model of computation maps well to reconfigurable devices such as FPGAs, and building the accelerator architecture a model of minima data movements reduces the potential for memory bottlenecks.

Codee currently offers built-in support for identifying code patterns suitable for GPU offload: https://www.codee.com/catalog/ . As part of the OPTIMA project, MAX analyzed if and how these existing patterns are applicable to Maxeler DFEs. Superficially, GPUs and FPGA-based accelerators, such as Maxeler DFEs, may seem similarly suitable to accelerate and offload compute intensive tasks. There are, however, some important differences between the capabilities of GPUs and DFEs. While it is beyond the scope of this document to offer a detailed comparison of these very different architectures, we can identify some significant key points:

- GPUs are characterized by a highly parallel SIMD architecture, which works well for workloads that can be highly parallelised or batched.
- FPGAs offer a fine-grain reconfigurable logic which works well for mapping more irregular compute problems, e.g. with very different pipeline stages.
- The fine-grain FPGA architecture supports many application specific optimisations, such as custom number formats, encodings and computational styles (serial vs

parallel vs pipelines). This is because custom compute pipelines can be created from the fine-grain logic.

- The dataflow model is ideal for throughput-oriented compute models where data transfers from and to DMA buffers can be fully overlapped with the actual compute on the chip.
- The static dataflow model employed in Maxeler DFEs is fully deterministic, which can be leveraged during the development process (e.g. making the implementation plannable and allowing the developer to optimize an architecture that has not been implemented yet) and during runtime (e.g. building accelerators that act with known and fixed throughput and latency which can be essential for networking applications).

Due to these differences, GPU coding styles may not always be appropriate for DFE accelerators. In the following, we analyse all existing GPU-oriented coding patterns in Codee. For the patterns that are applicable to DFEs, we give examples of MaxJ code implementations that would be suitable to realize this pattern on a Maxeler DFE. We also give a suggestion for an additional pattern where support could be added in the future.

The following list gives an overview of the performance issues and coding suggestions for the GPU offloading that are currently supported in Codee. All of these are based on static code analysis:

- Applying the offloading parallelism on the loop containing the forall pattern (PWR055)
- Applying the offloading parallelism on the loop containing the spare reduction by handling the race condition issue (PWR057)
- Offloading the scalar reduction (PWR056)
- Suggestion for the SIMD vectorisation directives for the offloaded region (RMK008)
- Hybrid parallelisation suggestion for the nested loops (GPU offloading and vectorisation) (RMK002)
- Pragma annotation suggestions for the OpenACC and OpenMP (PWR026, PWR027)
- Distributing the offloaded work by applying the OpenMP teams (PWR009)
- Prevention of copying the unused data to the GPU (PWR015)
- Offloading optimisation of the derived data types (PWR012)
- Optimizing the data transfer to the GPU (PWD005, PWD006, PWD003)
- Preventing the unwanted data movements to GPU (PWR013)

The following table gives an overview of how these coding patterns are applicable to DFE acceleration via MaxJ, and suggests what the MaxJ pattern would cover.

Catalog ID	Description	DFE offload	Note / MAXJ suggestion
PWR055	Applying the offloading parallelism on the loop containing the forall pattern	Directly applicable	It can be done as a single reduction pipeline or multiple pipelines for better performance.
PWR05	Applying the offloading	Partially	It can be implemented using

7	parallelism on the loop containing the spare reduction by handling the race condition issue	applicable	mapped memories. This implementation would have a loop in the design that would impact the performance.
PWR056	Offloading the scalar reduction	Directly applicable	It can be done as a single reduction pipeline or multiple pipelines for better performance
RMK008	SIMD vectorisation directives for the offloaded region	Directly applicable.	With FPGAs we can configure hardware as we want. PWR055 example shows SIMD behavior.
RMK002	Hybrid parallelisation suggestion for the nested loops	Directly applicable.	PWR055 example shows both vectorization and SIMD behavior.
PWR026	Pragma annotation suggestions for the OpenACC	Not applicable.	SLiC is a C/C++ library, not compiler extension.
PWR027	Pragma annotation suggestions for the OpenMP	Not applicable.	SLiC is a C/C++ library, not compiler extension.
PWR009	Distributing the offloaded work by applying the OpenMP teams	Partially applicable.	Similar behavior can be implemented in MaxJ. Multiple threads and multiple kernels. This would have greater overhead than having wider input to a single kernel and multiple SIMD pipelines.
PWR015	Prevention of copying the unused data to the GPU	Directly applicable.	Streaming unused data will impact the performance.
PWR012	Offloading optimisation of the derived data types	Directly applicable.	It helps code analysis. Same as PWR015.
PWD005	Optimizing the data transfer to the GPU	Directly applicable.	Not streaming enough data will cause MaxJ kernels to stall.
PWD006	Optimizing the data transfer to the GPU	Directly applicable.	Same explanation as for GPU.
PWD003	Optimizing the data transfer to the GPU	Directly applicable.	Same explanation as for GPU.
PWR013	Preventing the unwanted data movements to GPU	Directly applicable.	Streaming unused data will impact the performance. Setting unused mapped memories or mapped registers will impact the performance.

6.1 Detailed description

6.1.1 PWR055 coding pattern in MaxJ

1. Issue
   The loop containing the forall pattern can be made faster by offloading it to a DFE accelerator using the Maxeler programming environment.

2. Action
   Implement a version of the forall loop using an Application Program Interface (API) provided by the Maxj compiler of the Maxeler programming environment.

3. Relevance
   A loop often represents a computationally expensive block of computation and a FPGA-based DFE accelerator can be used to speed up the computation. A loop without loop-carried dependencies is straightforward to map to a simple MaxJ implementation (see code example V1 below), and it can be further optimised by parallelisation (see code example V2). The programmer does not need to explicitly manage data transfers between CPU and DFE accelerator but the IO overhead of data transfers needs to be considered in the overall accelerator development.

4. Code examples
   CPU:

```
void example(double *D, double *X, double *Y, int n, double a)
{
      for (int i = 0; i < n; ++i) { D[i] = a * X[i] + Y[i]; }
}
```

MAXJ:
V1(simple solution)

```
DFEVar X = io.input("X " , dfeFloat(11, 52));
DFEVar Y = io.input("Y" , dfeFloat(11, 52));
DFEVar a = io.scalarInput("a " , dfeFloat(11, 52));
io.output("D", dfeFloat(11, 52)) <== a * X * Y;
```

V2(optimized solution)

```
// N is a variable that can be tuned depending on available FPGA
resources
DFEVectorType<DFEVar> vectorType = new DFEVectorType<DFEVar>
                                                (dfeFloat(11, 52), N);
DFEVar X = io.input("X ", vectorType);
```

```
DFEVar Y = io.input("Y", vectorType);
DFEVar a = io.scalarInput("a " , dfeFloat(11, 52));
DFEVector<DFEVar> output = io.output("D", vectorType);
for(int i = 0; i < N; i++){
      output[i] <== a * X[i] * Y[i]
}
```

6.1.2 PWR056 coding pattern in MaxJ

1. Issue
   The loop containing the scalar reduction pattern can be made faster by offloading it to an accelerator.

2. Action
   The code example provides a MaxJ solution that could be applied to a scalar reduction pattern.

3. Relevance
   Offloading a loop to an DFE is one of the ways to speed it up. DFEs offer significant computation resources to accelerate the computation but in order to be efficient, the MaxJ implementation needs to consider the accumulation step of the computation. Controlling the degree of pipelining can be used to control latency, throughput and hardware resources needed by the hardware implementation.

4. Code example

   CPU:

```
void example(int *A, int n) {
  double sum = 0;
  for (int i = 0; i < n; ++i) {
    sum += A[i];
  }
  return sum;
}
```

   MaxJ:

```
// N is a variable that can be tuned depending on available FPGA
resources
DFEVectorType<DFEVar> vectorType = new DFEVectorType<DFEVar>
                                                    (dfeUInt(32), N);
DFEVar enableOutput = control.count.simpleCounter(32) === n;
DFEVEctor<DFEVar> A = io.input("A", vectorType );
DFEVector<DFEVar> output = io.output("sum", vectorType, enableOutput);

for(int i = 0; i < N; i++) {
```

```
DFEVar carriedSum = dfeUInt(32).newInstance(this); // sourceless
stream
      DFEVar sum = A[i] === 0 ? 0.0 : carriedSum;
      optimization.pushNoPipelining();
      DFEVar newSum = A[i] + sum;
      optimization.popNoPipelining();

      DFEVar newSumOffset = stream.offset(newSum, -1);

      carriedSum <== newSumOffset;
      output[i] <== sum;
}
```

6.1.3 PWR057 coding pattern in MaxJ

Applying the offloading parallelism on the loop containing the spare reduction by handling the race condition issue

1. Issue
   The loop containing the sparse reduction pattern can be made faster by offloading it to an accelerator, while properly handling the race condition issue to preserve correctness.

2. Action
   The code example provides a MaxJ solution that could be applied to a sparse reduction pattern.

3. Relevance
   Offloading a loop to an DFE is one of the ways to speed it up. A sparse reduction pattern can be implemented with a mapped memory on the DFE which offers an efficient way to handle the data dependent access patterns in the computation.

4. Code example

   CPU:

```
void example(int *A, int *nodes1, int n) {
  for (int nel = 0; nel < n; ++nel) {
    A[nodes1[nel]] += nel * 1;
  }
}
```

   MaxJ:

```
Memory<DFEVar> mappedMemoryA = mem.alloc(dfeUInt(32), 512);
mappedMemoryA.mapToCPU("A");
```

```
DFEVar enableInput =
control.count.makeCounter(control.count.makeParams(4).withMax(2)).getC
ount() === 0;
DFEVar nodes1 = io.input("nodes1", dfeUInt(32), enableInput)
nodes1 = nodes1.cast(dfeUInt(MathUtils.bitsToAddress(512)));

DFEVar nel = control.count.simpleCounter(32);

DFEVar A = mappedMemoryA.read(nodes1);
optimization.pushNoPipelining();
A = A + nel;
optimization.popNoPipelining();
mappedMemoryA.write( stream.offset(nodes1, -2), stream.offset(A, -2),
constant.var(true));
```

6.2 Suggestions for new coding patterns

As shown above, the GPU oriented coding patterns are applicable to DFEs with a varying degree. In the following we also propose a new coding pattern that maps very well to DFEs but is currently not available in Codee. This could be made available in a future version that is more optimized for DFE coding styles.

Proposal: Loop Unrolling

Loop unrolling is a common and important coding construct for DFE. It deals with loops that have a carried dependency between loop iterations. Due to the dependency, the loop cannot be parallelised into independent blocks of computation, but it can be unrolled into a pipeline that offers a high degree of concurrency and throughput.

Code examples
CPU:

```
void example(float *input, float* output, int numItems) {
  for (count=0; count<numItems; count += 1) {
    float d = input[count];
    float v = 2.9142 - 2*d;
    for (iteration=0; iteration < 4; iteration += 1) {
      v = v * (2.0 - d * v);
    }
    output[count] = v;
  }
}
```

MAXJ:

```
DFEVar d = io.input("input" , dfeFloat (8, 24));
```

```
DFEVar v = 2.9142 - 2.0 * d;
for (int iteration = 0; iteration < 4; iteration += 1) {
v = v * (2.0 - d * v);
}
io.output("output", v, dfeFloat (8, 24));
```

6.3 Summary

We have analyzed how Codee GPU coding patterns are applicable for Maxeler DFEs. While there are some similarities, we also note that there are differences due to the different compute architectures of these two accelerators. We gave some examples for MaxJ implementations that correspond to some of the existing code patterns and give recommendations for future extensions.

7. Conclusions

The OPTIMA project successfully showcased the potential of upcoming, FPGA-enhanced, diverse HPC systems across a range of applications. This has resulted in significant insights into the efficient programming of FPGA systems. This report distills earlier project insights into actionable guidelines for future technological progress. It captures the joint experiences and expertise of the application developers and hardware experts participating in OPTIMA, with the ultimate goal of developing and sharing detailed programming guidelines for FPGA-based HPC systems.