FACULTY OF ENGINEERING AND ARCHITECTURE

Accelerating numerical libraries for Python using FPGAs

Gerbrand De Laender
Student number: 01406041

Supervisors: Prof. dr. ir. Erik D'Hollander, Prof. dr. ir. Dirk Stroobandt

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2020-2021

GHENT UNIVERSITY

FACULTY OF ENGINEERING AND ARCHITECTURE

Accelerating numerical libraries for Python using FPGAs

Gerbrand De Laender
Student number: 01406041

Supervisors: Prof. dr. ir. Erik D'Hollander, Prof. dr. ir. Dirk Stroobandt

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2020-2021

GHENT UNIVERSITY

Preface

After an intensive period of about 10 months, I now put the finishing touches to this master's thesis. For me, this was a unique opportunity to get a grasp of the wonderful – but complex world – of embedded software and hardware design.

A lot of work has been put into this thesis and I can finally and truthfully thank the persons that helped me throughout the process. First of all I would like to thank my supervisors, Prof. dr. ir. Erik D'Hollander, Prof. dr. ir. Dirk Stroobandt for their trust, support and feedback. In particular, I would like to express my gratitude to Prof. dr. ir. Erik D'Hollander, for his intensive guidance during this thesis, from which I for sure learned a lot. Furthermore, I would like to thank you for your rapid and crystal-clear communication whenever I was stuck with a problem.

Furthermore, I would also like to thank my parents, for giving me the opportunity to complete this study programme and for the support over the past years. Last but not least I would like to thank my girlfriend Sarah for the great support during this period and for being particularly patient while I was working on this thesis "again".

A heartfelt thank you to everyone!

Gerbrand De Laender, august 2021

Accelerating numerical libraries for Python using FPGAs

Gerbrand De Laender, student number: 01406041

Supervisors: Prof. dr. ir. Erik D'Hollander, Prof. dr. ir. Dirk Stroobandt

Master's dissertation submitted in order to obtain the academic degree of Master of Science in Computer Science Engineering

Academic year 2020-2021

Abstract

While there exist several projects in which Python's most popular numerical library, NumPy, is accelerated using GPUs, multi-core CPUs, and clusters, there seems to be very little research that attempts to achieve this goal using FPGAs. In this work, a methodology is devised that allows numerical applications in Python to be accelerated using FPGAs. In order to validate this methodology, a custom Python module that can be used as a drop-in replacement for NumPy, is presented, in which routines are automatically and transparently forwarded to their hardware-accelerated counterpart on the FPGA, given that a compatible and profitable accelerator exists. To support this system, various NumPy routines are transformed into custom hardware accelerators by means of high-level synthesis and integrated into Python using parts of Xilinx's PYNQ platform. A speed-up by an order of magnitude is found to be possible in a practical application where the two-dimensional heat equation is solved numerically.

Keywords: *Python*, *NumPy*, *FPGA*, *Zynq*, *PYNQ*

Accelerating numerical libraries for Python using FPGAs

Gerbrand De Laender, Ghent University (gerbrand.delaender@ugent.be)

Supervisors: Prof. dr. ir. Erik D'Hollander, Prof. dr. ir. Dirk Stroobandt

***Abstract*— While there exist several projects in which Python's most popular numerical library, NumPy, is accelerated using GPUs, multi-core CPUs, and clusters, there seems to be very little research that attempts to achieve this goal using FPGAs. In this work, a custom Python module that can be used as a drop-in replacement of NumPy, is presented, in which routines are automatically and transparently forwarded to their hardware-accelerated counterpart on the FPGA, given that a compatible and profitable accelerator exists. To support this system, various NumPy routines are transformed into custom hardware accelerators by means of high-level synthesis and integrated into Python using parts of Xilinx's PYNQ platform. A speed-up by an order of magnitude is found to be possible in a practical application where the two-dimensional heat equation is solved numerically.**

I. INTRODUCTION

The integration of FPGA-based hardware accelerators into applications requires specific skills and knowledge of low-level tools that are out of reach for the largest part of software developers, which hinders a more widespread adoption. The use of high-level synthesis (HLS), in which a high-level algorithmic description is used to create digital hardware instead of a more tedious description at the register-transfer level (RTL), partially mitigates some of these difficulties, but it still essentially requires the same development process [1].

In an attempt to make it easier for designers of embedded systems to exploit the benefits of their devices, Xilinx released PYNQ (Python productivity for Zynq) in 2016. PYNQ tries to achieve this goal by allowing developers to build complex and capable applications very quickly, using the popular Python programming language along with its wide ecosystem of over 300,000 libraries[1]. In PYNQ, programmable logic circuits are presented as hardware libraries called overlays, which are analogous to software libraries. The hardware functionality is made available to the user through a Python API that allows the developer to select the overlay that best matches their application. Creating a new overlay still requires expertise in designing programmable logic circuits, but they are designed to be configurable and re-used as often as possible in many different applications, adhering to the build once, re-use many times paradigm[2].

In recent years, the PYNQ open-source community has produced a reasonable amount of projects that demonstrate the capabilities of the framework, with applications ranging from signal processing, video filtering, compression and encryption to real-time object detection and neural network acceleration[3]. Unfortunately, most important and widely used numerical libraries in Python, such as NumPy and SciPy have remained relatively untouched. When these libraries are used, they are directly executed on the ARM-processor and, hence, do not take advantage of the programmable logic, even though many of the computational kernels accommodate hardware acceleration. In this work, this aspect is further elaborated on and efforts are made to speed up these libraries using FPGA-based accelerators.

II. RELATED WORK

A. Accelerating Python using FPGAs

The idea of acceleraring Python code using FPGAs is not new, and the introduction of the PYNQ framework has led to an increased research activity over the past years. Countless applications that make use of PYNQ can be found in the areas of machine learning, image processing, computer vision, digital signal processing, data analytics, security (e.g. encryption), and many more. Some research take a more generalist approach or expand the framework, such as in [2], where the impact of using PYNQ on Zynq devices, its performance implications, and its bottlenecks are investigated. Other work includes that of [3], in which the functionality of PYNQ is extended to support dynamic partial reconfiguration, which allows to reconfigure parts of the programmable logic while the other parts remain active, allowing for a more efficient and flexible use of the available FPGA resources.

Furthermore there exists several works [4]–[9], that aim to use Python as a high-level hardware description language, in an attempt to reduce the development time, and bridge the gap between low-level and high-level hardware design. Other research tries to convert Python code to a high-level synthesisable language, such as in [10], [11], while in [12], [13] even the synthesis, bitstream generation and host program creation are automated into a single flow, leaving the developer with virtually no further work. Even though such systems significantly improve the developer's productivity, they don't seem to be able to outperform highly optimised CPU implementations without FPGA-specific programming efforts.

[1] https://pypi.org/

[2] https://pynq.readthedocs.io/en/v2.6.1/

[3] http://www.pynq.io/embedded.html

B. Accelerating NumPy using FPGAs

The concept of transparently accelerating NumPy on hardware is also not new, but most research so far has been targeting GPUs, including CuPy and ClPy [14], [15]. A more general approach is taken by the Bohrium project [16], in which a runtime system is created that maps array operations onto a number of different hardware platforms. It realises this by using a front-end component that compiles the user code into a Bohrium bytecode, which in its turn can be executed by architecture-specific implementations such as GPUs [17]. In the original article, the authors mention the future implementation of a Bohrium back-end that enables support for FPGAs as well, however this route seems to be discontinued.

The work that ties in most strongly with this thesis is that of [1], [18], in which the authors build upon the PYNQ framework to enable transparent hardware acceleration for scientific computations on Zynq platforms. They provide implementations for the correlation function, matrix dot product, standard deviation and the FFT, and validate their work using a biomedical use case. Similar to this work, they provide a run-time mechanism in which the accelerators can be transparently invoked by overloading standard NumPy methods. They also make use of a system that decides between software and hardware acceleration depending on the history of invocations. This contrasts with the system that is employed in this work, in which semi-automated benchmarks are used for this purpose. Also, they rely on writing lower-level drivers in CPython directly, as opposed to CFFI or pybind11, and provide no possibility to integrate multiple accelerators into a single hardware design, with an increased reconfiguration overhead as a result.

C. Hardware/software development environments

There also have been several studies, in particular in the area of reconfigurable computing, that consider the unification of hardware and software design on SoC and MPSoC, together with the creation of systems that attempt to relieve software developers of the burden of managing the hardware and associated data transfers at run-time. These include [19], [20], in which abstraction is made of the available hardware accelerator in the form of *hardware threads*, that can interact with software within a multithreaded environment.

In this work, the focus is more on the integration of different streaming-based kernels, which is better aligned with the work of [21], in which Redsharc is introduced. The authors elaborate further on the idea of creating an API to abstract communication between heterogoneous computational units, but besides that, they also present custom on-chip networks to support this API. By adapting a streaming model instead of an MPI- or threading-based system, they allow for a direct connection between the computational units as the streams only contain data. In this work, a lightweight mechanism similar to their stream switch network (SSN), which allows to route streams between different hardware/software kernels, is adopted.

III. METHODOLOGY

In this section, an overview is given of the methodology that is used in order to accelerate numerical Python applications. Note that the methodology is formulated in a generic way and is not particularly bound to specific platforms, architectures, systems or software tools.

A. Identification of the procedures that will be implemented in hardware

An FPGA can accelerate a wide variety of operations, and a choice must be made as to what operations will be performed on the hardware. This choice can be based on the specific application in mind, or it is also possible to take a more generic approach and pick functionalities that are useful in many different scenarios. In this thesis, a combination of both is used: solving the heat equation is put forward as an application, but by generalising the operations that are required for this purpose to ”classes of operations”, it is possible to speed up many more different programs as well. Alternatively, it is also possible to profile an existing numerical Python application and implement the procedures that takes the longest to execute.

B. Algorithmic description of the selected procedures and the introduction of streaming behaviour

To be able to implement the selected procedures in hardware, an algorithmic description of each of them is made first. In order to create an efficient and, most importantly, fast hardware accelerator, the use of input streams and output streams to represent (multidimensional) arrays is strongly encouraged. This means that in the algorithmic description, it must (conceptually) be taken into account that the inputs enter the system one element at a time, and also leave the system one element at a time.

C. Implementation of the selected procedures on hardware using HLS tools

High-level synthesis allows the developer to describe the hardware design at a high level of abstraction, using an algorithmic description in for example C or C++. A high-level synthesis tool such as Vitis HLS transforms (synthesises) this description into a RTL design in a HDL such as Verilog or VHDL. At this stage, it is also possible to verify the correctness of the implementation by writing a C or C++ test bench. In this way, it is possible to validate the functional design, both before synthesis (using a C simulation), and after synthesis (using software/hardware co-simulation).

D. Optimisation of the hardware accelerators

Based on the synthesis results, an estimate can be made of the throughput, latency and resource consumption of the hardware accelerators. Often, additional directives (or pragmas) are required to further optimise the design and achieve a higher throughput, a reduction of the latency of the core, or a reduction of the required area or device resource usage. In this step, the synthesis reports provide feedback

to the developer and can guide the design space exploration towards an optimal or nearly optimal solution.

E. Creation of a hardware block design that integrates multiple accelerators

In order to make use of the created accelerators, they must be incorporated into a hardware design that allows them to communicate with the processing system of the SoC, on which the target application is running. This entails the use of external, prefabricated components or intellectual property (IP) cores, which fulfil a certain functionality within the design. This may involve, for example, moving data from the main memory to the hardware accelerators in a streaming fashion (and vice versa), or linking the different accelerators together. At this stage, it is beneficial to add as many accelerators as possible to the design, to make full use of the available resources. In this work, a generic block design is created using Vivado, and can be used as a template for this purpose. The output of this stage is a bitstream file that can be loaded onto the FPGA.

F. Creation of C/C++ drivers that can interface with the block design

In order for an application to access the accelerators, it is necessary to create C or C++ drivers that are responsible for the communication. While it is possible to directly write drivers for the Linux operating system, it is advised to test the interfacing and the functionality of the hardware design on a platform without an operating system, i.e. running *bare metal*. Software tools such as Vitis are suitable for this purpose and also allow the use of a variety of pre-made and auto-generated drivers. In addition, they also allow the software to be debugged, which is more difficult in a running operating system. In a later stage, it is then possible to translate the functionality into Linux drivers, once the correct way to interface with the hardware design has been found. In this work, a tool has been created that aids the developer in creating Linux drivers, using automatically generated Jupyter Notebooks.

G. Binding of the C/C++ drivers to a Python module

To interface with the accelerators on the FPGA from Python, the C/C++ drivers need to be bound to a module, which can be done in various ways, for example using CFFI or pybind11. As part of this thesis, a Python module was created that keeps track of all available hardware accelerators in a library and allows their C/C++ drivers to be called in a convenient manner. If several hardware designs are used interchangeably, it is also necessary to create a system that allows the correct bitstream to be loaded onto the FPGA before invoking the drivers.

H. Profitability analysis of the hardware accelerators

When the drivers are in place and can be invoked from a Python program, it is also possible to check the performance gain that is achieved by invoking the hardware-accelerated version of a specific procedure instead of the original Python implementation. In many cases, the hardware accelerators provide only a minimal acceleration when the input streams are small, because of the overhead that is associated with the PS/PL interaction. In this work, a benchmark is done for each Python procedure that is to be accelerated, to find the minimals size of the input stream for which the hardware accelerator becomes profitable.

I. Acceleration of numerical applications in Python

Finally, it is possible to accelerate a given numerical application by invoking the corresponding drivers at run-time instead of the original procedures. As part of this thesis, a mechanism was created that automatically and transparently forwards method calls to their respective hardware accelerated counterparts, if the functionality of the method is implemented and it is profitable and possible to do so. At this point, it is also possible to obtain insights into the obtained speed-up.

IV. IMPLEMENTATION

A. Hardware implementation

As part of this work, a number of hardware designs have been created to facilitate the acceleration of commonly used NumPy functions. The block design shown in Figure 1 allows for the integration of multiple accelerators at once, and is used as a template for all implemented hardware designs. Data is streamed from the DRAM to the appropriate accelerator cores using all four available AXI HP ports, which are each connected to a dedicated AXI DMA channel (two incoming and two outgoing channels). The AXI-stream switch routes all incoming streams to the appropriate IP core(s) and forwards them back to the DMAs, in case the accelerator produces (an) output stream(s). The accelerator cores can each take up to two input streams and can produce up to two output streams. Furthermore, all cores are controlled using AXI-lite by making use of the general-purpose master interface of the processing system.

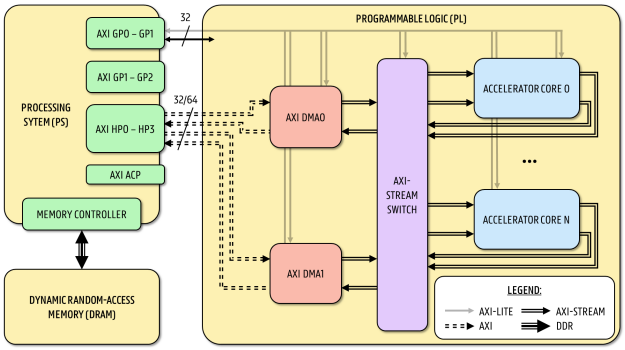

Fig. 1. Proposed generic block design. Data is streamed from the DRAM to the appropriate accelerator cores using all four available AXI HP ports, each connected to a dedicated AXI DMA channel. The AXI-stream switch routes all incoming streams to the appropriate IP core(s) and forwards them back to the DMAs if required. All PL components are controlled using AXI-lite over the GP master interface of the PS.

A first class of operations is implemented by the `ufunc__call__xx` cores, where `xx` designates the data type of the input and output streams, such as `i4` for integer and `f4` for floating-point. The core is able to apply a unary operation (arithmetic, goniometric, boolean...) on each element of its input stream, or a binary operation on both of its input streams, resembling the invocation of the `__call__` method on a NumPy universal function, with the input stream(s) as arguments[4]. This allows for the hardware acceleration of NumPy operations such as `np.sin(x)` and `np.multiply(x, y)`. Instead of creating several similar accelerators that each implement a single operation, the core is equipped with a mechanism that allows to select the operation using its AXI-lite control interface, which saves resources by allowing them to be shared.

A second class of operations is covered by the `ufunc__reduce_all__xx` cores which, unlike the previous class, only produces a single value from its input stream(s). The core is able to apply a unary operation on each element of its input streams, or a binary operation on both of its input streams, followed by a repeated application of a binary operation to the resulting stream, effectively reducing the stream to a single value. Furthermore, it is also possible to apply a final unary operation to this value before stopping the core and writing the return value to the output register. This behaviour resembles the invocation of the `__call__` method on a NumPy universal function, followed by the `reduce` method, with the input stream(s) as arguments. This allows for the hardware acceleration of a plethora of NumPy operations, including `np.product(x)`, `np.sum(x**2)`, `np.sum(np.abs(x - y))` and `np.log(np.sum(np.exp(x)))`. Again, all operations can be set by making use of the AXI-lite control interface, which prevents having to create all possible (or meaningful) combinations.

A third type of accelerator was created to support filter-type of operations, in which a kernel slides over a two-dimensional input array. An implementation was made of an average filter, which is useful for acceleration of the heat equation application that is put forward. It should be noted that several optimisations have been carried out during the development of the cores as to obtain an accelerators that can produce one new result every clock cycle, i.e. achieving an initiation interval (II) of one. For example, for the reduction-type cores, this involves a fully pipelined calculation of partial results, followed by a final reduction stage using a tree-like pattern. For the filter operations, an efficient line buffer mechanism is used to fully pipeline the operations.

B. Interfacing

Many researchers seem to make full use of the PYNQ framework to interface with the FPGA from a Python environment. This involves the creation of hardware drivers in Python, binding them to specific functions and invoking them from the PYQN enviroment, which keeps track of the hierarchy and status of the hardware design at all times. While this mechanism is very convenient and allows for rapid prototyping, it also seems to have a vast overhead, especially when accelerators are invoked repeatedly. In this work, lower-level C/C++ drivers are created for all accelerators directly and are bound to Python using the CFFI and pybind11 libraries. A tool is created to aid the developer with the creation of these drivers by generating a Jupyter Notebook that contains most of the boiler-plate code that is required to interface with the DMAs, switches, and individual accelerators. Furthermore, the tool also provides a timing analysis such as shown in Figure 2, which allows to determine the minimum size of the input stream for which a hardware acceleration is profitable, i.e. provides speedup when compared to the default NumPy implementation.

C. Software implementation

A Python module, named `ZyPy`, was created that acts as a drop-in replacement for NumPy, i.e. it is possible to use all accelerated and default NumPy functionality using the `import zypy as np` statement.

On the one hand, the module keeps track of all available hardware accelerators in a dictionary structure as shown in Listing 1, in which each method (e.g. `add.__call__`) is mapped to possibly multiple hardware implementations (e.g. `add_call_f4_cffi`).

```
hw_accelerators = {
  "add.__call__": {
    "add_call_f4_cffi": {
        "driver": add_call_f4_cffi,
        "accepts": lambda args, kwargs: len(args) == 2 and
                  args[0].shape == args[1].shape and
                  args[0].dtype == args[1].dtype ==
                  ↪ np.float32,
        "provides_acceleration": lambda x: x > 0,
        "provides_acceleration_reconfig": lambda x: x >
        ↪ 524288,
        "bitfile": "zypy/overlays/ufunc_call_f4.bit",
        "imports": "from lib.ufunc_call_f4_cffi import lib
        ↪ as calc",
      },  # ...
    },  # ...
}
```

Listing 1. The `ZyPy` module makes us of a dictionary to store information about all available accelerators on the system and to link them with their respective Python method calls.

Each implementation requires the following fields:

- `driver`, indicating which lower-level driver to invoke when the method call can be accelerated,
- `accepts`, which returns whether the given set of input arguments is compatible with the driver,
- `provides_acceleration`, which returns whether it is beneficial to invoke the hardware accelerator or not, given the size of the input (based on the timing analysis presented in Section IV-B),
- `provides_acceleration_reconfig`, which is the same as the previous field, but in which the reconfiguration overhead (≈0.2 seconds) is also taken into account,

[4]See https://numpy.org/doc/stable/reference/ufuncs.html

- `bitfile`, which is the path to the bitfile that should be loaded onto the FPGA, and
- `imports`, which a string that specifies which CFFI or pybind11 modules need to be imported when the corresponding bitfile is loaded for the first time.

On the other hand, the module overloads all hardware accelerated methods, and in a certain sense, is able to intercept method calls, check if they can be accelerated, forward the arguments to the appropriate driver in case an acceleration is possible, and fall back to the default implementation when this is not the case.

V. EXPERIMENTAL RESULTS

A. Test setup

All tests were carried out on the PYNQ-Z2 board (ZYNQ XC7Z020-1CLG400C), which includes a dual-core Cortex-A9 processor clocked at 650Mhz and 512MB of DDR3 memory. Its programmable logic runs at 100MHz and consists of 13,300 logic slices, 630 KB of BRAM, and 220 DSP slices.

B. Performance of the hardware accelerators

Different FPGA-accelerated implementations of the element-wise multiplication (`multiply_call_f4`) operation were compared with the NumPy implementation for different values of the input stream length, as is shown in Figure 2. The implementations make use of exactly the same hardware design, but use different drivers: the PYNQ implementation uses Python drivers, while the other implementations make use of lower-level C drivers that are bound to Python using CFFI and pybind11. On the left vertical axis, the run time per element can be observed when looking at the solid curves. The speedup relative to the NumPy implementation can be read off from the right vertical axis, when only considering the dotted lines. As expected, the interfacing overhead is more noticeable for smaller input streams. For large values of the input size, the CFFI and pybind11 implementations approach a run time of 10 ns per element while the NumPy implementation takes more than 300 ns per element, leading to a speedup of a factor of 35 for large input streams. From the graph, it can also be seen that the PYNQ implementation is lagging well behind and only provides acceleration when the input stream length approaches one million elements. Surprisingly, the CFFI implementation even outperforms the NumPy implementation for input streams containing only a dozen of elements.

C. Performance of the ZyPy module

Figure 3 shows the total run time of benchmark in which the heat equation is solved on a square grid, versus the size of the input stream, for three different implementations: NumPy, PYNQ and ZyPy. The PYNQ implementation makes use of the same hardware accelerators as the ZyPy version, but not of the run-time overloading mechanism that it uses. As can be seen from the graph, the ZyPy implementation clearly outperforms the NumPy implementation, in particular

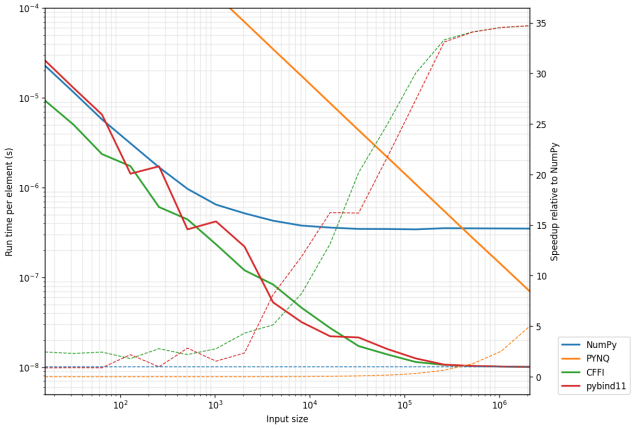

Fig. 2. Comparison of the different FPGA-accelerated implementations of the `multiply_call_f4` operation with the NumPy implementation, for different values of the input stream length. On the left vertical axis, the run time per element can be read off when combined with the solid coloured lines. On the right vertical axis, the speed-up relative to NumPy can be observed, when only considering the dashed coloured lines.

for large inputs. For 64x64 inputs, the ZyPy implementation provides a speedup of a factor close to 2, while for 384x384 inputs, the ZyPy implementation is well over 10x faster than the NumPy implementation. Extrapolating the results to larger input sizes suggests that the run time of the NumPy implementation is nearly two orders of magnitude higher than the ZyPy-accelerated version, for input arrays as small as 512x512. As expected, the PYNQ implementation suffers from huge overheads and is therefore not suitable for this kind of application.

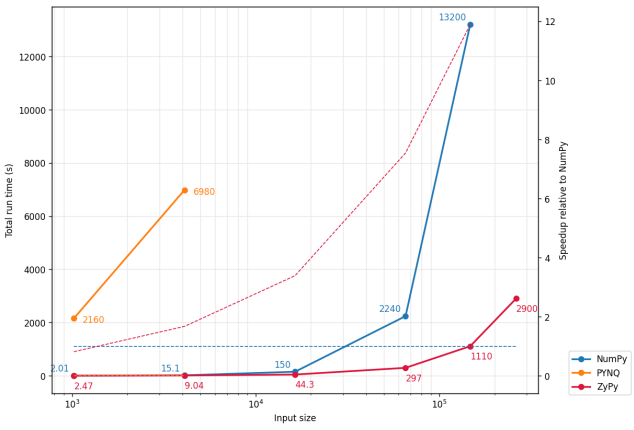

Fig. 3. Comparison of the run time of the Numpy, PYNQ and ZyPy implementations of the heat equation benchmark, for different values of the input stream length. On the left vertical axis, the total run time of the benchmark can be read off when combined with the solid coloured lines. On the right vertical axis, the speed-up relative to NumPy can be observed, when only considering the dashed coloured lines.

VI. CONCLUSION AND FUTURE WORK

In this work, a methodology is devised that allows numerical applications in Python to be accelerated using FPGAs. In order to validate this methodology, a custom Python module that can be used as a drop-in replacement of NumPy, is implemented, in which routines are automatically and transparently forwarded to their hardware-accelerated counterpart on the FPGA, given that a compatible and profitable accelerator exists. To support this system, various NumPy routines are transformed into custom hardware accelerators by means of high-level synthesis, and integrated into Python using parts of Xilinxs PYNQ platform. More concretely, three classes of NumPy operations are covered and optimised in hardware: element-wise operations on `ndarrays`, reduction-type operations on `ndarrays` and filter-type operations on two-dimensional `ndarrays`.

Furthermore, a reusable hardware block design is created that allows multiple hardware accelerators to be combined, by making use of a stream switch. A tool is also devised that eases the creation of lower-level software drivers for each of the hardware designs. A speed-up by an order of magnitude is found to be possible in a practical application where the two-dimensional heat equation is solved numerically, while a direct invocation of the individual accelerators is able to outperform default NumPy operations by factors up to 35x. It is also concluded that the PYNQ framework, despite its flexibility and prototyping capabilities, is not able to achieve this kind of acceleration, which led to various parts of the framework being bypassed.

Even though the results seem to be encouraging, there is still potential for improvement and further research, in particular by creating more hardware accelerators, by increasing the throughput of the existing accelerators, by removing the dependency on the PYNQ framework, and by making use of scatter/gather DMAs. It also seems useful to further investigate the possibilities of linking multiple operations together, both in software (e.g. lazy evaluation combined with template matching), and in hardware (e.g. by introducing FIFO stages). Finally, the use of dynamic partial reconfiguration could lead to a better utilization of the available FPGA resources and could even enable the ability to load in the accelerator of the next operation, while the current operation is still in progress.

REFERENCES

[1] L. Stornaiuolo, F. Carloni, R. Pressiani, G. Natale, M. Santambrogio, and D. Sciuto, "Enabling transparent hardware acceleration on Zynq SoC for scientific computing," *ACM SIGBED Review*, vol. 17, pp. 30–35, 07 2020. [Online]. Available: https://doi.org/10.1145/3412821.3412826

[2] A. G. Schmidt, G. Weisz, and M. French, "Evaluating Rapid Application Development with Python for Heterogeneous Processor-Based FPGAs," *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 121–124, 2017. [Online]. Available: https://doi.org/10.1109/FCCM.2017.45

[3] B. Janßn, P. Zimprich, and M. Hübner, "A dynamic partial reconfigurable overlay concept for PYNQ," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4. [Online]. Available: https://doi.org/10.23919/FPL.2017.8056786

[4] J. Decaluwe, "MyHDL: a Python-based hardware description language," *Linux Journal*, vol. 2004, p. 5, 2004. [Online]. Available: https://api.semanticscholar.org/CorpusID:195996891

[5] P. Haglund, O. Mencer, W. Luk, and B. Tai, "Hardware Design with a Scripting Language," vol. 2778, 09 2003, pp. 1040–1043. [Online]. Available: https://doi.org/10.1007/978-3-540-45234-8_115

[6] D. Lockhart, G. Zibrat, and C. Batten, "PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research," *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 280–292, 2014. [Online]. Available: https://doi.org/10.1109/MICRO.2014.50

[7] E. Logaras, O. G. Hazapis, and E. S. Manolakos, "Python to Accelerate Embedded SoC Design: A Case Study for Systems Biology," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4, March 2014. [Online]. Available: https://doi.org/10.1145/2560032

[8] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood, "A pythonic approach for rapid hardware prototyping and instrumentation," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–7. [Online]. Available: https://doi.org/10.23919/FPL.2017.8056860

[9] R. Peschke, K. Nishimura, and G. Varner, "ARGG-HDL: A High Level Python Based Object-Oriented HDL Framework," 2020. [Online]. Available: https://arxiv.org/abs/2011.02626

[10] D. H. Noronha, B. Salehpour, and S. J. E. Wilton, "LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks," 2018.

[11] S. Skalicky, J. Monson, A. Schmidt, and M. French, "Hot & Spicy: Improving Productivity with Python and HLS for FPGAs," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 85–92. [Online]. Available: https://doi.org/10.1109/FCCM.2018.00022

[12] Y. Uguen and E. Petit, "PyGA: a Python to FPGA compiler prototype," *Proceedings of the 5th ACM SIGPLAN International Workshop on Artificial Intelligence and Empirical Methods for Software Engineering and Parallel Computing Systems*, 2018. [Online]. Available: https://doi.org/10.1145/3281070.3281072

[13] S. Huang, K. Wu, H. Jeong, C. Wang, D. Chen, and W.-m. Hwu, "PyLog: An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 227228. [Online]. Available: https://doi.org/10.1145/3431920.3439478

[14] R. Okuta, Y. Unno, D. Nishino, S. Hido, and Crissman, "CuPy : A NumPy-Compatible Library for NVIDIA GPU Calculations," 2017. [Online]. Available: https://www.semanticscholar.org/paper/CuPy-%3A-A-NumPy-Compatible-Library-for-NVIDIA-GPU-Okuta-Unno/a59da4639436f582e483347a4833e7659fd3e598

[15] T. Higuchi, N. Yoshifuji, T. Sakai, Y. Kitta, R. Takano, T. Ikegami, and K. Taura, "ClPy: A NumPy-Compatible Library Accelerated with OpenCL," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 933–940. [Online]. Available: https://doi.org/10.1109/IPDPSW.2019.00159

[16] M. Kristensen, S. Lund, T. Blum, K. Skovhede, and B. Vinter, "Bohrium: Unmodified NumPy Code on CPU, GPU and Cluster," 11 2013. [Online]. Available: https://www.researchgate.net/publication/281647460_Bohrium_Unmodified_NumPy_Code_on_CPU_GPU_and_Cluster

[17] T. Blum, M. R. Kristensen, and B. Vinter, "Transparent GPU Execution of NumPy Applications," *2014 IEEE International Parallel and Distributed Processing Symposium Workshops*, pp. 1002–1010, 2014. [Online]. Available: https://doi.org/10.1109/IPDPSW.2014.114

[18] Stornaiuolo, Luca, "Exploiting FPGA from Data Science Programming Languages," Master's thesis, Politecnico di Milano, 2017.

[19] A. Ismail and L. Shannon, "FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators," 06 2011, pp. 170 – 177. [Online]. Available: https://doi.org/10.1109/FCCM.2011.48

[20] A. Agne, M. Happe, A. Keller, E. Lbbers, B. Plattner, M. Platzner, and C. Plessl, "ReconOS: An Operating System Approach for Reconfigurable Computing," *Micro, IEEE*, vol. 34, pp. 60–71, 01 2014. [Online]. Available: https://doi.org/10.1109/MM.2013.110

[21] W. Kritikos, A. Schmidt, R. Sass, E. Anderson, and M. French, "Redsharc: A Programming Model and On-Chip Network for Multi-Core Systems on a Programmable Chip," *International Journal of Reconfigurable Computing*, vol. 2012, 01 2012. [Online]. Available: https://doi.org/10.1155/2012/872610

Contents

List of Figures

List of Tables

List of Acronyms

ABI	application binary interface
ACP	accelerator coherency port
API	application programming interface
APU	application processing unit
ASIC	application-specific integrated circuit
AXI	Advanced eXtensible Interface
BD	buffer descriptor
BRAM	block random-access memory
BSP	board support package
CFFI	C Foreign Function Interface
CLB	configurable logic block
CMA	contiguous memory allocation
CPU	central processing unit
DDR	double data rate
DFX	dynamic function exchange
DMA	direct memory access
DMAC	direct memory access controller
DPR	dynamic partial reconfiguration
DRAM	dynamic random-access memory
DSP	digital signal processor
FF	flip-flop
FFT	fast Fourier transform
FIFO	first in, first out
FPGA	field-programmable gate array
FSBL	first-stage boot loader
GP	general purpose
GPIO	general purpose input/output
GPU	graphics processing unit
HDL	hardware description language

HLS	high-level synthesis
HP	high-performance
HPC	high-performance computing
I/O	input/output
I2C	Inter-Integrated Circuit
II	initiation interval
IL	iteration latency
IOB	input/output block
IP	intellectual property
L1	level 1
L2	level 2
LKM	loadable kernel module
LUT	lookup table
MAC	multiply-accumulate
MM2S	memory-mapped to stream
MMIO	memory-mapped I/O
MMU	memory management unit
MPSoC	multiprocessor system on a chip
NoC	network on a chip
OCI	on-chip interconnect
OCM	on-chip memory
OS	operating system
PL	programmable logic
PLC	programmable logic circuit
PLL	phase-locked loop
PS	processing system
PSU	power supply unit
PYNQ	Python productivity for Zynq
RP	reconfigurable partition
RTL	register-transfer level
S2MM	stream to memory-mapped

SCU	snoop control unit
SD	Secure Digital
SG	scatter/gather
SoC	system on a chip
SSN	stream switch network
UIO	user space I/O
VHDL	VHSIC hardware description language
XML	Extensible Markup Language
XRT	Xilinx runtime

1

Introduction

1.1 Problem statement

In the past decades, there has been an increased interest in the design of heterogeneous computing systems to improve the performance of computational systems while keeping the energy consumption low. A heterogeneous computing system refers to a system that contains different types of computational units, such as multi-core central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs), and application-specific integrated circuits (ASICs). CPUs can efficiently run generic tasks, GPUs are optimal for massively parallel repetitive tasks, and FPGAs can be configured to provide parallel and/or pipelined hardware implementations of a set of instructions for an efficient execution [1].

The computational units in a heterogeneous system typically include a general-purpose processor that runs an operating system. Processors other than the general-purpose processor are called *accelerators* because they accelerate a specific type of computation by assisting the general-purpose processor [2]. Heterogeneous computing systems can be found in every domain of computing, from high-end servers and high-performance computing (HPC) all the way down to low-power embedded devices, including mobile phones and tablets. Platforms usually combine multiple heterogeneous processing elements with a memory hierarchy, I/O components and other

specific functionalities according to the needs of the specific domain. The different components are linked to each other using on-chip interconnects (OCIs), such as buses and networks on a chip (NoCs). These types of platforms are usually referred to as systems on a chip (SoCs) or – in case there are multiple microprocessors – multiprocessor systems on a chip (MPSoCs) [3].

In 2011, Xilinx introduced the Zynq-7000 family of SoCs, whose architecture integrates a dual-core ARM Cortex-A9 processor with an 28nm Artix-7 or Kintex-7 based programmable logic. The general-purpose ARM processor enables booting immediately at power-up and allows a software developer to run a variety of operating systems – including Linux – independently of the programmable logic [4]. The integration of FPGA-based hardware accelerators into applications still requires specific skills and knowledge of low-level tools that are out of reach for the largest part of software developers, which still hinders a more widespread adoption. The use of high-level synthesis (HLS), in which a high-level algorithmic description is used to create digital hardware instead of a more tedious description at the register-transfer level (RTL), partially mitigates some of these difficulties, but it still essentially requires the same development process [5].

In an attempt to make it easier for designers of embedded systems to exploit the benefits of their devices, Xilinx released PYNQ (Python productivity for Zynq)[1] in 2016. PYNQ tries to achieve this goal by allowing developers to build complex and capable applications very quickly, using the popular Python programming language along with its wide ecosystem of over 300,000 libraries [9, 10]. In PYNQ, programmable logic circuits (PLCs) are presented as hardware libraries called *overlays*, which are analogous to software libraries. The hardware functionality is made available to the user through a Python API that allows the developer to select the overlay that best matches their application. Creating a new overlay still requires expertise in designing programmable logic circuits, but they are designed to be configurable and re-used as often as possible in many different applications, adhering to the *build once, re-use many times* paradigm [11]. In recent years, the PYNQ open-source community has produced a reasonable amount of projects that demonstrate the capabilities of the framework, with applications ranging from signal processing, video filtering, compression and encryption to real-time object detection and neural network acceleration [12].

Unfortunately, most important and widely used numerical libraries in Python, such as NumPy and SciPy have remained relatively untouched [13, 14]. When these libraries are used, they are directly executed on the ARM-processor and, hence, do not take advantage of the programmable logic circuits, even though many of the computational kernels accommodate hardware acceleration. In this thesis, this aspect is further elaborated on and efforts are made to speed up these libraries using FPGA-based accelerators.

[1] It should be noted that – despite its name – PYNQ is not restricted to the Zynq-architecture, but also supports other XRT-based platforms including Amazon AWS EC2 F1 and Alveo for cloud and on-premise deployment [6, 7, 8].

1.2 Thesis goals

The goal of this thesis is to analyse and accelerate (parts of) numerical libraries for Python by implementing them in hardware, which is expected to give an order of magnitude speed-up for a typical numerical application. The research focusses on optimising strategies for creating parallel and pipelined hardware cores and their organisation in memory, defining the best practices using high-level synthesis reports as a means to guide the design space exploration, and analysing a typical numerical application in Python with the aim of transparently selecting the most promising hardware procedures based on communication, computation and resource requirements.

More concretely, this entails:

(i) Exploring the interface between the interpreted language Python, the Linux operating system, the memory management and low-level C/C++ routines.

(ii) Implementing selected Python library procedures in hardware, using high-level synthesis as a tool to simplify their development.

(iii) Optimising the hardware library routines for maximum acceleration, taking into account the available resources and the requirements of the program.

(iv) Creating a mechanism in Python that allows to transparently invoke a routine from the hardware library at run-time.

(v) Accelerating a numerical application in Python and determining the obtained speed-up. In this work, finding the numerical solution to the two-dimensional heat equation, on a discrete grid with given boundary conditions, is put forward as the target application. A particular solution to it is visualised in Figure 1.1.

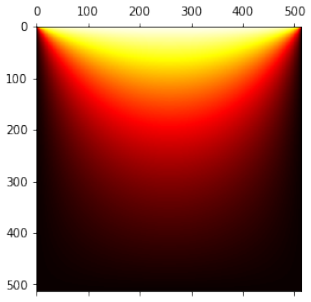

Figure 1.1: *In this work, finding the numerical solution to the two-dimensional heat equation, on a discrete grid with given boundary conditions, is put forward as the target application.*

1.3 Thesis organization

This thesis is organised as follows:

- Chapter 2 first presents some background information, followed by the methodology that will be adhered to. Finally, the related work in the field is presented.
- Chapter 3 describes the implementation of a generic hardware block design that facilitates the creation of hardware libraries, and also addresses the creation and the high-level synthesis implementation of several accelerator cores, taking into account interfacing, performance optimisation, and resource utilisation aspects.
- Chapter 4 goes into more detail on the interfacing between the programmable logic and the processing system on SoCs. More precisely, the communication between the FPGA and the CPU without an operating system, with an operating system and with an operating system from a Python environment is considered.
- Chapter 5 presents the design and implementation of a Python module that manages the available hardware accelerators and allows for the overloading of Python method calls in order to accelerate them.
- Chapter 6 evaluates the experimental results that are obtained by benchmarking the implemented hardware accelerators, and the Python module that is used to manage them.
- Chapter 7 concludes the work and provides an overview of topics that may be of interest for further research.

2

Background, methodology and related work

This chapter first presents some background information, more specifically on the Zynq-7000 family of systems on a chip, and on the use of numerical libraries in Python. Next, the methodology that will be adhered to, in order to accelerate numerical procedures in Python, is outlined. Finally, the related work in the field is presented.

2.1 Background

2.1.1 The Zynq-7000 family

Architecture

The Xilinx Zynq-7000 family, to which the development board used in this thesis belongs to, is a class of systems on a chip (SoCs) that integrates an ARM Cortex-A9 based hard-core processor and a Xilinx 7-series based FPGA onto a single chip, with the ARM core being the central component, and the FPGA acting as slave. The main components of the system are illustrated in Figure 2.1. The processing system (PS) is made up of the application processing unit (APU), a central interconnect and several interfaces to the outside world. The core of the processing system consists of the two ARM Cortex-A9 CPUs, each with their own L1 data and instruction

caches, which connect to the L2 cache and on-chip memory (OCM) via the snoop control unit (SCU), which maintains data coherency between the different cores. The main dynamic random-access memory (DRAM) is connected via a memory controller, and additional controllers also allow the system to interface with a plethora of peripheral devices.

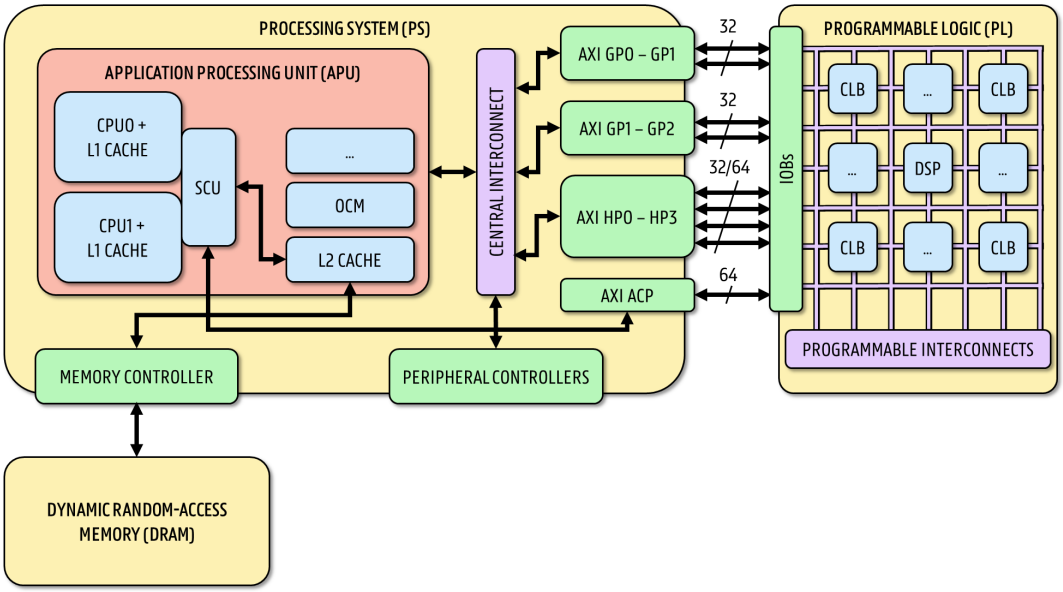

Figure 2.1: *Xilinx Zynq-7000 SoC architecture.*

The FPGA fabric and all supporting circuitry, such as clock managers and configuration modules, form the programmable logic (PL). The FPGA consists of three types of elements: configurable logic blocks (CLBs), programmable interconnects and input/output blocks (IOBs). The configurable logic blocks are responsible for performing logic operations and storing their results. The programmable interconnects are used to link together configurable logic blocks into larger circuits to meet the required functionality. Finally, the input/output blocks connect the internal circuitry to the external pins of the FPGA and enable communication with other parts of the hardware system, such as external clocks, external memories, general purpose processors and even other FPGAs [15]. The programmable logic also contains additional computational and data storage blocks that increase the density and efficiency of the device. These elements include embedded memories for distributed data storage, phase-locked loops (PLLs) for driving the FPGA fabric at different clock rates, high-speed serial transceivers, off-chip memory controllers and multiply-accumulate (MAC) blocks that are used as digital signal processors (DSPs). This variety of components allows the FPGA to implement virtually any software algorithm running on a microprocessor [16].

Interfacing between the PS and the PL

Components on the Zynq SoC mostly communicate using the AXI4 protocol, which includes support for separate control and data phases, allows burst-based, outstanding and out-of-order transactions, and supports unaligned transfers[1]. There are three categories within the AXI4 standard:

- **AXI4-full**, which is suitable for high-performing memory-mapped[2] interfaces and allows bursts of up to 256 data transfer cycles, with just a single address phase.

- **AXI4-lite**, which is a subset of AXI4-full that provides a register-like structure with reduced features and complexity. It has a small logic footprint and is suitable for simple low-throughput memory-mapped communication.

- **AXI4-stream**, which completely removes the requirement for an address phase and allows for unlimited data bursts. It is suitable for high-speed streaming data. Besides the main data channel, an AXI4-stream signal also contains a minimal set of side channels to, for example, indicate the end of the stream or to mark the validity of individual bytes in the data channel.

As shown in Figure 2.1, the processing system and programmable logic can communicate using three types of AXI interfaces: HP, GP and ACP. The high-performance (HP) ports will be used to transfer the bulk of the data between the PS and the PL, making use use of direct memory access (DMA)[3] components to convert memory mapped regions into data streams and vice versa. The general purpose (GP) ports on the other hand, will be used to control different cores on the FPGA, typically using the AXI4-lite protocol. The third type of interface, the accelerator coherency port (ACP), which provides cache coherent access to the memory, will not be used.

[1] Unaligned memory accesses occur when N bytes of data are read from a memory address that is not evenly divisible by N (i.e. addr $\% N \neq 0$) [17].

[2] In memory-mapped I/O (MMIO), the same address space is used to address both memory and peripheral devices.

[3] DMA is a feature of a hardware system that makes it possible to transfer data between the system's main memory and some other location with minimal intervention of the CPU. The usage of DMA allows other instructions to be executed on the CPU while a transfer is in progress.

2.1.2 Numerical libraries for Python

Python is a general-purpose programming language that combines expressive power with specialized libraries for rapid program development. It is a high-efficiency programming language that prioritises productivity over performance, but nevertheless, its popularity has also been growing in the numerical computing community over the last decade. Although it may seem counter-intuitive, it is in fact easy to explain since Python allows for an easy expansion with highly optimised and highly performing libraries written in languages like C, C++ and FORTRAN.

The de-facto standard for scientific applications written in Python is NumPy, which provides a rich set of high-level numerical operations and is targeted at the CPython reference implementation. Because of CPython's interpreted nature, many mathematical algorithms written for this version of Python run much slower than their compiled equivalents. NumPy addresses this problem by providing a powerful multidimensional array object (`ndarray`) that promotes the use of vector programming/array programming/vectorisation, in which numerical operations and functions operate on complete arrays rather than on individual scalars. While this style often requires rewriting some code, mostly inner loops, it also allows for a flexible and more optimised implementation.

Another advantage of NumPy is that it allows to easily change the implementation of the array object (and operations on it) without breaking the compatibility with existing code. Furthermore, the `ndarray` is simply a strided view on contiguous memory (note that `ndarrays` are homogeneously typed, unlike Python's built-in `List` object), which allows NumPy to represent buffers that have been created and most oftenly manipulated by low-level extensions, without having to copy around data. These factors make NumPy the ideal canditate to be accelerated using dedicated hardware, and FPGAs in particular in this work.

Note that, while this work may be focussed towards NumPy and its `ndarray` object, the concepts and methodology put forward in this thesis are also portable to possible other numerical libraries that are not discussed here. It should also be noted that a tremendous amount of scientific libraries build on NumPy and its arrays, in domains ranging from signal processing and statistical computing to bioinformatics and quantum computing, and can therefore benefit from any possible acceleration as well [4].

[4] These include for example SciPy, OpenCV, NetworkX, BioPython, Pandas and Seaborn. For a more in-depth overview please visit https://numpy.org.

2.2 Methodology

In this section, an overview is given of the methodology that is used in order to accelerate numerical Python applications. Developers that aim to extend the results of this thesis can follow the same approach. Note that the methodology is formulated in a generic way and is not particularly bound to specific platforms, architectures, systems or software tools. The process is described below:

1. **Identification of the procedures that will be implemented in hardware.**
   An FPGA can accelerate a wide variety of operations, and a choice must be made as to what operations will be performed on the hardware. This choice can be based on the specific application in mind, or it is also possible to take a more generic approach and pick functionalities that are useful in many different scenarios. In this thesis, a combination of both is used: solving the heat equation is put forward as an application, but by generalising the operations that are required for this purpose to "classes of operations", it is possible to speed up many more different programs as well. Alternatively, it is also possible to profile an existing numerical Python application and implement the procedures that takes the longest to execute.

2. **Algorithmic description of the selected procedures and the introduction of streaming behaviour.**
   To be able to implement the selected procedures in hardware, an algorithmic description of each of them is made first. In order to create an efficient and, most importantly, fast hardware accelerator, the use of input streams and output streams to represent (multidimensional) arrays is strongly encouraged. This means that in the algorithmic description, it must (conceptually) be taken into account that the inputs enter the system one element at a time, and also leave the system one element at a time.

3. **Implementation of the selected procedures on hardware using high-level synthesis (HLS) tools.**
   High-level synthesis allows the developer to describe the hardware design at a high level of abstraction, using an algorithmic description in for example C or C++. A high-level synthesis tool such as Vitis HLS transforms (synthesises) this description into a register-transfer level (RTL) design in a hardware description language such as Verilog or VHDL. At this stage, it is also possible to verify the correctness of the implementation by writing a C or C++ test bench. In this way, it is possible to validate the functional design, both before synthesis (using a C simulation), and after synthesis (using software/hardware co-simulation).

4. **Optimisation of the hardware accelerators.**

Based on the synthesis results, an estimate can be made of the throughput, latency and resource consumption of the hardware accelerators. Often, additional directives (or pragmas) are required to further optimise the design and achieve a higher throughput, a reduction of the latency of the core, or a reduction of the required area or device resource usage [18]. In this step, the synthesis reports provide feedback to the developer and can guide the design space exploration towards an optimal or nearly optimal solution.

5. **Creation of a hardware block design that integrates multiple accelerators.**
   In order to make use of the created accelerators, they must be incorporated into a hardware design that allows them to communicate with the processing system (PS) of the SoC, on which the target application is running. This entails the use of external, prefabricated components or intellectual property (IP) cores, which fulfil a certain functionality within the design. This may involve, for example, moving data from the main memory to the hardware accelerators in a streaming fashion (and vice versa), or linking the different accelerators together. At this stage, it is beneficial to add as many accelerators as possible to the design, to make full use of the available resources. In this work, a generic block design is created using Vivado , and can be used as a template for this purpose. The output of this stage is a bitstream file that can be loaded onto the FPGA.

6. **Creation of C/C++ drivers that can interface with the block design.**
   In order for an application to access the accelerators, it is necessary to create C or C++ drivers that are responsible for the communication. While it is possible to directly write drivers for the Linux operating system, it is advised to test the interfacing and the functionality of the hardware design on a platform without an operating system, i.e. running *bare metal*. Software tools such as Vitis are suitable for this purpose and also allow the use of a variety of pre-made and auto-generated drivers. In addition, they also allow the software to be debugged, which is more difficult in a running operating system. In a later stage, it is then possible to translate the functionality into Linux drivers, once the correct way to interface with the hardware design has been found. In this work, a tool has been created that aids the developer in creating Linux drivers, using automatically generated Jupyter Notebooks.

7. **Binding of the C/C++ drivers to a Python module.**
   To interface with the accelerators on the FPGA from Python, the C/C++ drivers need to be bound to a module, which can be done in various ways, for example using CFFI or pybind11. As part of this thesis, a Python module was created that keeps track of all available hardware accelerators in a library and allows their C/C++ drivers to be called in a convenient manner. If several hardware designs are used interchangeably, it is also necessary to create a system that allows the correct bitstream to be loaded onto the FPGA before invoking the drivers.

8. **Profitability analysis of the hardware accelerators.**

When the drivers are in place and can be invoked from a Python program, it is also possible to check the performance gain that is achieved by invoking the hardware-accelerated version of a specific procedure instead of the original Python implementation. In many cases, the hardware accelerators provide only a minimal acceleration when the input streams are small, because of the overhead that is associated with the PS/PL interaction. In this work, a benchmark is done for each Python procedure that is to be accelerated, to find the minimals size of the input stream for which the hardware accelerator becomes profitable.

9. **Acceleration of numerical applications in Python.**
   Finally, it is possible to accelerate a given numerical application by invoking the corresponding drivers at run-time instead of the original procedures. As part of this thesis, a mechanism was created that automatically and transparently forwards method calls to their respective hardware accelerated counterparts, if the functionality of the method is implemented and it is profitable and possible to do so. At this point, it is also possible to obtain insights into the obtained speed-up.

2.3 Related work

In this section, various studies related to this work are briefly explained in three subsections. First, the acceleration of Python using FPGAs in general is outlined. Next, the transparent acceleration of NumPy in particular, is discussed in more detail. Finally, there is a glance at mechanisms that allow for the cooperation of different hardware accelerators and the processing system, and at ways to abstract this away from the software developer.

Accelerating Python using FPGAs

The possibilities offered by the PYNQ framework by Xilinx, whose goal is to accelerate Python using FPGAs, have already been extensively explored. Countless applications can be found in the areas of machine learning (e.g. used to accelerate convolutional and recurrent neural networks, pointer network models, support vector machines, Gaussian processes...), image processing and computer vision (e.g. lane line detection, autonomous driving systems, SIFT, ORB, ...), digital signal processing (e.g. software defined radios and modulation, electroencephalography, audio alignment, filtering...), data analytics (e.g. Apache Spark), security (e.g. encryption) and many more. Without going into detail on the specific applications, they share a common denominator: achieving speedups and power savings by implementing software procedures on hardware.

Some authors take a more generalist approach or expand the framework, for example in [19], where the impact of using PYNQ on Zynq devices, its performance implications, and its bottle-

necks are investigated based on a combined Gaussian blur and Canny edge detection hardware design. Using the Redsharc stream switch network (see 2.3), they achieve a speedup of a factor three over the OpenCV implementation running on the CPU. Other work includes that of [20], in which the functionality of PYNQ is extended to support dynamic partial reconfiguration, which allows to reconfigure parts of the programmable logic while the other parts remain active. This allows for a more efficient and flexible use of the available FPGA resources but also introduces additional complexity. The authors report a 40% cut in resource usage, which is why this module could, in a future stage, be incorporated into this work to accomodate more hardware accelerators on the FPGA.

Furthermore there exists several works, including MyHDL [21], PyHDL [22], PyMTL [23], SysPy [24], PyRTL [25] and ARGG-HDL [26], that aim to use Python as a high-level hardware description language, in an attempt to reduce the development time, and bridge the gap between low-level and high-level hardware design. Other research tries to convert Python code to a high-level synthesisable language, such as in LeFlow [27], Hot & Spicy [28], PyGA [29] and PyLog [30]. The latter two even integrate the synthesis, bitstream generation and host program creation into a single flow, leaving the developer with virtually no further work. The PyGA compiler generates host code in C++ and kernel code in OpenCL, which is translated into an LLVM intermediate representation and converted into bitstreams using the Intel FPGA SDK for OpenCL, and finally runs the design using the PyGA runtime. Similarly, PyLog takes in Python functions, generates a PyLog intermediate representation, performs several optimization passes, including pragma insertion, design space exploration, and memory customization, and finally creates the complete FPGA system design. Even though such systems significantly improve the developer's productivity, they don't seem to be able to outperform highly optimised CPU implementations without FPGA-specific programming efforts. In PyGA, the results are not competitive with optimised CPU implementations, while in PyLog only a speedup of a factor of three is achieved. Moreover, such systems are not appropriate within the context of this thesis, since in this work numerical Python methods (e.g. NumPy, SciPy) are accelerated, which themselves often interface with highly optimised C, C++ and even FORTRAN code instead of Python code that can be synthesised using such systems.

Accelerating NumPy using FPGAs

The concept of transparently accelerating NumPy on hardware is also not new, but most research so far has been targeting GPUs, including CuPy and ClPy[5] [32, 33]. A more general approach is taken by the Bohrium project [34], in which a runtime system is created that maps array operations onto a number of different hardware platforms. It realises this by using a front-end

[5]Not to be confused with PyCUDA and PyOpenCL, which provide easy and Pythonic access to NVIDIA's CUDA and OpenCL parallel computation APIs respectively [31].

component that compiles the user code into a Bohrium bytecode, which in its turn can be executed by architecture-specific implementations such as GPUs [35]. To reduce the overhead related to generating and processing the bytecode, the bridge that connects the front-end and back-end uses lazy evaluation for recording instructions until a side effect can be observed. This technique allows to optimise for memory usage (e.g. by removing temporary arrays) and also facilites the combined execution of multiple statements if the back-end supports it. In the original article, the authors mention the future implementation of a Bohrium back-end that enables support for FPGAs as well, however this route seems to be discontinued.

In [36], the same authors also present a back-end framework that allows for the separation of the NumPy API from its implementation. Even though some of the proposed concepts are outdated, because NumPy nowadays offers standardised mechanisms to adapt the behaviour of `ndarrays`[6], the idea of creating a custom array class as drop-in replacement of the NumPy array to separate the front- and back-end, is also adopted in this work.

The work that ties in most strongly with this thesis is that of [37, 5], in which the authors build upon the PYNQ framework to enable transparent hardware acceleration for scientific computations on Zynq platforms. They provide implementations for the correlation function, matrix dot product, standard deviation and the FFT, and validate their work using a biomedical use case. Similar to this work, they provide a run-time mechanism in which the accelerators can be transparently invoked by overloading standard NumPy methods. They also make use of a system that decides between software and hardware acceleration depending on the history of invocations. This contrasts with the system that is employed in this work, in which semi-automated benchmarks are used for this purpose. Also, they rely on writing lower-level drivers in CPython directly, as opposed to CFFI or pybind11, and provide no possibility to integrate multiple accelerators into a single hardware design, with an increased reconfiguration overhead as a result.

Hardware/software development environments

There also have been several studies, in particular in the area of reconfigurable computing, that consider the unification of hardware and software design on SoCs and MPSoCs, together with the creation of systems that attempt to relieve software developers of the burden of managing the hardware and associated data transfers at run-time.

In [38], FUSE is introduced, which is a framework that provides operating sytem abstraction of hardware accelerators. The authors propose a mechanism that allows user applications to access low-level hardware in a safe and transparent manner, by presenting the hardware accelerators to

[6] https://numpy.org/doc/stable/user/basics.dispatch.html#writing-custom-array-containers

software designers as *hardware tasks* that can be interacted with from a multithreaded software environment. The principle of hardware threads is also adopted by the ReconOS creators [39], in which the operating system interface allows hardware threads to interact with software threads using well known mechanisms including semaphores, mutexes, condition variables and message queues. While there are several similarities between these systems and the one presented in this thesis, including the use of lookup mechanisms to bind accelerators to their respective tasks or methods, the use of memory-mapped I/O to communicate with the accelerators, and the creation of decision trees to instantiate either hardware or software threads, there are also several differences and disadvantages of such systems within this context. The use of loadable kernel modules (LKMs) to control and access the local memory of each individual accelerator makes the design process more complex, and the on-demand loading of these modules also creates a certain overhead. In the case of ReconOS, a middleware layer is used that takes a considerable amount of the available resources on the FPGA. Furthermore, the hardware developer controls whether an accelerator is profitable for a certain task or not, while in the system proposed in this thesis, it is under the control of the software developer. Finally, these systems make use of (proprietary) shared communication buses, while standard on-chip interconnects for FPGAs do not scale very well and cause contention, which can quickly limit the attainable performance.

In this work, the focus is more on the integration of different streaming-based kernels, which is better aligned with the work of [40], in which Redsharc is introduced. The authors elaborate further on the idea of creating an API to abstract communication between heterogoneous computational units, but besides that, they also present custom on-chip networks to support this API. By adapting a streaming model instead of an MPI- or threading-based system, they allow for a direct connection between the computational units as the streams only contain data. In this work, a lightweight mechanism similar to their stream switch network (SSN), which allows to route streams between different hardware/software kernels, is adopted.

Another promising development environment is the recently released Xilinx Vitis Unified Software Platform, created to build and seamlessly deploy accelerated applications on Xilinx platforms. Vitis supports application programming using C, C++ and OpenCL, an offers a set of performance-optimised, open-source libraries that ease the application development. The communication between the host application and the accelerators is facilitated by the Xilinx runtime (XRT), which provides APIs for accelerator life-cycle management, accelerator execution management, memory allocation, and data communication between the host application and the accelerators [41]. While it seems to be possible to use this new environment within the context of this thesis, it would still essentialy require a very similar design flow, in which numerical Python procedures need to be converted into an algorithmic C or C++ description that is suitable for HLS, and in which the accelerator management still needs to be controlled from within Python. Moreover, this platform is still considered to be at a premature stage at the time of writing, and the lack of unambiguous documentation and scientific support, especially

concerning the integration with Python, has caused this avenue not to be explored any further.

3

Hardware design & implementation

As part of this work, a number of hardware designs have been created to facilitate the acceleration of commonly used NumPy functions. First, an overview is given of the architecture of a generic block diagram that allows multiple accelerator cores to be combined into a single hardware design. Next, the implementation of the individually realised accelerators is addressed, in which the high-level synthesis flow is considered, taking into account interfacing, performance optimisation, and resource optimisation aspects.

3.1 Generic block design

With the aim of combining as many NumPy procedures into a single hardware design as possible, a layout was devised in which streams of input data can be processed by the FPGA (possibly by multiple accelerator cores) and streamed back to the DRAM. Streaming interfaces are used wherever possible because random access to the DDR memory is a costly operation that can take many more cycles than sequential accesses [42].

Moving data between PS and PL

Before revealing the block design, it should be noted that there are different methods that can be used to move data between the processing system, which connects to the DRAM, and the programmable logic on the Zynq-7000 SoC. First of all, it is possible to use the CPU directly to move any data between the PS and the PL, making use of the general purpose (GP) master ports of the Zynq device. This results in a simple software implementation and a low resource usage, at the cost of a low throughput. Other methods involve the use of direct memory access (DMA) components to move data without much intervention of the CPU. More concretely, it is possible to create DMA components on the programmable logic, which can then make use of the general purpose (GP), high-performance (HP) or accelerator coherency port (ACP) slave interfaces of the processing system, to fetch data from the main memory. Alternatively, the Zynq architecture also contains a flexible DMA engine and controller (DMAC) with a low resource footprint, that allows the CPU to instantiate data transfers by writing dedicated DMA microinstructions to a part of the system's memory. To achieve the highest possible aggregate bandwidth, the choice was made to adhere to a method that makes use of all four available AXI high-performance (HP) ports as an interface between the PS and the PL, and in which the DMA components reside on the FPGA fabric.

Block diagram

The final block design is shown in Figure 3.1. As can be seen from the diagram, two 64-bit HP ports are used to transfer data from the PS to the PL, while the remaining two are used in the opposite direction. Each HP port is associated with a dedicated DMA channel (each DMA component contains two independent channels), which provides high-bandwidth direct memory access between the DRAM and its streaming interface. The AXI-stream switch routes all incoming streams to the appropriate accelerator core(s) and (possibly) streams the results back to the DMAs. The AXI-stream switch makes it possible to integrate multiple accelerator cores into the design, which each can have (up to) two input streams and (up to) two output streams. All PL components can be controlled by the CPU through one of the 32-bit AXI general purpose (GP) master interfaces, typically making use of the AXI4-lite protocol.

AXI DMA

The AXI DMA components provide high-bandwidth direct memory access between their AXI memory mapped and AXI-stream interfaces. As illustrated in Figure 3.2, each DMA has two data channels that operate independently. The memory-mapped to stream (MM2S) channel is used to forward data from the DRAM to an AXI stream, using its AXI read master

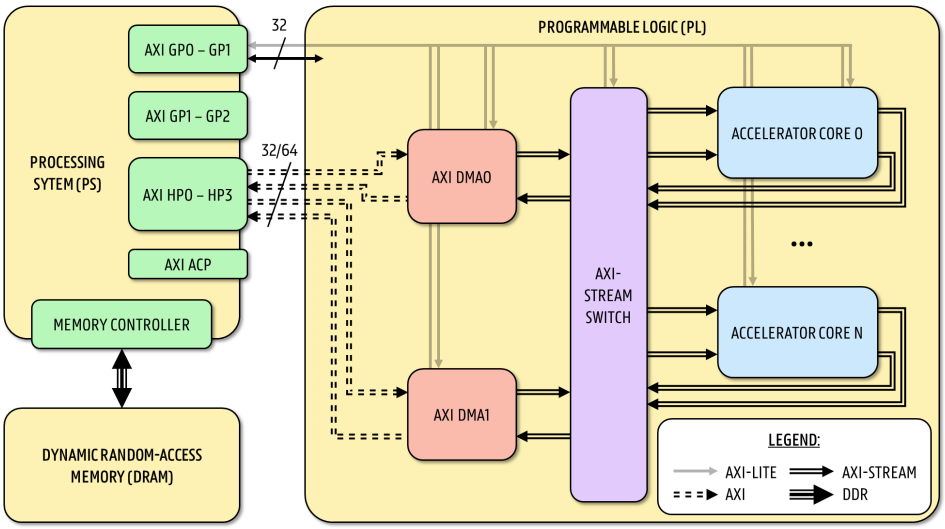

Figure 3.1: *Proposed generic block design. Data is streamed from the DRAM to the appropriate accelerator cores using all four available AXI high-performance (HP) ports, each connected to a dedicated AXI DMA channel. The AXI-stream switch routes all streams to the appropriate IP core(s) and forwards them back to the DMAs if required. All PL components are controlled using AXI-lite over the general purpose (GP) master interface of the PS.*

(`M_AXI_MM2S`) and AXI MM2S master (`M_AXIS_MM2S`) interfaces. On the other hand, the stream to memory-mapped (S2MM) channel is used to forward data from an AXI stream to the DRAM, using its AXI S2MM slave (`S_AXIS_S2MM`) and AXI write master (`M_AXI_S2MM`) interfaces. The DMA's memory mapped interface supports data widths from 32 up to 1024 bits, while the streaming interface allows widths from 8 up to 1024 bits[1].

The AXI DMA components have three major modes of operation: a direct register mode, a scatter/gather (SG) mode and a micro mode[2]. In the direct register mode, transfers are initiated by setting a source/destination address (for MM2S/S2MM transactions respectively), along with the number of bytes to transfer. The registers are an integral part of the DMA and can be controlled using the AXI-lite interface (`S_AXI_LITE`) of the component. The scatter/gather (SG) mode allows to describe multiple transfers at once, while the micro mode creates a low footprint, low performance IP that can handle the transfer of small packets. In the proposed block diagram, the choice was made to use the direct register mode in order to keep the design simple and performing, while also keeping resource utilisation under control.

[1] In this work, they are configured to 64 bits and 32 bits, respectively.
[2] See AXI DMA LogiCORE IP Product Guide.

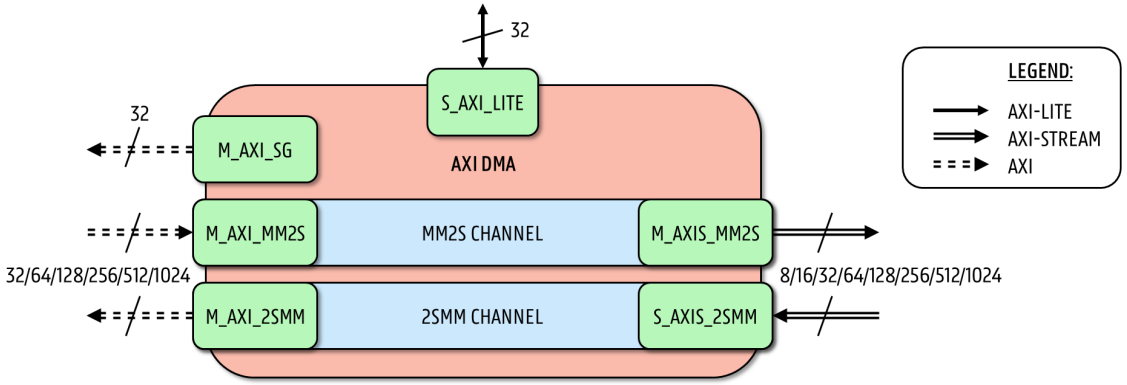

Figure 3.2: *Schematic overview of the AXI DMA component. There are two data channels (MM2S and S2MM) that operate independently. The component is controlled using an AXI-lite interface. When using the scatter/gather mode, an additional AXI master port is used.*

AXI-stream switch

The used AXI4-stream switch provides configurable routing between up to 16 masters and 16 slaves. The use of control register routing enables an AXI4-lite interface through which the the routing table can be configured, using a dedicated register for each of the master interfaces, in which the corresponding slave designator can be written. Once the registers have been programmed, a commit register transfers the programmed values from the register interface into the switch. The component requires that there is precisely only one path between every master and slave. When the component is instantiated in the block design, it is possible to indicate that certain routes between master and slave devices will not be used, which allows resources to be conserved when they are not needed, and at the same time prevents invalid routes[3].

3.2 Accelerator for element-wise operations

As part of this thesis, an IP core was realised using high-level synthesis that can process streams of data by applying an operation to each element. The core is able to process one input stream (in case of a unary operation) or two input streams (in case of a binary operation), which is facilitated by the reusable block design. The component is named `ufunc_call_xx`, because the operation corresponds to a particular NumPy universal function (ufunc), on which the `__call__()` method is invoked. The final two characters (`xx`) designate the data type that is used for the operations, such as `f4` for 4-byte floating point numbers or `i4` for 4-byte integer numbers. The core's functionality is described in Algorithm 1.

[3]See AXI4-Switch LogiCORE IP Product Guide.

Algorithm 1 Processing of stream(s) using an element-wise operation.

1: **function** ProcessStreams($a[, b], c[, d], length$)
2: $i \leftarrow 0$
3: **while** $i < length$ **do**
4: $c_i[, d_i] \leftarrow f(a_i[, b_i])$ ▷ f is a unary or binary operation
5: $i \leftarrow i + 1$
6: **end while**
7: **end function**

3.2.1 Interfacing

In Vitis HLS, the interfaces and ports (grouping multiple signals) to communicate with external components are synthesised based on the arguments of the selected C++ top-level function. The types of interfaces that are created depend on the data type, the direction of the parameters and any specified `INTERFACE` pragmas or directives [43]. The signature of the top-level function `ufunc_call_xx` can be found in Listing 3.1 and shows the two possible input streams (`in1_s` and `in2_s`) and output streams (`out1_s` and `out2_s`). To make sure these are synthesised as AXI-stream interfaces, the pragma `HLS interface axis port=xx` is used. Likewise, the `args` parameter, which is used to send additional information to the core, is accompanied by the `HLS interface s_axilite port=xx` pragma to indicate it is an AXI-lite interface in which the parameters can be accessed by reading from or writing to 32-bit registers.

The `packed_t` type is used to pack as much data as possible into a single register (currently only 8 bits are used by the `element_op` field, which will be explained later). Suppose there was the need to have an additional parameter that indicates whether implementation "A" or implementation "B" were to be used for a particular operation. In that case, a simple 1-bit field could be added to the `packed_t` struct instead of creating a new AXI-lite parameter. The latter would result in the synthesis of two 32-bit registers, which not only increases the resource usage, but also requires two AXI-lite transactions to set the parameters instead of just one [4].

[4] It is possible that bit fields are padded and aligned at the byte-level rather than at the bit-level, depending on the specific version of Vitis HLS that is used. For more information, refer the documentation of the `HLS aggregate` pragma [44]. Anyhow, the concept of packing data using a struct remains the same.

```
1  typedef struct {
2    unsigned char element_op;   // Element-wise operation to use.
3  } packed_t;
4
5  void ufunc_call_xx(
6    stream_t &in1_s, stream_t &in2_s, stream_t &out1_s, stream_t &out2_s,
7    packed_t args
8    ) {
9    #pragma HLS interface axis port=in1_s
10   #pragma HLS interface axis port=in2_s
11   #pragma HLS interface axis port=out1_s
12   #pragma HLS interface axis port=out2_s
13   #pragma HLS interface s_axilite port=args
```

Listing 3.1: *Illustration of the use of pragmas to guide the interface synthesis of the* `ufunc_call_xx` *top-level function.*

3.2.2 Optimising for performance

It is possible to translate Algorithm 1 directly to C++ code that is suitable for high-level synthesis, as is shown in Listing 3.2. The `while`-loop has been transformed into a `do/while`-loop (labeled `main`) that runs until the last element of an input stream is encountered. In the AXI4-stream protocol, this is indicated by the assertion of the `T_LAST` signal that is part of the stream's side channels. The read and write functions convert individual stream elements (represented by the `stream_t` type) to their numerical representation (such as a floating-point number) and vice versa. The multiplication can easily be replaced by another operation. Most unary and binary operations that are part of the C++ math library (`<cmath>`) are already optimised for high-level synthesis and can be found in the Vitis HLS math library (`<hls_math.h>`) [45, Ch. 24]. A floating point exponentiation would for example be realised by replacing line 8 with `out1 = hls::logf(in1);`.

The use of high-level synthesis facilitates the pipelining of loops to increase the possible throughput. Because many hardware operations take N clock cycles to complete, a purely sequential loop execution would result in the production of a new result every N clock cycles: the loop is said to have an initiation interval (II) of N. Pipelining a loop allows the operations of the loop to be implemented in a concurrent manner rather than sequentially, thereby reducing the initiation interval. This is achieved by splitting up the operation into multiple stages and by executing multiple stages in parallel. This is illustrated in Figure 3.3 with an operation (`OP`) that can be split up into five stages (`OP1-OP5`). A purely sequential execution (A) produces one result every 5 cycles, while the pipelined version (B) produces one result every cycle, after an initial iteration latency (IL) of 5 cycles. It is clear that the initiation interval is an important metric that is directly related to the throughput of the accelerator. When optimising loops in HLS

```
(...) // Initialisation etc.

main: do {
    #pragma HLS pipeline II=1

    in1 = read_stream_1(...);   // Read the first input stream.
    in2 = read_stream_2(...);   // Read the second input stream.
    out1 = in1 * in2;           // Do the operation.
    write_stream_1(out1);       // Write to the output stream.

  } while(t_last_not_asserted());
```

Listing 3.2: *High-level synthesis code illustrating the element-wise multiplication of two streams of numbers.*

components, the goal is therefore to obtain an initiation interval of one in order to achieve an optimal throughput. The HLS reports provide feedback and also indicate possible bottlenecks (e.g. data dependencies) that prevent an II=1 from being achieved.

As can be seen in Listing 3.2, loop pipeling is applied to the main `do/while`-loop by specifying the pragma `HLS pipeline II=1`. Since there are no data dependencies between the loop iterations, there is no problem to obtain an II=1.

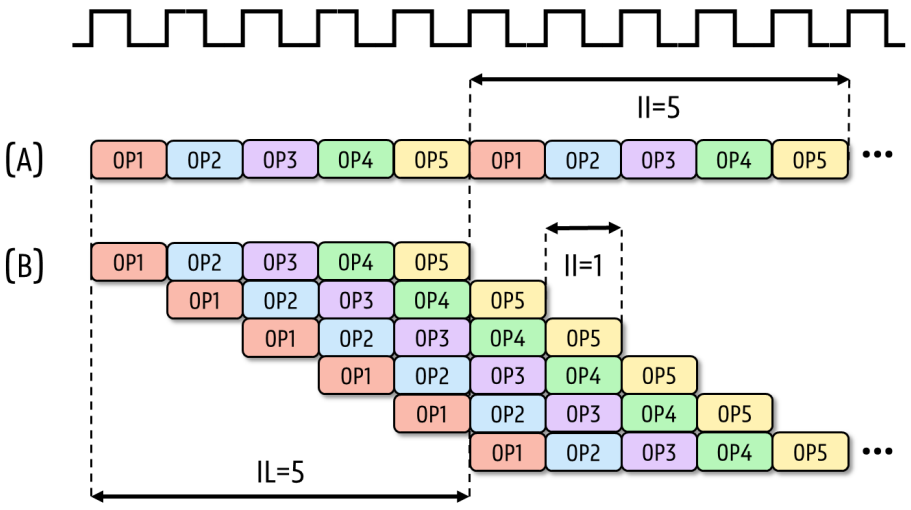

Figure 3.3: *Visualisation of loop pipelining. A purely sequential execution (A) produces one result every 5 cycles, while the pipelined version (B) produces one result every cycle, after an iteration latency (IL) of 5 cycles.*

3.2.3 Optimising for resource usage

If multiple different element-wise operations are required or desired, it would be possible to spawn multiple cores that each implement a particular operation, and use the AXI-switch to select the appropriate one at run-time. However, the choice was made to combine as many operations as possible into a single core and select the desired operation using the `args.element_op` parameter, which is synthesised as an AXI-lite interface. This does not only reduce the amount of HLS code duplication, but it is also beneficial for the resource usage: the hardware used to read/write the streams and registers is now shared and it also allows the HLS compiler to do a global optimisation of the resources. To illustrate this, the core was synthesised multiple times including different operations. The resulting usage is shown in Table 3.1. As can be seen from the last row, the resource usage of the core with all four operations combined is much lower than the sum of the usages of the individually realised cores.

The operator selection can be realised by adding a `switch/case` statement to the main loop specified in Listing 3.2, which branches based on the value of the `args.element_op` parameter. It is also possible to implement each operation inside a separate C++ function in order to keep the code well-structured. However, in that case the pragma `HLS inline` should be used inside each function to indicate that the hardware resources may be shared. This because after inlining, the target function is resolved into the calling function and no longer appears as a separate hierarchical level inside the RTL representation. This design allows functionality to be added or removed with minimal effort. One drawback from this method is that the iteration latency (IL) of the main loop increases proportionally with the latency of the longest operation inside the `switch/case` structure.

Finally, it should be noted that – since all element-wise operations are independent of each other – there is no need to store any intermediate results inside the IP core's memory such as

Table 3.1: *Comparison of the resource usage of the `ufunc_call_xx` core when different operations are included. As can be seen from the last two rows, it is beneficial to combine multiple operations within a core instead of spawning multiple cores that each implement only a single operation.*

Operation(s)	**BRAM**	**DSP**	**FF**	**LUT**
Addition	0	2	428	614
Subtraction	0	2	428	614
Multiplication	0	3	365	545
Division	0	0	996	1186
$\sum$	**0**	**7**	**2217**	**2959**
Combined	0	5	1541	1982

the BRAM. This means that the resource usage of the core is mostly determined by the resource usage of the included operations.

3.3 Accelerator for reduction-type operations

Another IP core that was realised using high-level synthesis can apply reduction-type of operations to incoming data streams. In accordance with the previous section, this component is named named `ufunc_reduce_all_xx`, because the operation corresponds to a particular NumPy universal function (ufunc), on which the `__reduce__()` method is invoked. The `all` indicates that the input stream, which can represent a multi-dimensional structure, will be reduced along all axes at once to create a single value. The core's functionality is described in Algorithm 2.

In its most basic form, the core converts a single input stream into a single value by repeatedly applying a certain binary operation g. For example, when $g(x, y) = x + y$, the core will calculate the sum of all elements in the input stream. Additionally, it is also possible to specify a second, unary or binary, operation f, which will be applied to each (pair of) incoming element(s) before applying the reduction operation. For example, when $f(x) = x^2$ (unary operation) and $g(x, y) = x + y$, the core will calculate the sum of the squares of all elements in the input stream. Alternatively, when $f(x, y) = x \cdot y$ (binary operation) and $g(x, y) = x + y$, the core will calculate the dot product between the elements of the two input streams. Finally, it is also possible to add a third, unary, operation h that will be applied to the result before returning it. For example, when $f(x) = e^x$ (unary operation), $g(x, y) = x + y$ and $h(x) = log(x)$, the core will calculate the expression $log(\sum e^{a_i})$ with a_i representing an element from the input stream a.

Algorithm 2 Processing of stream(s) using a reduction-type of operation.

1: **function** ProcessStreams($a[, b], length$)
2: $res \leftarrow f(\mathrm{a}_0[, \mathrm{b}_0])$ ▷ f is an optional unary or binary operation
3: $i \leftarrow 1$
4: **while** $i < length$ **do**
5: $res \leftarrow g(res, f(\mathrm{a}_i[, \mathrm{b}_i]))$ ▷ g is a binary operation
6: $i \leftarrow i + 1$
7: **end while**
8: **return** $h(res)$ ▷ h is an optional unary operation
9: **end function**

3.3.1 Interfacing

The signature of the top-level function `ufunc_reduce_all_xx` can be found in Listing 3.3 and shows the two possible input streams (`in1_s` and `in2_s`) and the `args` parameter that is used to send additional data to the core. Again, the pragmas `HLS interface axis port=xx` and `HLS interface s_axilite port=xx` are used to indicate the types of interfaces. Contrary to the previous IP core, this top-level function returns a single value, instead an output stream. The associated pragma indicates that the return value needs to be synthesised as an AXI-lite interface, which means that it will be possible to fetch the result by reading from a register in the core, which will be mapped to a fixed location in the DRAM.

Similar to the previous IP core, it is possible to select the required operations using the `args` parameter. The functions f, g and h can be selected using `args.element_op`, `args.reduce_op` and `args.final_op` respectively. Because structs used as arguments to the top-level function are aggregated (packed) by default, these three functions can be selected by using just a single 32-bit register, as outlined in Section 3.2.1.

```
1  typedef struct {
2    unsigned char reduce_op;    // Reduction operation to use.
3    unsigned char element_op;   // Element-wise operation to use.
4    unsigned char final_op;    // Element-wise operation to do on the result.
5  } packed_t;
6
7  return_t ufunc_reduce_all_xx(
8      stream_t &in1_s, stream_t &in2_s,
9      packed_t args
10   )
11 {
12     #pragma HLS interface axis port=in1_s
13     #pragma HLS interface axis port=in2_s
14     #pragma HLS interface s_axilite port=args
15     #pragma HLS interface s_axilite port=return
```

Listing 3.3: *Illustration of the use of pragmas to guide the interface synthesis of the `ufunc_reduce_all_xx` top-level function.*

3.3.2 Optimising for performance

It is possible to translate Algorithm 2 directly to C++ code that is suitable for high-level synthesis using a `do/while`-loop that runs until the last element of the stream is encountered, and in which a `switch/case`-statement is used to branch depending on the selected operation(s).

However, unlike the previous core, there are data dependencies between the iterations of the

main loop because of the expression $res \leftarrow g(res, f(\mathrm{a}_i[, \mathrm{b}_i]))$. Therefore, it is not possible to achieve an initiation interval of one, unless the operation takes less than one cyle. In order to achieve a higher overall throughput, the core instead calculates a series of p partial results in parallel (in fact, fully pipelined with an initiation interval of one). After this initial stage, the array of partial results is further processed using a tree-like access pattern to produce the final return value.

The mechanism used to calculate the partial results is illustrated in Figure 3.4, in which it is assumed that the pipeline has an iteration latency (IL) of four cycles[5]. The first partial result, `res_p[0]`, is calculated based on every fourth value in the input stream. Likewise, partial result `res_p[i]` is calculated based on every n-th value in the input stream for which $(n-1) \equiv i \mod p$. Setting the number of partial results equal to the iteration latency of the pipeline (four in this case) allows to fully mask the data dependencies, which are shown as black curved arrows. This makes it possible to produce one partial result every clock cycle, hence achieving an II=1 during the calculation of the partial results.

The structure used to calculate the final result from the partial results is illustrated in Figure 3.5. In a first stage, the reduction operation is applied to `res_p[0]`/`res_p[1]` and `res_p[2]`/`res_p[3]`. After that, these results are further combined to produce the final output value `res`. More generally, there will be an additional $p-1$ reduction operations spread over $log_2(p)$ stages with p equal to the number of partial results (assumed to be a power of two). In total, this stage adds an additional $(p-1)(log_2(p)+1)$ clock cycles to the latency of the core.

It should be noted that this method of applying reduction-type of operations comes with its limitations. First of all, the reduction operation must be commutative, because the results are rearranged during the calculation. It is, for example, not possible to directly calculate $a_0 - a_1 - ... - a_n$ because the subtraction is not commutative. However, in this case this can be circumvented by calculating $a_1 + ... + a_n$ using the core and by calculating the final result $a_0 - (a_1 + ... + a_n)$ on the CPU. Furthermore, it is assumed that all input streams are longer than the number of partial results p, because the reduction of the partial results doesn't take into account the length of the stream. Alternatively, the partial results could be initialised with the value of the neutral element of the operation that is selected.

The essentials of the resulting high-level synthesis code are shown in Listing 3.4. The calculation of the partial results is realised in the loop labeled `main`, which makes use of the pragmas `HLS pipeline II=1` and `HLS dependence variable=res_p inter false`. The latter instructs the compiler to remove any dependencies between elements of `res_p` that are accessed in the different loop iterations (such as `res_p[0]` in every fourth cycle). Although this is a pragma that needs to be handled with great care, in this case, the automatic dependence anal-

[5] In reality, the iteration latency is provided to the developer in the high-level synthesis reports

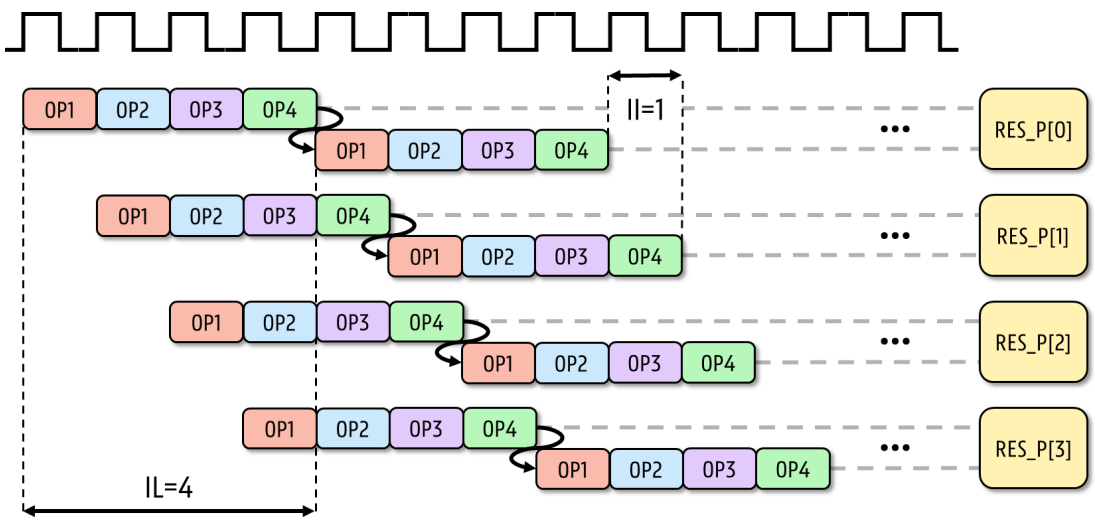

Figure 3.4: *Illustration of the pipelining mechanism that is used to calculate the partial results in the* `ufunc_reduce_all_xx` *core. The data dependencies shown as black curved arrows are masked by the calculation of the remaining partial results. The grey lines indicate the memory locations where the results are stored.*

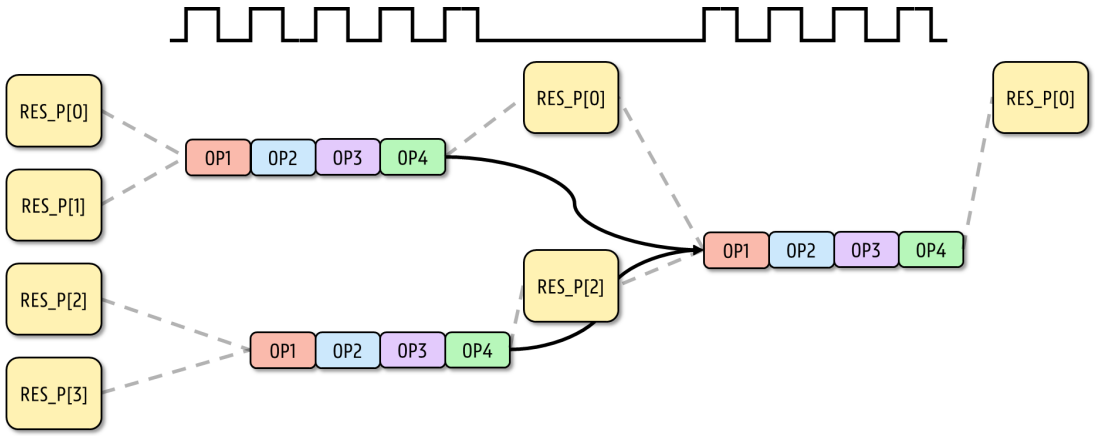

Figure 3.5: *Illustration of the reduction of partial results into a single value in the* `ufunc_reduce_all_xx` *core, using a tree-like structure. Data dependencies are shown as black curved arrows and read/write accesses are indicated using grey lines.*

ysis is too conservative and fails to filter out these false dependencies when it is left out. The `do_operation()` function contains the `switch/case`-statement, applies the selected operation to its input argument(s) and stores the result back into its second argument. Line 14 is used to initialise the partial results, because the first `PIPE_DEPTH`[6] input values cannot be reduced yet. The variable p is used to switch between the different partial results in every clock cycle.

The reduction of the partial results is realised in the nested loop labeled `fini`, which iterates over the $log_2(p)$ stages in the outer loop, and over the indices of the remaining partial results in the inner loop. The outer loop is unrolled using the pragma `HLS unroll`, which transforms loops by creating multiple copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel, at the cost of an increased resource utilisation. Applied to the reduction of the partial results, it implies that the $log_2(p)$ stages are implemented separately from each other, resulting in a slightly lower latency[7].

The other involved pragma, `HLS array_partition variable=res_p complete`, partitions the array `res_p` into `PIPE_DEPTH` individual memories or registers. By default, arrays are synthesised into a single large dual-port memory, which means that within a clock cycle (i) two elements can be read, (ii) two elements can be written to, or (iii) one element can be read and another element can be written to the memory. When p is large enough, it is possible that two values of `res_p` are being read in a given cycle while another instance writes back a value to the same memory. To prevent these types of collisions, the array is fully partitioned into individual registers.

[6] `PIPE_DEPTH` is macro definition that contains the depth of the pipeline, as returned by the HLS reports.

[7] Note that they are not truly parallel, as can be seen in Figure 3.5, which will be discussed in Section 3.3.3.

```
unsigned int p = 0, n = 0;

#pragma HLS allocation function instances=do_operation limit=1
#pragma HLS dependence variable=res_p inter false

main: do {
  #pragma HLS pipeline II=1

  (...) // Read the input(s).

  do_operation(&args.element_op, &in1, &in2);     // Apply element-wise operation.
  do_operation(&args.reduce_op, &res_p[p], &in1); // Apply reduction operation.

  if(n++ < PIPE_DEPTH) res_p[p] = in1;           // Simply store the first PIPE_DEPTH elements.

  p = (p == PIPE_DEPTH - 1 ? 0 : p + 1);          // Increment modulo PIPE_DEPTH.
} while(t_last_not_asserted());

#pragma HLS array_partition variable=res_p complete

fini: for(int depth = 0; depth < LOG2_PIPE_DEPTH; depth++) {
  #pragma HLS unroll
  for(int i = 0; i < PIPE_DEPTH; i += (2 << depth)) {
    do_operation(&args.reduce_op, &res_p[i], &res_p[i + (1 << depth)]);  // Apply reduction operation.
  }
}

do_operation(&args.final_op, &res_p[0]);  // Apply final operation.
return res_p[0];
```

Listing 3.4: *High-level synthesis code illustrating the complete reduction of two streams of numbers.*

3.3.3 Optimising for resource usage

As with the previous IP core, the choice was made to combine as many (required) operations as possible and select the desired operations using the `args` parameter, which enables resource sharing between different operations. An interesting optimisation is done through the use of the `HLS allocation function instances=do_operation limit=1` pragma, which limits the number of RTL instances of the function `do_operation()` to one. This reduces the hardware resources used by the function, but negatively impacts performance by forcing the resources to be shared (i.e. the `do_operation()` cannot be made truly parallel). However, the impact is very limited in this case:

- The `main` loop only receives one (or two) new element(s) in every clock cycle, which means that a truly parallel implementation will have no advantage over it.

- The `fini` loop will benefit most from a truly parallel implementation, because the result will be available after $p \cdot log_2(p)$ cycles, rather than after $(p-1)(log_2(p)+1)$ cycles. For a realistic value of $p = 16$, the difference is 11 clock cycles. The benefit of these cycles is neglegible, especially taking into account that the resource usage would have to be increased by a factor of $p/2 = 8$, when implemented in parallel.

The `HLS unroll` pragma used in the `fini` loop is also affected by the allocation pragma. In a regular scenario, loop unrolling tends to increase the resource usage with a factor that is proportional to the unroll factor, $log_2(p)$ in this case. However, the number of instances of the `do_operation()` function is limited to one, which only causes the residual logic to be duplicated. Another benefit of the allocation pragma is that the functionality involving the `args.element_op` and the `args.final_op` can be added to the core without much resource overhead. This effect is further emphasised by the fact that the element-wise and reduction operation are often the same, or at least very similar operations.

3.4 Accelerator for filter-type operations

A third type of IP core was realised to showcase the usage of filter operations, in which a fixed window slides over a two-dimensional input array. More concretely, an implementation of an average filter was made, in which each output element is produced by taking the average of the corresponding input element and its four direct neighbours. The component is named `filter_avg_xx` and its functionality is described in Algorithm 3. It should be noted that the elements on the borders are left untouched; alternatives include calculating the average over only the available values, or wrapping the boundary values.

Algorithm 3 Average filtering algorithm.

1: **function** AVERAGEFILTER($a, b, height, width$)
2: **for** $i \leftarrow 0$ to $height$ **do**
3: **for** $j \leftarrow 0$ to $width$ **do**
4: **if** $(i = 0)$ or $(i = height - 1)$ or $(j = 0)$ or $(j = width - 1)$ **then**
5: $b_{i,j} = a_{i,j}$ ▷ Elements on the borders are left untouched.
6: **else**
7: $b_{i,j} = \frac{1}{5} \cdot (\mathrm{a}_{i-1,j} + \mathrm{a}_{i,j} + \mathrm{a}_{i+1,j} + \mathrm{a}_{i,j-1} + \mathrm{a}_{i,j+1})$
8: **end if**
9: **end for**
10: **end for**
11: **end function**

3.4.1 Interfacing

The interfacing with the component is very similar to what is explained in Section 3.3.1. The realised IP core accepts a single input stream and produces a single output stream, which is indicated by the use of the pragma `HLS interface axis port=xx`. Furthermore, the component requires the height and width of the two-dimensional input, which are passed to the core as AXI-lite parameters, by making use of the pragma `HLS interface s_axilite port=xx`.

3.4.2 Optimising for performance

It is not possible to translate Algorithm 3 directly to C/C++ code that is suitable for high-level synthesis, because the algorithm assumes the neigbouring elements are available at all times, while the data is streamed in one element at a time. The solution is to always store the `2W` most recently read elements in a local memory structure called a *line buffer*, with `W` representing the number of columns or "width" of the input array. The principle is illustrated in Figure 3.6: the elements in green are stored in the line buffer and the element in red (labeled `E`) corresponds to the last element that was streamed in. It is clear that, at that point in time, it is possible to calculate the output value at the position `C`. Furthermore, it is possible to immediately calculate the output value at position `D` when the next element has been read in. More generally, the output at position (i, j) is produced immediately when the input value at position $(i + 1, j)$ becomes available[8]. Overall, this method allows to produce a new result every clock cycle, hence achieving an initiation interval of one.

The HLS code to realise this functionality is shown in Listing 3.5. Initially, the line buffer is empty and needs to be filled by streaming in the first `2W + 1` values, in which `W` is used as a shorthand for the `args.width` parameter. Furthermore, the first `W + 1` values can be directly streamed out because they are on the border of the two-dimensional array. The filter operation starts by iterating over the rows and columns of the input array using two nested `for`-loops. Both are fully pipelined to achieve the lowest possible initiation interval, which is indicated by the use of the pragma `HLS pipeline II=1`[9].

[8] To be precise: the new element can enter the pipeline stage, but it will still take some time until the actual result is streamed out, depending on the depth of the pipeline.

[9] Note that it is not strictly required to explicitly use the pragma, because loops are automatically pipelined in Vitis HLS. It is used for illustration purposes.

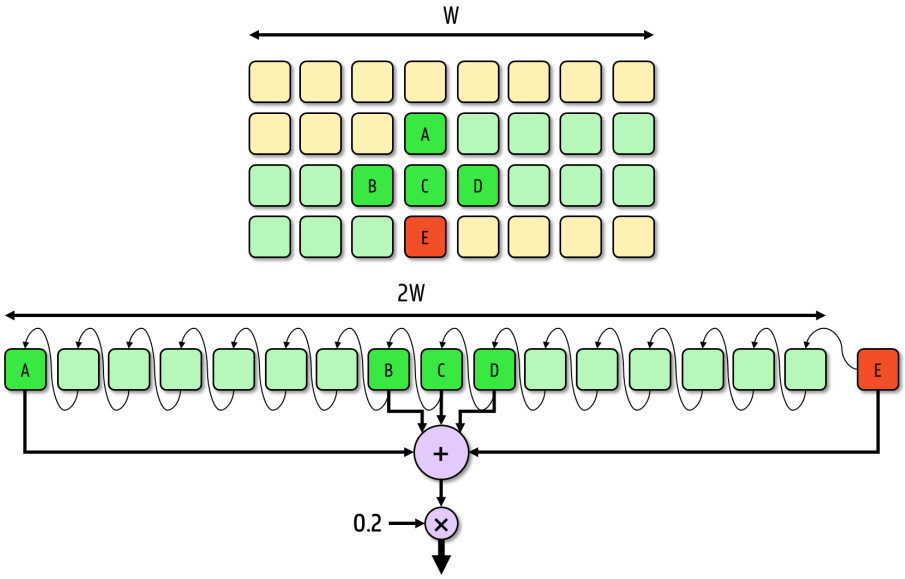

Figure 3.6: *Illustration of the use of a line buffer to facilitate the efficient filtering of a two-dimensional array that is streamed into the accelerator, in an element-by-element fashion. As shown in the lower part of the figure, the buffer can be implemented using shift registers.*

```
    (...) // Read in the first "2 * width" elements from the stream.
    (...) // Write out the first "width + 1" elements, because they are on the border.

  bool first = true;
  main_y: for(int y = 1; y < args.height - 1; y++) {
    main_x: for (int x = 0; x < args.width; x++) {
        #pragma HLS pipeline II=1

        if(first) {  // Make sure that first element of the second row is skipped.
          first = false; continue;
        }

        // Read in new value and output the new result. If (y, x) is currently on the
        // border, output the border value stored in lb_3 instead of the average.
        new_value = read_stream(in_s);
        write_stream(out_s, on_border_x(x, args.width) ?
          lb_3.read() :
          0.2f * (lb_1.read(args.width - 2) + lb_2.read(0) + lb_3.read(0) + lb_4.read(0) + new_value)
        );

        // Shift the new element into the line buffer.
        SHIFT_IN(new_value, args.width);
    }
  }

  (...) // Stream out the last remaining row, which is left untouched.
```

Listing 3.5: *High-level synthesis code illustrating the use of a line buffer to facilitate the filtering of an incoming data stream that represents a two-dimensional array.*

In each iteration, three operations take place:

- A new value of the input stream is read, which corresponds to the position $(y+1, x)$.
- A new value is written to the output stream which corresponds to the position (y, x). The value is obtained by taking the average of the relevant items in the line buffer (indicated by `A`, `B`, `C` and `D` in Figure 3.6) and the newly read value (`E`). If (y, x) is currently on the border, only the value of `C` is written to the output stream.
- The newly read value is shifted into the line buffer and the oldest value from it is discarded.

The line buffer can be implemented as a one-dimensional shift register of length `2W`. Implementing the shift register using a single memory inherently leads to scheduling problems, because at every iteration, four values need to be read, and a fifth value needs to be written to the same memory. A possible solution is to fully partition the line buffer using the pragma `HLS array_partition variable=lb complete`, which results in an RTL implementation with multiple smaller memories. This allows these operations to take place in parallel, and prevents that the initiation interval increases due to the limited number of I/O ports of the memory elements.

After completing the nested for loops (`main_y` and `main_x`), the IP core still needs to stream out the last remaining row, which is left untouched because it lies on the border. It should also be noted that during the very first iteration of the main loops, no operation takes place because the corresponding value was already streamed out during the initialisation phase. While it is possible to split off this iteration from the nested loops (often referred to as *loop peeling*), it does not provide any advantage in this particular case.

3.4.3 Optimising for resource usage

The resource usage of the component is rather minimal, as it only requires four adders and a single multiplier, besides the interfacing logic and the logic that is required to set up the pipelines. One optimisation that was carried out is related to the partitioning of the line buffer. The buffer needs to be partitioned in order to achieve an initiation interval of one, however it does not need to be *completely* partitioned into single memory elements. It is clear from Figure 3.6 that most elements only require a single read and a single write per cycle, which is possible because of the dual ported-nature of the memories. The only elements that need to be accessed more than twice per cycle, are those at positions `B`, `C` and `D`.

In Vitis HLS, it is not possible to partition an array at arbitrary locations. However, it is possible to manually partition the array by putting five shift registers in series[10]:

- The first shift register has a size of `W - 1` and contains all values from `A` to `B` (not included).
- The next three (shift) registers can only hold a single element and contain the values `B`, `C` and `D`, respectively.
- The fifth shift register has a size of `W - 2` and contains the remaining (most recently read) values.

Because the width of the input array is not known in advance, a maximum capacity needs to be assigned to the shift registers so that they can be synthesised. This capacity can be chosen based on the targeted applications, but is also limited by the number of available LUTs that can be used as shift registers.

[10] In Listing 3.5, these are represented by the variables `lb_1` to `lb_5`.

4

Processing system/programmable logic interfacing

In Chapter 3, the design and implementation of several hardware components that can be loaded onto the FPGA were discussed, along with the possibilities to integrate them into a single hardware overlay. Keeping in mind the goal of accelerating numerical libraries for Python, which runs on the CPU, this chapter will cover the possibilities to communicate between the programmable logic and the processing system. The interfacing is considered at three different levels, which each introduce additional overhead in terms of latency and throughput.

At the lowest level, it is possible to directly control the IP cores on an overlay by using a standalone program, in which instructions are directly executed on the CPU without intervention of an operating system (i.e. *bare metal*). At the level above it, an operating system such as Linux is used to manage all hardware and software resources, and to provide common services to programs that run on it. Finally, at an even higher level, the interfacing between a Python application running on Linux and the IP cores on an overlay, is examined. The bare metal, Linux and Python interfacing are discussed in Sections 4.1, 4.2 and 4.3 respectively.

4.1 Bare metal interfacing

Memory mapped I/O

As discussed in Chapter 3, hardware components residing in the programmable logic are connected to the processing system using the AXI general purpose (GP) and high-performance (HP) interfaces. Each slave device is assigned an address range in which the corresponding master device can access it, as is shown in Figure 4.1. In the example, the AXI-lite slave interfaces of the `IP` and `DMA` components are mapped to the address ranges `0x4000_0000-0x4001_0000` and `0x4040_0000-0x4041_0000` of the CPU, which acts as master, respectively. The high-performance interface `HP0` also acts as a slave (which can fetch data from the main memory for the `DMA` master) and is assigned to the same address range as the physical DRAM, which has a size of 512MB on the PYNQ-Z2 board. The CPU can access individual registers within an IP core by adding a register-specific offset to the base address. In the example, the `args` register can be accessed by the CPU using the physical address `0x4040_0010`. For Xilinx devices, the address mapping is done automatically when creating a Vivado block design. An XML-representation of the mappings and register offsets can be found inside the hardware handoff file (`.hwh` file), which is part of an exported Vivado hardware design (`.xsa` file).

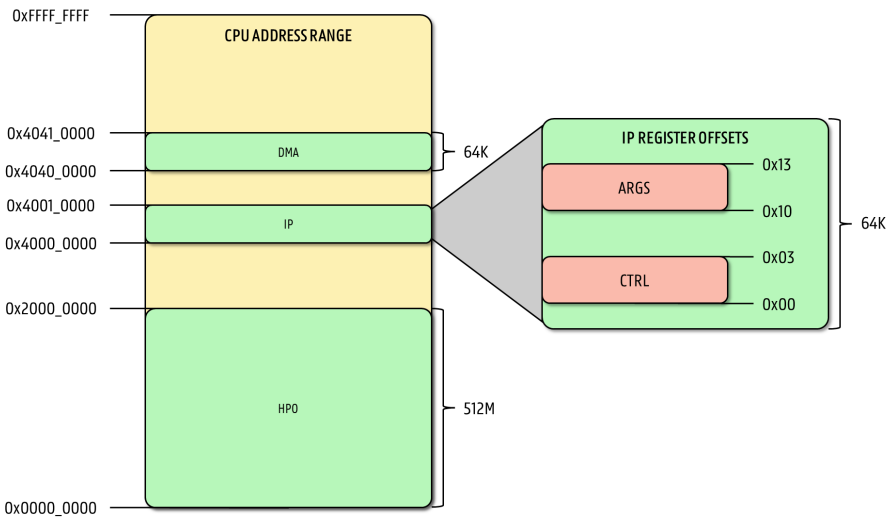

Figure 4.1: *Illustration of the mapping of slave memories into the address space of a master device. In this example, the CPU acts as a master and the `DMA0`, `IP0` and `HP0` interfaces act as slaves. The CPU can address individual IP registers by reading from or writing to the base address plus some register-specific offset. In this case, the CPU can use the physical address `0x4040_0010` to interface with the `args` register inside `IP`.*

A C or C++ standalone (bare metal) application can be created by using the Vitis software development platform, which creates all components required to run a standalone program, including the generation of the first-stage boot loader (FSBL) and the board support package (BSP). The FSBL configures the FPGA with the hardware bitstream, loads the standalone image from the non-volatile memory to the main memory and starts executing it [46]. The BSP on the other hand, is a collection of libraries and drivers that forms the lowest layer of the application software stack, against which the standalone application is linked. For example, it includes standard C libraries (`libc.a`, `libm.a`) and standalone device drivers for the peripherals in the design[1], including a header file `"xparameters.h"` that contains all the mappings and register offsets as specified in the `.hwh` file [47].

Interfacing with a single IP core

The interfacing between a standalone C/C++ application and an IP core is illustrated in g 4.1, in which the goal is to set the value of the `args` register of the core named `"my_ip"` to `0xDEADBEEF`. The first method directly writes to the register by using the base address of the IP core and the offset of the `args` register, as specified in the `"xparameters.h"` file. The `volatile` keyword is used to indicate that the value may change between different accesses (even if it does not appear to be modified) and prevents a compiler from optimising away subsequent reads or writes. The second method uses the (auto-generated) standalone drivers to achieve the same result. The drivers use structs to represent IP instances and their configuration data, and make use of standardised function calls to read, modify and write to the underlying registers of the associated IP core.

Interfacing with the generic block design

The methodology to use the generic block design that is presented in Section 3.1 in a standalone application is outlined below:

1. Initialise the hardware platform (this includes setting up the processing system, the power supply unit, and loading the appropriate bitstream to the PL).
2. Initialise the IP core(s), AXI-stream switch and DMA components using the structs that are used in the standalone drivers.
3. Allocate the required input and output buffers on the stack or on the heap (e.g. using the `static` keyword). Populate the input buffer(s) with test data.

[1]Standalone device drivers are auto-generated for all IP cores that are synthesised using Vitis HLS. Standalone device drivers for other peripherals can be found here.

```c
////////////////////////////////////////////////////////////////////
// Method (1) : Write directly to memory mapped register address. //
////////////////////////////////////////////////////////////////////

#include "xparameters.h"  // Includes the address mapping and offsets.

// #define XPAR_XMY_IP_S_AXI_CONTROL_BASEADDR   0x40000000
// #define XMY_IP_CONTROL_ADDR_ARGS             0x10

*((volatile unsigned int*)(XPAR_XMY_IP_S_AXI_CONTROL_BASEADDR + XMY_IP_CONTROL_ADDR_ARGS)) = 0xDEADBEEF;

//////////////////////////////////////////////////////////
// Method (2) : Using auto-generated standalone drivers. //
//////////////////////////////////////////////////////////

#include "xmy_ip.h"

// Initialise structs representing the IP core and its configuration.
XMy_ip ip;
XMy_ip_Config *ip_cfg = XFilter_avg_f4_LookupConfig(0);
XMy_ip_CfgInitialize(&ip, ip_cfg);

// Set the "args" register of the IP core to 0xDEADBEEF.
XMy_ip_Set_args(&ip, 0xDEADBEEF);
```

Listing 4.1: *Interfacing with an IP core from a standalone C/C++ application. It is possible to directly write the value* `0xDEADBEEF` *to a memory-mapped register of the IP core (method 1), or to use the auto-generated standalone drivers (method 2).*

4. Use the standalone drivers to pass any parameters to the required IP core and to start it.

5. Use the standalone drivers to configure the AXI-stream switch: forward the `MM2S`-streams of `DMA0` and `DMA1` to the inputs of the appropriate core and forward the outputs back to the `S2MM`-interfaces of `DMA0` and `DMA1`. Any unused streams can be left out, e.g. when a core produces only a single output stream.

6. Start the DMA transactions from the PS to the PL by writing the source address(es) of the input buffer(s) and the length of the input stream(s) to the appropriate registers in the DMA component(s).

7. If applicable, start the DMA transactions from the PL to the PS by writing the destination address(es) of the output buffer(s) and the length of the output stream(s) to the appropriate registers in the DMA component(s).

8. Wait until the core has finished processing, either through software polling or by using hardware interrupts. If applicable, also wait for the completion of the DMA transactions from the PL to the PS.

9. Read the output buffer(s) from the memory at the provided destination address(es) or, in case no output streams are produced, fetch the result from the appropriate registers in the IP core, again using the standalone drivers.

One aspect to keep in mind is the use of CPU caching in combination with direct memory access. When the DMA components on the programmable logic are connected to the HP ports of the processing system, they are unaware of the contents of the L1 or L2 data caches of the ARM CPUs. Therefore it is important to flush the corresponding cache lines before starting a DMA transaction from the PS to the PL (i.e. before step 6) and to invalidate them when completing a transaction from the PL to the PS (i.e. before step 9).

Overall, the interfacing between the PS and the PL using a bare metal application is rather straightforward, because the use of memory mapped I/O allows for a direct control of the registers inside an IP core.

4.2 Linux interfacing

Kernel space and user space drivers

To be able to use the Python ecosystem on the processing system, an operating system such as Linux is required. This implies that the hardware resources are no longer under direct control of the user, but are managed by the kernel. In Linux, hardware devices typically communicate with user space applications using kernel space drivers, whose development process is considered to be be difficult because it requires writing kernel code. However, many devices (including the IP cores in the generic block design) do not need to take advantage of the resources that the kernel provides and only require access to the memory space of the device, for example. Two ways to achieve direct access to the hardware from user space are accessing the physical memory via `/dev/mem`, and using the user space I/O (UIO) framework [48]. Even though the latter has the advantage of providing support for interrupts, it is also more complicated because it requires rebuilding the Linux kernel (using PetaLinux) to enable the framework, requires changes to the device tree[2] for each core that is added and generally involves more interfacing code than the other method. Therefore, the choice was made to adhere to the first technique, which is detailed below.

Interfacing with a single IP core

In contrast to bare metal applications, it is not possible to directly access the registers of the IP cores in a Linux application. This is because the operating system makes use of virtual memory, in which memory addresses used by a process, called *virtual* addresses, are mapped to addresses of the main memory, called *physical* addresses. This translation is typically done by the memory management unit (MMU) of the CPU.

It is however possible to work around this issue by using the construct shown in Listing 4.2, in which the value `0xDEADBEEF` is again written to the `args` register, similar to Section 4.1. First, the `/dev/mem/` file, a character device file that is an image of the main memory, is opened [49]. This means that byte addresses in `/dev/mem` are to be interpreted as physical memory addresses. This file is then mapped into the virtual address space of the calling process using the `mmap()` system call [50]. The contents of the file mapping are initialised with `PAGE_SIZE` bytes, starting at the offset specified by the `ip_addr_aligned` variable, which needs to be aligned to a page boundary. Taking into account the possible alignment offset[3], the `args` register can then be

[2]The device tree is a data structure that describes the hardware components of the system, such that the Linux kernel can use and manage them (e.g. assign the correct driver to a device that is loaded at run-time).

[3]In practice it may not be necessary to work with page boundary aligning because the physical base addresses of the cores are typically aligned to 64kB, while the default page size in Linux is just 4kB.

accessed by reading from or writing to the address pointed to by the virtual base address, plus the register specific offset corresponding to the `args` parameter. Overall, this method allows for fast prototyping by directly accessing the physical registers using their virtual counterpart.

```c
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>

typedef unsigned int uint;

long PAGE_SIZE = sysconf(_SC_PAGESIZE); // Page size (4KB by default).
uint ip_addr = 0x40000000;              // Physical base address of the IP core.

uint ip_addr_aligned = (ip_addr & (~(PAGE_SIZE - 1))); // Align the address to a page boundary.
uint align_offset = ip_addr - ip_addr_aligned;         // And remember the offset.

// Create a mapping from the physical memory to the virtual address space of the calling process.
int fd = open("/dev/mem", O_RDWR);
volatile uint *ip_vaddr_aligned = mmap(
    NULL,                   // Kernel chooses the address at which to create the mapping.
    PAGE_SIZE,              // Number of Bytes to initialise mapping with.
    PROT_READ | PROT_WRITE, // Allow both reading and writing.
    MAP_SHARED,             // Mapping is preserved across fork(2) system calls
    fd,                     // This refers to /dev/mem/
    ip_addr_aligned         // Page aligned physical address as offset from fd.
);
close(fd);

// The 0x10 refers to the register offset of the "args" parameter as specified in the "xparameters.h" file.
*((volatile uint*)(ip_vaddr_aligned + align_offset + 0x10)) = 0xDEADBEEF;

// Unmap the page after use.
munmap(ip_vaddr_aligned, PAGE_SIZE);
```

Listing 4.2: *Interfacing with an IP core from a Linux C/C++ application. It is possible to directly write the value `0xDEADBEEF` to a memory-mapped register after mapping the `/dev/mem/` file, which represents the physical memory, into the virtual address space of the calling process. Note that any error handling code has been left out for the sake of clarity.*

Interfacing with the generic block design

The use of `/dev/mem/` to access the IP registers using their physical adresses effectively allows for a very similar interfacing with the generic block design as with the bare metal case presented in Section 4.1, except from the following points:

- The standalone drivers that come with most IP cores are not (directly) portable to a Linux environment. The IP cores – including the AXI-stream switch and the DMAs – are controlled by directly accessing the registers using the `/dev/mem/` method. Nevertheless, the drivers can serve as a good basis to illustrate how a core is intended to be used.

- The allocation of input and output buffers in step 3 requires special attention. First of all, pointers returned by `malloc` for example, refer to virtual addresses of which the physical address is unknown (an not straigtforward to retrieve). This renders the DMA components useless, because they require the physical address and length of the buffer to read from or write to. Even if the physical address were known, large arrays that are contiguous in virtual memory would rarely be contiguous in physical memory because of the use of demand paging by the operating system. A solution is to use contiguous memory allocation (CMA), in which a device driver allocates a contigouous portion of the physical memory at boot- and/or run-time. On the Zynq board, the `xlnk` driver allows for such functionality. By default, around 130MB of data can be contiguously allocated on the PYNQ-Z2 board, but this limit can be altered when configuring the Linux kernel.

- The flushing and invalidation of the cache can also be handled by the `xlnk` driver, at least for buffers that are contiguously allocated with it.

- The `/dev/mem/` method does not allow for interrupt handling. Software polling can be used or, alternatively, a (UIO) device driver can be written to manage interrupts.

- The use of Linux allows the programmable logic to be reconfigured at run-time, which means that different overlays can be loaded during the run-time of a program. The Xilinx FPGA manager is a device driver that is created for this purpose and allows to download a `.bin` file (which is part of the `.bit` file) onto the FPGA.

To conclude, it is possible to communicate between the PS, which is running a C/C++ application in Linux, and the PL, which contains IP cores that can be controlled using their AXI(-lite) interfaces, by making use of the `/dev/mem/` file to map physical regions of the memory into the application's virtual address space. Contiguous memory allocation should be used whenever buffers are to be transferred using direct memory access.

4.3 Python interfacing

The final step towards accelerating numerical libraries in Python is the communication between a running Python process and the IP cores that reside on the programmable logic. The most straightforward method is to make full use the PYNQ framework, in which a high-level Python representation is made of the currently loaded hardware design and in which the individual IP cores can be controlled using drivers written in Python. As will be discussed in Chapter 6, this also introduces run-time overhead which deteriorates the performance that can be achieved by combining the power of the processing system and the programmable logic. In order to counter this effect, user space drivers can be written in C or C++ using the method outlined in Section 4.2 and Python bindings can be generated to bridge the gap. More specifically, the pybind11 and C Foreign Function Interface (CFFI) bindings are considered.

In this section, the usage of PYNQ to interface between the PS and PL will be discussed first. Afterwards, the pybind11 and CFFI bindings are considered. Finally, light is shed on the automatic creation and benchmarking of these drivers in order to improve the developer's productivity.

4.3.1 PYNQ

The PYNQ framework offers the possibility to exploit the PL within a Python environment by means of hardware libraries or *overlays*. These overlays are essentially high-level representations of hardware designs that can be loaded onto the FPGA, and whose functionality is made available to the user as a Python API. In such a way, developers can import hardware designs much like software libraries and exploit the benefits of the programmable logic while staying at the pure software level.

Overlays

The creation of an overlay requires two files: a bitstream (`.bit`) file that can be loaded onto the FPGA, and a hardware handoff file (`.hwh` file), which represents the actual layout of the hardware design including the register mappings and offsets. Both files can be generated using the Vivado hardware design suite and are placed in a directory on the SD card of the board. When a developer instructs PYNQ to load a new overlay, it parses the hardware handoff file to create a high-level representation of the hardware design using Python objects. It also applies the PL clock configuration and downloads the bitstream file onto the FPGA.

The framework provides a Python API for a number of common AXI hardware components,

including GPIOs, I2C components, DMAs and video DMAs, and also allows to extend this API to support additional hardware components. Control of memory mapped IP cores (AXI or AXI-lite) is realised with the PYNQ `MMIO` class. Similar to what is discussed in Section 4.2, it makes use of the mapping of the `/dev/mem/` file into the virtual address space of the process using Python's built-in `mmap` functionality. Contiguous memory allocation for buffers that are shared between the PS and the PL is realised using the `PynqBuffer` class. It wraps the lower level `xlnk` drivers and is also a subclass of NumPy's `ndarray`, which makes it a flexible data structure that can easily interact with other NumPy structures.

Interfacing with the generic block design

The high-level interfacing in Python allows the code to be rather compact, as is shown in Listing 4.3. In the example, the (used part of the) overlay consists of a single DMA (`dma_0`), an AXI-stream switch (`axis_switch_0`) and a custom IP core (`custom_ip`). First, the overlay is initialised and the two-dimensional input and output buffers (`x1` and `out` respectively) are allocated in contiguous memory. Next, the PYNQ `MMIO` methods are used to pass arguments to the custom IP core and to start it. In this case, the height and width of the input are passed as 16-bit parameters to the 32-bit register of the core at an offset of `0x10`. Likewise, the routes within the AXI-switch can also be programmed by using memory-mapped I/O. Finally, the incoming and outgoing DMA transfers are started using the default PYNQ drivers. Note that the buffer passed to the `transfer` is required to be a (contiguously allocated) `PynqBuffer`.

It should be noted that, in order to increase the software developer's productivity, PYNQ promotes the use of Jupyter Notebooks to run Python code directly on the target board from a host computer that is connected to the board via Ethernet.

4.3.2 C/C++ bindings for Python

As will be shown in Chapter 5, the overhead of the device/overlay management and the performance of the Python classes to manage data transfers prevent the framework from reaching its full potential performance. To counter this, high performance drivers are created in a low-level language such as C or C++ and they are invoked instead of the PYNQ drivers. The drivers can be called from Python using CFFI, which supports C, and pybind11, which supports C++.

CFFI

The C Foreign Function Interface (CFFI) is a package that allows Python code to interact with almost any C code. CFFI can be used in one of four modes: *ABI* versus *API* mode, each

```python
from pynq import allocate, Overlay

ol = Overlay("<path to bitstream file>")

x1 = allocate((64, 64), dtype="f4")
out = allocate((64, 64), dtype="f4")

ol.custom_ip.mmio.write(0x10, (x1.shape[0] << 0) | (x1.shape[1] << 16))     # Set args
ol.custom_ip.mmio.write(0x00, 0x01)                                         # Start the IP core

ol.axis_switch_0.mmio.write(0x40 + 4 * i, j)            # Route master i to slave j
ol.axis_switch_0.mmio.write(0x40 + 4 * i, 0x8000_0000)  # Disable master i
x = ol.axis_switch_0.mmio.read(0x00)
ol.axis_switch_0.mmio.write(0x00, (x & 0x2))            # Do the update

ol.axi_dma_0.sendchannel.transfer(x1)   # Start the MM2S DMA transaction
ol.axi_dma_0.recvchannel.transfer(out)  # Start the S2MM DMA transaction
ol.axi_dma_0.recvchannel.wait()         # Wait until transaction is done
```

Listing 4.3: *Interfacing with the generic block design from high-level PYNQ code. In this example, the (used part of the) overlay consists of a single DMA (`dma_0`), an AXI-stream switch (`axis_switch_0`) and a custom IP core (`custom_ip`). The registers are controlled using PYNQ's `MMIO` class, while the DMA component is handled using the default PYNQ DMA drivers.*

with an *in-line* or *out-of-line* compilation option. The ABI mode accesses the libraries at the binary level, whereas the faster API mode accesses them with a C compiler. In the in-line mode, everything is set up every time the Python code is set up, while in the out-of-line mode, there is a separate C compilation step that produces a module that can be imported by the main program [51]. For performance reasons, the out-of-line API mode is used, in which the C driver sources are directly specified in the build script that compiles the final library.

pybind11

An alternative to CFFI is pybind11, a lightweight header-only library that exposes C++ types in Python and vice versa. The PYNQ framework allows to use a special cell magic `%%pybind11` inside their Jupyter Notebooks to automatically create Python bindings for the C++ code present in the corresponding cell. An advantage of pybind11 over CFFI is that it has built-in support for handling NumPy arrays and, for example, exposes its attributes such as dimensions and strides to the C++ code.

Interfacing with the generic block design

The Python code that is used to interface with the generic block design via low-level C or C++ drivers is shown in Listing 4.4. The low-level drivers that control the IP cores (including the AXI-stream switch and DMAs) are compiled into a shared object (`.so`) using CFFI or pybind11, and can be directly imported into Python as a module. The higher level `custom_ip_python` function invokes the driver by providing it with the correct arguments, which, in this case are the physical addresses of the two-dimensional in- and output buffers as well as their height and width [4].

```python
#   // This is a part of the C driver that is compiled into `custom_ip_bindings.so`.
#   void custom_ip_c(uint x1_addr, uint out_addr, uint height, uint width)
#   {
#       init(); // `mmap` the relevant address ranges into virtual address space.
#       control_ip_cores(); // Use mmap'ed registers to control the IP cores.
#       exit(); // `munmap` again.
#   }

from custom_ip_bindings import lib

def custom_ip_python(x1, out):
    assert x1.ndim == out.ndim == 2, "Inputs not supported!"
    lib.custom_ip_c(x1.physical_address, out.physical_address, *x1.shape)

if __name__ == "__main__":
    from pynq import allocate, Overlay

    ol = Overlay("<path to bitstream file>")

    x1 = allocate((64, 64), dtype="f4")
    out = allocate((64, 64), dtype="f4")

    custom_ip_python(x1, out)
```

Listing 4.4: *Interfacing with the generic block design from Python code that binds to lower-level C/C++ drivers. The* `custom_ip_python` *function invokes the lower level C/C++ driver* `custom_ip_c` *(of which a part is shown in the comment), and passes any required arguments to it.*

Overall, this way of interfacing between the processing system and the programmable logic from Python, is very similar to the Linux methodology presented in Section 4.2. One notable exeption is that the contiguous memory allocation (CMA) is done implicitly by using the `PynqBuffer` class, which wraps around the `xlnk` driver, and is therefore also responsible for flushing and

[4] Note that in the example, tuple unpacking is used to pass the height and the width of the input buffer based on its shape. From a C/C++ point of view, four parameters are given, which matches the signature of `custom_ip_c`.

invalidating the appropriate cache lines before and after DMA transfers.

4.3.3 Automating the driver generation

As part of this work, a script was created to generate most of the boilerplate code that is required when writing both higher-level Python, and lower-level C/C++ drivers. The script takes as an input the hardware handoff file (`.hwh`) that Vivado generates and produces a Jupyter Notebook that can be run directly on the Zynq platform.

It includes the following functionality:

- Semi-automatic creation of PYNQ, CFFI and pybind11 drivers for each IP core in the design. For each core in the hardware design, boilerplate code is generated to control the AXI-stream switch, the DMA(s) and to start the core. Manual adjustments are still required to, for example, set IP-specific parameters or to adjust the naming or signatures of the generated drivers.
- Automatic creation of a `"defines.h"` file that is used by the CFFI and pybind11 drivers. The header file contains getters and setters for all memory mapped registers in the hardware design and also provides `init()` and `exit()` functions that implement the `mmap`-ing of the `/dev/mem/` file. It resembles the `<xparameters.h>` file that is used in bare metal applications, but it doesn't require the Vitis software to be used or installed.
- Creation of a Python/NumPy stub for each custom IP core, which is used as a reference implementation and allows to validate the results produced by the drivers/cores.
- Generation of benchmarks between the Numpy, PYNQ, CFFI and pybind11 implementations. A graph of the run time vs. the input stream size is generated, and the values of the stream size for which the hardware implementations (PYNQ, CFFI, pybind11) provides acceleration are determined. These values are essential in the Python module that is presented in the next chapter.

The tool was used to create drivers for several hardware designs that make use of the custom high-level synthesised IP cores presented in Chapter 3. In the next chapter, the usage of these drivers to transparently accelerate numerical applications in Python is outlined.

5

Software design & implementation

In Chapter 4, the interfacing possibilities between the processing system and the programmable logic have been discussed, along with the creation of drivers for the IP cores that are present in a hardware design. To be able to transparently use these drivers to accelerate numerical libraries in Python, and NumPy in particular, a mechanism needs to be put in place that:

- is aware of the available hardware accelerators on the system and of their capabilities,
- is able to intercept Python method calls and check if they can be accelerated,
- is able to forward the arguments to the appropriate driver in case an acceleration is possible,
- and, is able to fall back to the default implementation when this is not the case.

The Python module that was created to implement this functionality was called ZyPy, which is a contraction Zynq and Python. First, an overview is given of the ways in which ZyPy manages the available hardware accelerators (bitstreams and drivers) on the system. In the next section, it is discussed how ZyPy is able to transparently overload function calls and invoke the required drivers when the method call can be accelerated using the programmable logic. Finally, the module that can be used as a drop-in replacement of NumPy, is introduced.

5.1 Hardware accelerator management

The ZyPy module keeps track of all hardware-accelerable methods and their corresponding drivers using a dictionary, as is shown in Listing 5.1. Each method (e.g. `add.__call__`[1]) is mapped to one or more hardware accelerated implementations (`add_call_f4_cffi` in this case). It is also possible to have multiple implementations for a single method, in case different hardware designs have implemented the same functionality or to support different classes of inputs (e.g. floating point vs. integer). Each implementation requires the following fields:

- `driver` This field refers to the method that can be called instead of the original Python method. It has the same signature – except for any optional arguments or `kwargs` – and invokes the CFFI or pybind11 driver that interfaces with the programmable logic. In fact, it adapts the signature of the Python method to that of the lower-level C/C++ driver. In the example, the physical addresses and lengths of the `PynqBuffers` that are involved in the operation are passed to the drivers.

- `accepts` This field contains a function or lambda that checks whether the given set of input arguments (`args` and `kwargs`) is compatible with the driver. It can for example check if the number of input arguments matches with the hardware driver, and if the given types and dimensions are supported.

- `provides_acceleration` This field contains a function or lambda that checks whether it is beneficial to execute the function on the programmable logic or not, given the number of elements `x` that need to be processed. The threshold can be determined by the benchmark that is generated when new drivers are created using the tool described in Section 4.3.3.

- `provides_acceleration_reconfig`: This field is similar to the previous one, but takes into account the FPGA reconfiguration overhead. When the bitstream associated with the driver is currently not active, it needs to be downloaded onto the FPGA. The execution time will typically increase by 0.2 to 0.3 seconds, which causes this threshold to be much higher.

- `bitfile` This field contains a string with the absolute path to the `.bit` file (which contains the bitstream) associated with the current driver.

- `imports` This field contains a string that specifies which CFFI or pybind11 modules need to be imported when the corresponding bitfile is loaded for the first time. This prevents importing all possible libraries when the ZyPy module is loaded.

[1]Note that the `.__call__` is used because the `add` operation is a NumPy ufunc that also supports other methods such as `.reduce` and `.accumulate`.

```python
class ZyPy():

    def add_call_f4_cffi(x1, x2, out):
        calc.add_call_f4(x1.physical_address, x2.physical_address, out.physical_address, x1.nbytes)

    def subtract_call_f4_cffi(x1, x2, out):
        calc.subtract_call_f4(x1.physical_address, x2.physical_address, out.physical_address, x1.nbytes)

    # ...

    hw_accelerators = {
        "add.__call__": {
            "add_call_f4_cffi": {
                "driver": add_call_f4_cffi,
                "accepts": lambda args, kwargs: len(args) == 2 and args[0].shape == args[1].shape
                                                and args[0].dtype == args[1].dtype == np.float32,
                "provides_acceleration": lambda x: x > 0,
                "provides_acceleration_reconfig": lambda x: x > 524288,
                "bitfile": "/home/xilinx/jupyter_notebooks/zypy/overlays/ufunc_call_f4.bit",
                "imports": "from lib.ufunc_call_f4_cffi import lib as calc",
            },  # ...
        },
        "subtract.__call__": {
            "subtract_call_f4_cffi": {
                "driver": subtract_call_f4_cffi,
                "accepts": lambda args, kwargs: len(args) == 2 and args[0].shape == args[1].shape
                                                and args[0].dtype == args[1].dtype == np.float32,
                "provides_acceleration": lambda x: x > 0,
                "provides_acceleration_reconfig": lambda x: x > 524288,
                "bitfile": "/home/xilinx/jupyter_notebooks/zypy/overlays/ufunc_call_f4.bit",
                "imports": "from lib.ufunc_call_f4_cffi import lib as calc",
            },  # ...
        },  # ...
    }
```

Listing 5.1: *The ZyPy module makes us of a dictionary to store information about all available accelerators on the system and to link them with their respective Python method calls.*

5.2 Overloading mechanism

Now that the module is aware of the available hardware accelerators, it can intercept Python method calls at run-time and check if they can be accelerated. The ZyPy module distinguishes two cases: the overloading of NumPy universal functions (ufuncs), which operate on `ndarrays` (and therefore on `PynqBuffers`) in an element-by-element fashion, and the overloading of more generic methods. Their respective implementations are discussed in the following two subsections.

5.2.1 Overloading NumPy ufuncs

In NumPy, any class can define a `__array_ufunc__` method in order to override the behaviour of its ufuncs. If one of the input or output arguments of a ufunc has a `__array_ufunc__` method, it is executed instead of the default ufunc. If more than one of the arguments implements `__array_ufunc__`, they are tried in the order: subclasses before superclasses, inputs before outputs, otherwise left to right [13]. Implementing this method for the `PynqBuffer` therefore allows to forward the execution of the ufunc to one of the drivers that is present in ZyPy's hardware accelerators dictionary. More concretely, its implementation is described in Listing 5.2.

The method takes as inputs the called universal function (e.g. `add`), the invoked method (e.g. `__call__`) and the input arguments (e.g. `x1` and `x2`). First, the ZyPy dictionary is used to check whether any hardware accelerator exists for the current ufunc and method name. To minimise the run-time overhead, the interception is aborted when the size of the input is smaller than some predefined minimum threshold. Next, the method iterates over the possible implementations and checks whether any of them accepts the current set of input arguments (shape, type...). If a candidate has been found, the mechanism checks whether the associated bitfile is loaded onto the FPGA. If the bitfile is loaded, the low-level driver is invoked using the current set of inputs. It is also possible that the bitfile is not loaded, but the acceleration is still worth it, even when taking into account the reconfigation overhead. In that case the bitstream is downloaded, the relevant imports are done and the execution is handed over to the low-level driver.

When there is no hardware acceleration possible, the method invokes the default `ndarray` implementation of the ufunc. The input arrays are passed as a "view on an `ndarray`" instead of a "view on a `PynqBuffer`", in order to prevent intercepting them again recursively. To be able to support dynamic allocation of output buffers, a mechanism is present that allocates a new `PynqBuffer` when the `out` argument is not provided. This allows for operations such as `(c = ) a + b` without having to preallocate `c`.

```python
def intercept_ufunc(ufunc, method, *args, **kwargs):

    if (hasattr(args[0], "size") and args[0].size > MIN_INPUT_SIZE  # The game's not worth the candle.
        and f"{ufunc}.{method}" in ZyPy.hw_accelerators):           # Check whether an accelerator exists.
        for impl in ZyPy.hw_accelerators[method].values():
            if impl["accepts"](args, kwargs):

                bitfile_loaded = (ZyPy._current_bitfile_name == impl["bitfile"])

                if(bitfile_loaded and impl["provides_acceleration"](args[0].size)):
                    pass
                elif(not bitfile_loaded and impl["provides_acceleration_reconfig"](args[0].size)):
                    ZyPy.download_bitfile(impl["bitfile"])  # Download the new bitfile.
                    exec(impl["imports"], globals())        # Do the imports required to access the drivers.
                else:
                    continue  # No acceleration possible using current implementation.

                # ... (Code to handle output allocation is left out)

                return impl["driver"](*args, **kwargs)  # Run the lower-level driver.

        f = {
            "__call__": ufunc,
            "reduce": ufunc.reduce,  # ... (More methods are left out)
        }

        # ... (Code to cast inputs and outputs to ndarrays is left out)

        return f[method](*view_as_ndarray(*inputs), **kwargs)  # Run the default NumPy implementation.
```

Listing 5.2: *Mechanism used to forward the execution of NumPy ufuncs to a hardware accelerator at run-time. When no acceleration is possible, the default implementation is used.*

5.2.2 Overloading other methods

ZyPy also supports the overloading of non-universal functions, such as discrete Fourier transforms, statistical methods, linear algebra functions, filter operations, etc. This is achieved by creating a `intercept_custom` method inside the ZyPy class, which is very similar to the `intercept_ufunc` method. Unlike the latter, it does not return a value when there is no hardware acceleration possible. This allows the calling method to fall back to its default implementation, as is shown in Listing 5.3. In Python, it is possible possible to dynamically modify attributes of a class or module, which is known as *monkey patching* [52]. This allows to keep the original naming, as is shown in the example: the original `np.fft.fft` implementation is patched with the hardware-accelerated implementation `new_fft`, which can still invoke the original implementation by keeping a reference to it in the `old_fft` variable.

5.3 The ZyPy module

The resulting ZyPy module combines the hardware accelerator management and the run-time interception mechanism to transparently accelerate numerical methods in Python. The end user can simply `import zypy as np` to benefit from the hardware accelerators while writing regular NumPy code, as shown in Figure 5.1. The user should as well be aware of some the restrictions that apply in order to accelerate their functionality. These limitations will be discussed in the following subsection.

5.3.1 Features & limitations

The following is a list of features and restrictions of the ZyPy module. Note that most limitations are inherent to the nature of the hardware design, rather than the software design.

Contiguous memory allocation

It is only possible to accelerate array methods when they are stored as contiguously allocated buffers, i.e. they are created using the `allocate` method. While the module allows compatibility with the default array structures, they will not be accelerated because the default methods will be used. Due to the limited amount of contiguous physical memory available, buffers that are no longer used should also be deallocated using the `freebuffer` method. Buffers that go out of scope are automatically deallocated.

```python
import numpy as np
from zypy import ZyPy

def new_fft(*args, **kwargs):
    res = ZyPy.intercept_custom("fft", *args, **kwargs)
    if res is not None:
        return res
    return old_fft(*args, **kwargs)

# Store the old implementation and monkey patch it with the new implementation.
old_fft, np.fft.fft = np.fft.fft, new_fft
```

Listing 5.3: *Mechanism used to forward the execution of an arbitrary (NumPy) method to a hardware accelerator at run-time. When no acceleration is possible, the default implementation is used.*

```python
import zypy as np

a = np.allocate(shape=(1_000_000,), dtype=np.float32)
b = np.allocate(shape=(1_000_000,), dtype=np.float32)

a[:] = np.arange(a.size)
b[:] = 2

print(a + b)
print(a - b)
print(a * b)
print(a / b)
```

```
[2.000000e+00 3.000000e+00 ... 9.999990e+05 1.000000e+06 1.000001e+06]
[-2.00000e+00 -1.00000e+00 ...  9.99995e+05  9.99996e+05  9.99997e+05]
[0.000000e+00 2.000000e+00 ... 1.999994e+06 1.999996e+06 1.999998e+06]
[0.000000e+00 5.000000e-01 ... 4.999985e+05 4.999990e+05 4.999995e+05]
```

Figure 5.1: *The end user can simply* `import zypy as np` *to benefit from the hardware accelerators while writing regular NumPy code*

.

Reconfiguration overhead

The ZyPy module only considers a single execution of a method at a time. When the bitstream of the corresponding operation is loaded, this is not a problem. However when the bitstream is not loaded, the interception mechanism will not spread the reconfiguration overhead over the number of iterations, possibly leading to suboptimal results. The same scenario occurs when there are different methods that can be accelerated using the same bitstream. To counter this, ZyPy makes it possible for a user to add a `hint(<method>, <args>)` call before the actual method is invoked. This effectively downloads the bitstream that matches the given method and set of input arguments and allows any subsequent method calls to be accelerated. Automating this process requires a thorough analysis of the program using, for example, parsing of the Python abstract syntax tree [53].

Array broadcasting

In NumPy, it is possible to treat arrays with different shapes during arithmetic operations using *array broadcasting*. Subject to certain constraints, a smaller array can be broadcast across the larger one so they have compatible shapes, usually without having to make needless copies of the data in memory. As an example, it is possible to do an operation between an array and a scalar by repeatedly reusing the scalar value. There are several possibilities to support array broadcasting:

- Explicitly allocate a buffer into which copies of the smaller array or scalar to be broadcast are placed. This is illustrated in Figure 5.1 on line 7, in which the value `2` is copied into buffer `b`. This is the simplest solution, but comes at the cost of an increased memory usage and run-time overhead to due to the copy operations.
- Implement the broadcasting mechanism inside the custom IP core that is used as an accelerator. This necessitates storing the array to be broadcast in the local memory of the core (e.g. BRAM), and also requires the dimensions of both arrays to be passed to it. The accelerator can then keep track of the element that is currently accessed by using an n-dimensional iterator, which can be implemented in hardware using for example ripple counters.
- Another possibility to implement array broadcasting is to make use of the scatter/gather-mode of the DMA components. This mode makes it possible to offload some of the DMA management from the CPU by describing multiple transfers at once in a memory-mapped structured called a buffer descriptor (BD) ring. The ring consists of buffer descriptors that each detail which data needs to be transferred from or to the DRAM. This allows

the scatter/gather engine to do a rapid succession of transfers, possibly in a cyclic mode of operation. The buffer descriptor can be placed in a memory location that is separated from the data buffers (e.g. in BRAM) using the dedicated AXI scatter/gather master interface (`M_AXI_SG`), as can be seen in Figure 3.2. Due to the increased complexity, this mode takes more programmable logic resources than the direct register mode.

The scatter/gather principle applied to the array broadcasting is exemplified in Figure 5.2. The yellow, red and green structures represent arrays in memory, using a row-major type of ordering. Suppose that an operation is required between the array shown in yellow and the one in green, which needs to be broadcast to the same shape. In that case, a single buffer descriptor in cyclic mode (5x) suffices to describe the transactions that the DMA needs to handle. On the other hand, when the red array is broadcast to the shape of the yellow one, twenty entries are required to describe the entire transaction, which is clearly not optimal[2]. Overall, this method may only be viable for certain types of broadcast operations.

Array slicing and masking

In Python, it is possible to select a only as subset of a list of object by using the slice operator `[start:stop:step]` on it. NumPy extends this concept to multiple dimensions by using a very similar syntax `[start:stop:step, start:stop:step, ...]`. However, this basic slicing mechanism merely creates a new view over the same data in memory. This implies that (most of the time) sliced arrays are not contiguous in physical memory, even if the base array is contiguous. Possible methods to support slicing are similar to the ones presented in the context of array broadcasting:

- Explicitly allocate a buffer into which the sliced part of the original array is copied into. Again, this comes at the cost of an increased memory usage and run-time overhead to due to the copy operations.
- Implement the slicing mechanism inside the custom IP core, send the slice information as parameters to the intellectual property (IP) core and stream the complete, original, array. This allows the DMA to keep operating in a simple mode, but may also introduce a lot of wasted cycles in case the sliced data is much smaller than the original data.
- Implement array masking inside the custom IP core. A masked array is the combination of a standard `ndarray` and a boolean mask, which indicates the positions of the elements

[2] In fact, the CPU would need to write fourty values to the memory allocated for the buffer descriptor (twenty for the addresses and twenty for the lengths), while an explicit copy would only require twenty copies.

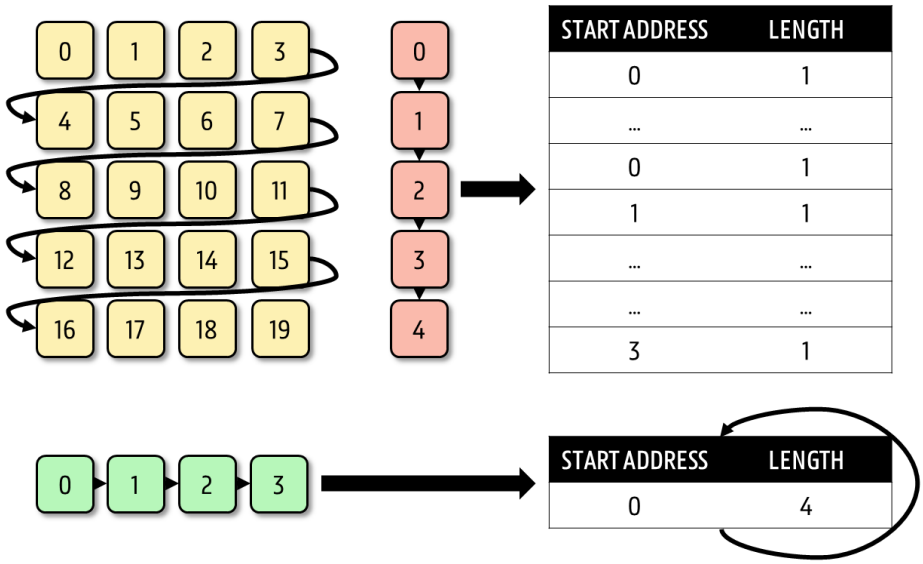

Figure 5.2: *Illustration of the use of buffer descriptors to describe the DMA transactions that are required to broadcast an array and send it to the programmable logic. When the green array is broadcast to the shape of the yellow one, a single descriptor in cyclic mode suffices. In contrast, the broadcasting of the red array to the shape of the yellow one, requires twenty entries and is clearly not optimal.*

that are invalid or need not to be considered for the specific application. This can be realised by sending an additional stream in parallel with the regular array stream and by applying the mask before doing an operation inside the accelerator. However, again a lot of bandwidth is lost when many elements are masked.

- Finally, it is also possible to implement slicing using the scatter/gather-mode of the DMA components, similar as is discussed in the previous subsection. Some DMA components have built-in support for such strided access patterns[3].

[3]See AXI DMA LogiCORE IP Product Guide.

6

Results

Now that both the hardware and software implementations have been described along with the possibilities to interface between the two, the experimental results can be presented. First, the test setup and the implemented hardware accelerators are discussed. Then, insights are given into the resource usage of the realised hardware designs. Next, the performance of the individual accelerator cores is evaluated. Finally, the entire ZyPy module, including the run-time interception mechanism, is evaluated using the heat equation benchmark.

6.1 Test setup and implemented hardware accelerators

All tests were carried out on the PYNQ-Z2 board (ZYNQ XC7Z020-1CLG400C), which is based on the Xilinx Zynq SoC. It includes a dual-core Cortex -A9 processor clocked at 650Mhz, 512MB of DDR3 memory, and its programmable logic is equivalent to an Artix-7 FPGA and runs at 100MHz. More concretely, it consists of 13,300 logic slices (each with four 6-input LUTs and 8 flip-flops), 630 KB of block random-access memory (BRAM) and 220 DSP slices.

Several accelerator cores and hardware block designs were created for the purpose of this work. The IP cores that will be considered in this chapter are summarised in Table 6.1. Most cores are implemented in a generic way and allow for rapid addition of new functions, depending on

the application that is targeted. It should be noted that the cores are designed to work with 32-bit data types, which is also the default size used in the PYNQ framework, while NumPy uses 64-bit data types by default. Five block designs were created in which only a single accelerator is placed (but note that it is still possible to select multiple operations within that core). A final block design includes two hardware accelerators and illustrates the use of the AXI-stream switch mechanism to allow multiple accelerators to be used. It is tailored towards a specific application, which in this case is finding a numerical solution to the two-dimensional heat equation.

6.2 Resource utilisation

Table 6.2 provides a summary of the resource utilisation of the different hardware designs. For each design, the total utilisation of block random-access memory (BRAM), DSP slices, flip-flops (FFs) and lookup tables (LUTs) is indicated in bold. For each block design, the resource usage of the included accelerators, as estimated by the Vitis HLS tool, is also shown (in *italics*). As can be seen from the table, all block designs still allow for more functionality to be added, and they are not constrained in any way. It should be noted that the resource utilisation reports generated by the high-level synthesis tool should be handled with care. They are merely an estimate and are still subject to changes, as can be seen from the LUT utilisation of the `ufunc_call_f4` block design. The LUT utilisation is reported to be around 91%, while after synthesis, this is only 38%, because some of the lookup tables used as memory have been replaced by block random-access memory (BRAM). It may not be feasible to re-synthesise the complete hardware design during the development of an accelerator, just to take note of its resource utilisation. Nevertheless, the developer should be aware that such scenarios may occur.

It is also worthwhile to consider the consumption of the IP cores that are not used directly as accelerators, but serve to support them, e.g. the AXI-stream switches, the DMA components and the AXI interconnects. The DMA components (configured in direct register mode) take around 1.5% of the LUTs, 1% of the FFs and 3% of all available BRAM per read or write channel that is in use. The AXI-stream switch uses less than one percent of the available LUTs and FFs and has a limited impact. A final component that is worthwhile mentioning is the main AXI interconnect to the processing system, which takes a little over 1% of all lookup tables.

Table 6.1: *Overview of the implemented hardware accelerators and their functionality.*

Accelerator name	**Supported data type**	**Functionality**
`ufunc_call_f4`	32-bit floating point	Element-wise addition, subtraction, multiplication, true division, sine, square root, arctan, hyperbolic sine, natural exponentiation and natural logaritm.
`ufunc_call_i4`	32-bit signed integer	Element-wise addition, subtraction, multiplication, true division, square root, natural exponentiation and natural logaritm.
`ufunc_reduce_all_f4`	32-bit floating point	Complete reduction using addition, subtraction, multiplication, true division, (base 2 or natural) logarithm of sum of natural exponentials, dot product, sum of squares and sum of absolute differences.
`ufunc_reduce_all_i4`	32-bit signed integer	Complete reduction using addition, subtraction, multiplication, true division, (base 2 or natural) logarithm of sum of natural exponentials, dot product, sum of squares and sum of absolute differences.
`filter_avg_f4`	32-bit floating point	Average filtering of two-dimensional input array.
`filter_avg_f4`	32-bit floating point	Average filtering of two-dimensional input array combined with the sum of absolute differences.
`sad_reduce_all_f4`	32-bit floating point	Sum of absolute differences.

Table 6.2: *Overview of the resource usage of the realised block designs (shown in **bold**). For each block design, the resource usage of the included individual accelerators, as estimated by the Vitis HLS tool, is also shown in italics.*

Block design	BRAM	DSP	FF	LUT
ufunc_call_f4_bd	**18 (12%)**	**64 (29%)**	**25405 (23%)**	**20671 (38%)**
ufunc_call_f4	*0 (0%)*	*51 (23%)*	*32078 (30%)*	*48916 (91%)*
ufunc_call_i4_bd	**19 (13%)**	**3 (1%)**	**11047 (10%)**	**7556 (14%)**
ufunc_call_i4	*2 (0%)*	*0 (0%)*	*4824 (4%)*	*4347 (8%)*
ufunc_reduce_all_f4_bd	**9 (6%)**	**50 (22%)**	**8089 (7%)**	**9340 (17%)**
ufunc_reduce_all_f4	*0 (0%)*	*50 (22%)*	*5763 (5%)*	*8891 (16%)*
ufunc_reduce_all_i4_bd	**11 (7%)**	**12 (5%)**	**5490 (5%)**	**5226 (9%)**
ufunc_reduce_all_i4	*4 (1%)*	*0 (0%)*	*3807 (3%)*	*4283 (8%)*
filter_avg_f4_bd	**9 (6%)**	**11 (5%)**	**6621 (6%)**	**8565 (16%)**
filter_avg_f4	*0 (0%)*	*11 (5%)*	*3389 (3%)*	*3340 (6%)*
heat_eqn_f4_bd	**13.5 (9%)**	**15 (6%)**	**10539 (9%)**	**12277 (23%)**
filter_avg_f4	*0 (0%)*	*11 (5%)*	*3389 (3%)*	*3340 (6%)*
sad_reduce_all_f4	*4 (1%)*	*0 (0%)*	*2568 (2%)*	*2605 (4%)*

6.3 Performance analysis of the hardware accelerators

Four operations were selected from the available hardware accelerators to assess the benefits of carrying out certain operations on the FPGA:

- **multiply_call_f4**, which represents an element-wise operation in which one floating-point operation is done per cycle (i.e. 1 FLOP/cycle).
- **add_reduce_all_f4**, which also represents an element-wise operation, but with data dependencies between the different iterations. This operation also requires 1 FLOP/cycle.
- **logaddexp_reduce_all_f4**, which represents a combination of the above and hence requires 2 FLOPs/cycle[1].
- **filter_avg_f4**, which requires four additions and a single multiplication in each cycle, leading to 5 FLOPs/cycle[2].

Six benchmarks were created to measure the performance of these operations at different levels:

- A NumPy (Python) implementation running on the CPU.

[1]Note that the final `log` operation is not considered here, so in fact it is slightly more than 2 FLOPs/cycle.

[2]Same remark, but now it will be slightly less than 5 FLOPs/cycle because the boundary elements are simply passed to the output.

- A C bare-metal implementation running on the CPU.
- A C bare-metal implementation that makes use of the FPGA accelerator.
- A PYNQ (Python) implementation that makes use of the FPGA accelerator.
- A Python implementation that makes use of the FPGA accelerator using CFFI bindings.
- A Python implementation that makes use of the FPGA accelerator using pybind11.

The results of these benchmarks are shown in Figure 6.1. Overall, the NumPy implementations are the worst performing, followed by the bare metal CPU and, surprisingly, the PYNQ implementation. A deeper investigation of the PYNQ implementation using Python's built-in cProfiler suggests that this discrepancy is due to the nature in which PYNQ communicates with the device, i.e. using a client-server architecture in which multiple hardware devices and multiple processes can communicate with each other. As can be seen from the graph, this overhead is detrimental and even allows the FPGA implementation to be outperformed by the bare metal CPU implementation. The other FPGA-accelerated implementations seem to be very similar and achieve a performance close to 0.1, 0.1, 0.2 and 0.5 GFLOP/s respectively. Taking into account that the programmable logic is clocked at 100 MHz (or a period of 10 ns), these results confirm that the hardware accelerators are indeed operating with an initiation interval equal to one.

The results shown in Figure 6.1 are based on benchmarks that make use of fixed-size, but large streams of test data. In order to better map the PS/PL interfacing overhead, the different FPGA-accelerated implementations of the `multiply_call_f4` operation were compared with the NumPy implementation for different values of the input stream length, as is shown in Figure 6.2. On the left vertical axis, the run time per element can be observed when looking at the solid curves. The speedup relative to the NumPy implementation can be read off from the right vertical axis, when only considering the dotted lines. Notice that the left vertical axis and the horizontal axis make use of a logarithmic scale. As expected, the interfacing overhead is more noticeable for smaller input streams. For large values of the input size, the CFFI and pybind11 implementations approach a run time of 10 ns per element while the NumPy implementation takes more than 300 ns per element, which is consistent with the previous results. As can also be read off from the right axis, a speedup of a factor 35 is possible when using the FPGA rather than the default NumPy implementation. From the graph, it can also be seen that the PYNQ implementation is lagging well behind and only provides acceleration when the input stream length approaches one million elements. Surprisingly, the CFFI implementation even outperforms the NumPy implementation for input streams containing only a dozen of elements.

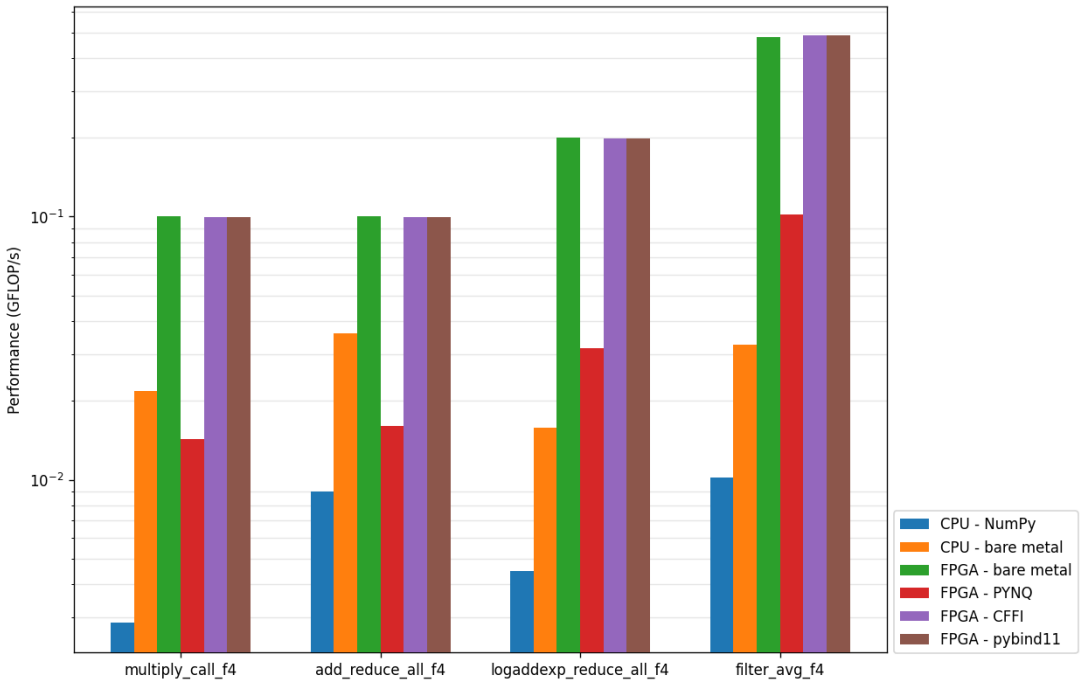

Figure 6.1: *Comparison of the performance of four different operations, each implemented six times: using the CPU (Python), using the CPU (C, bare metal), using the FPGA (C, bare metal interfacing), using the FPGA (Python, PYNQ interfacing), using the FPGA (Python, CFFI interfacing), and finally using the FPGA in combination with Python using the pybind11 interface.*

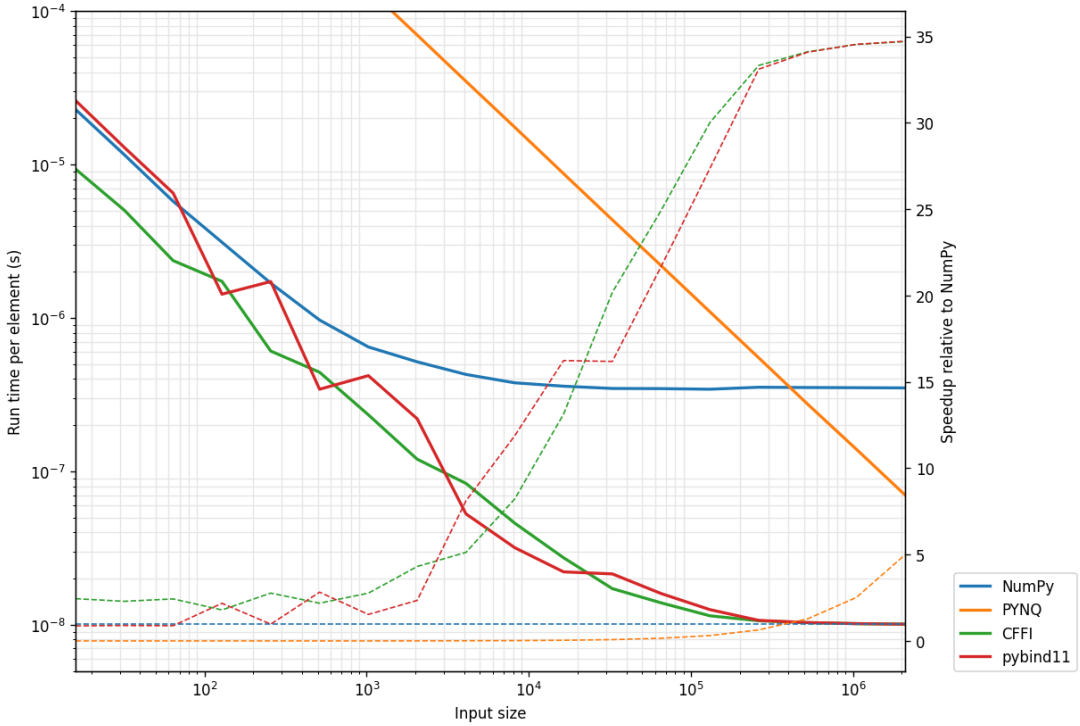

Figure 6.2: *Comparison of the different FPGA-accelerated implementations of the* `multiply_call_f4` *operation with the NumPy implementation, for different values of the input stream length. On the left vertical axis, the run time per element can be read off when combined with the solid coloured lines. On the right vertical axis, the speedup relative to NumPy can be observed, when only considering the dashed coloured lines.*

6.4 Performance analysis of the ZyPy module

In the previous section, the performance analysis was targeted at individual IP cores. In this section, these results are extended by analysing the performance of a practical application that makes use of the ZyPy module (i.e. using the run-time interception mechanism) as described in Chapter 5. First the used benchmark is discussed, afterwards the results and efficiency of the ZyPy module are considered.

6.4.1 Benchmark

The benchmark that is used to test the ZyPy module consists of the simulation of the heat transfer on a surface represented by a two-dimensional grid, implemented using Jacobi-iteration with numerical convergence [36]. As shown in Listing 6.1, a grid is initialised with zeros, and the boundaries are filled with arbitrary initial values. Inside the `while`-loop, each element is replaced by the average of itself and its four neighbours until there is convergence, i.e. the total sum of the absolute difference between the grid at iteration `i` and `i + 1` becomes smaller than the value of `epsilon`. Mind the use of array slicing to allow NumPy to vectorise its operations.

In Listing 6.2, a slightly modified version of the algorithm is proposed, which allows for a better cooperation with the ZyPy module. The averaging operation is replaced with the cutsom implementation of the filter operation that was discussed in Section 3.4, and the sum of absolute differences is replaced by the implementation that was discussed in Section 3.3. Furthermore, two grids are allocated in contiguous memory, of which `grids[0]` is filled similar to the other method. In the first iteration, `grids[0]` is used as input for the filter operation and the result is streamed back into `grids[1]`. In the second iteration, this order reverses: `grids[0]`, which contains the most recent value of the grid, is used as an input and `grids[0]` awaits the result[3]. This process is repeated until there is convergence, and makes efficient use of the available contiguous memory.

[3]While it may seem strange at first sight, it is possible to index a loop using a boolean value in Python. This allows to toggle the indices between each iteration in a convenient way.

```
def solve_numpy(h, w, epsilon=0.005):
    grid = np.zeros((h + 2, w + 2), dtype="f4")

    grid[:, 0] = -273.15
    grid[:, -1] = -273.15
    grid[0, :] = 0
    grid[-1, :] = -273.15

    C = grid[1:-1, 1:-1]
    N = grid[:-2, 1:-1]
    E = grid[1:-1, :-2]
    S = grid[2:, 1:-1]
    W = grid[1:-1, 2:]

    delta = epsilon + 1

    while delta > epsilon:
        tmp = 0.2 * (C + N + E + S + W)
        delta = np.sum(np.absolute(tmp - C))
        center[:] = tmp

    return center
```

Listing 6.1: *Original method.*

```
def solve_zypy(h, w, epsilon=0.005):
    grids = [np.allocate((h + 2, w + 2)),
             np.allocate((h + 2, w + 2))]

    grids[0][:] = 0
    grids[0][:, 0] = -273.15
    grids[0][:, -1] = -273.15
    grids[0][-1, :] = 0
    grids[0][0, :] = -273.15

    delta = epsilon + 1
    switch = False

    while delta > epsilon:
        np.filter_avg(
          grids[switch],
          grids[not switch]
        )
        delta = np.sad(grids[0], grids[1])
        switch = not switch

    return grids[0]
```

Listing 6.2: *ZyPy-optimised method.*

6.4.2 Benchmark results

Figure 6.3 shows the total run time of the benchmark versus the size of the input stream, for three different implementations: NumPy, PYNQ and ZyPy. Again, the speedup relative to the NumPy implementation can be read off from the rightmost vertical axis, when only considering the dotted lines. The NumPy implementation makes use of the code shown in Listing 6.1, while the PYNQ and ZyPy implementations make use of the adapted code shown in Listing 6.2. Note that the PYNQ implementation makes use of the same hardware accelerators as the ZyPy version, but not of the run-time interception mechanism.

As can be seen from the graph, the ZyPy implementation clearly outperforms the NumPy implementation, in particular for large inputs. For 64x64 inputs, the ZyPy implementation provides a speedup of a factor close to 2, while for 384x384 inputs, the ZyPy implementation is well over 10x faster than the NumPy implementation. Extrapolating the results to larger input sizes suggests that the run time of the NumPy implementation is nearly two orders of magnitude higher than the ZyPy-accelerated version, for input arrays as small as 512x512. As expected, the PYNQ implementation suffers from huge networking overheads and is therefore not suitable for this kind of application.

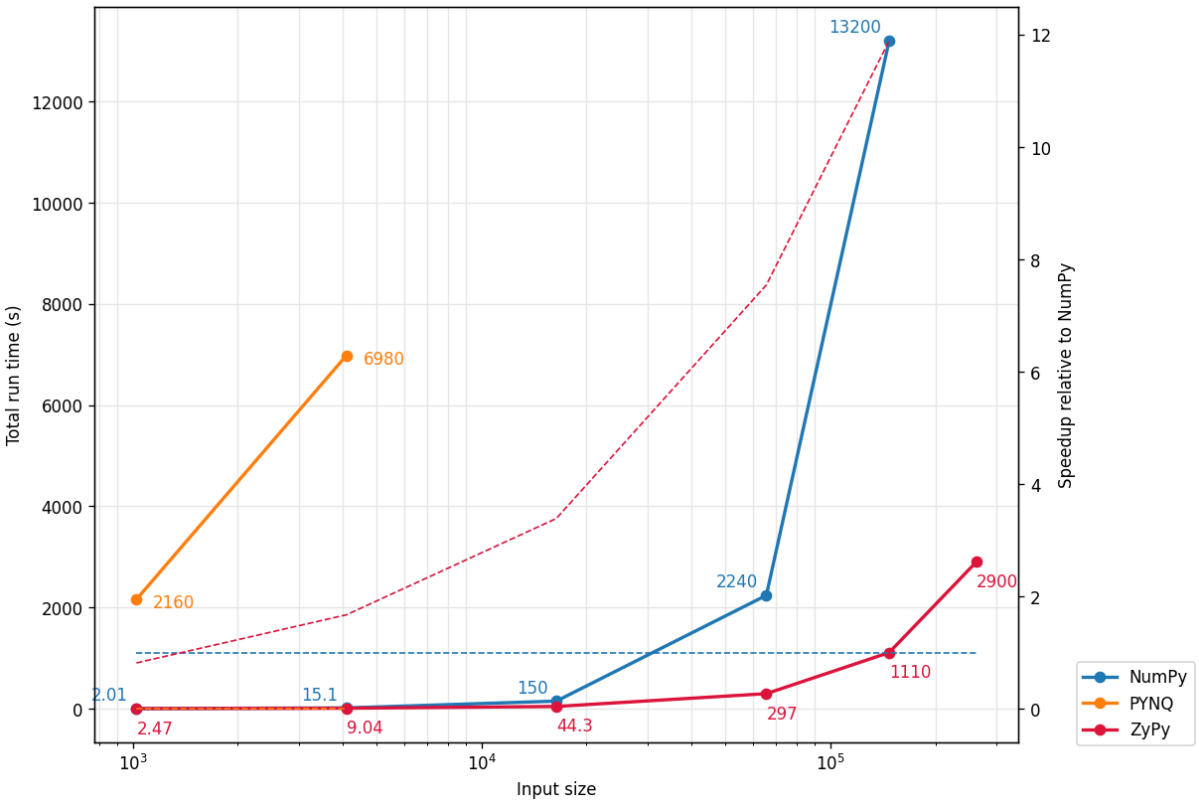

Figure 6.3: *Comparison of the run time of the Numpy, PYNQ and ZyPy implementations for the heat equation benchmark, for different values of the input stream length. On the left vertical axis, the total run time of the benchmark can be read off when combined with the solid coloured lines. On the right vertical axis, the speedup relative to NumPy can be observed, when only considering the dashed coloured lines.*

6.4.3 Analysis

Efficiency

To get a better understanding of the overhead of the run-time interception mechanism of ZyPy, its heat equation benchmarks were profiled using Python's built-in cProfiler. A profile is a set of statistics that describes how often and for how long various parts of the program executed. For the ZyPy benchmarks, most of the time is spent executing the CFFI/pybind11 drivers that are used to interface with the programmable logic. This is not surprising, as software polling is used to wait for each result. The graph in Figure 6.4 shows, for different input sizes, the percentage of time that is *not* spent executing these lower-level drivers, and hence provides an estimate of the ZyPy overhead. For very small inputs, the overhead is slightly below 50%, while it decreases and seems to stabilise around 5% for input sizes of around a quarter of a million elements (512x512 inputs). When taking a look at the functions that make up the overhead, most of the time seems to be spent during the evaluation of the (lambda) functions that check the conditions that the inputs must meet in order to be able to use the lower level drivers.

Bridging the gap: drivers speed-up vs. ZyPy speed-up

The results in Figure 6.2 suggest that the run time of the low-level drivers is about ten times smaller than the default NumPy run time, considering inputs in the range of 128x128 (16384) elements[4]. However, the ZyPy speedup shown in Figure 6.3 is "only" a factor of three to four. In order to explain this phenomenon, the run time is reviewed from a theoretical perspective as shown in Equations 6.1 and 6.2.

$$T_{np} = N \cdot \{t_{filt,np}(N) + t_{sad,np}(N)\} \cdot i \tag{6.1}$$

$$T_{zp} = \frac{N}{\eta(N)} \cdot \{t_{filt,zp}(N) + t_{sad,zp}(N)\} \cdot i + T_{rec} \tag{6.2}$$

The variables T_{np} and T_{zp} represent the total run time of the benchmark for the NumPy and ZyPy implementations, respectively. The $t_{filt,np}(N)$ and $t_{sad,np}(N)$ represent the per-element run time of the "filter" and "sum of absolute differences" operations for the NumPy implementation, while the $t_{filt,zp}(N)$ and $t_{sad,zp}(N)$ represent them for the ZyPy implementation. As can be seen from Figure 6.2, the per-element run time indeed depends on the value of N, the input size. The variable $\eta(N)$ represents the efficiency (one minus the overhead) of the ZyPy module,

[4]In fact, Figure 6.2 shows the results for the `_multiply_call_f4` accelerator, but the speedups are very similar to the `filter_avg_f4` and `sad_reduce_all_f4` accelerators.

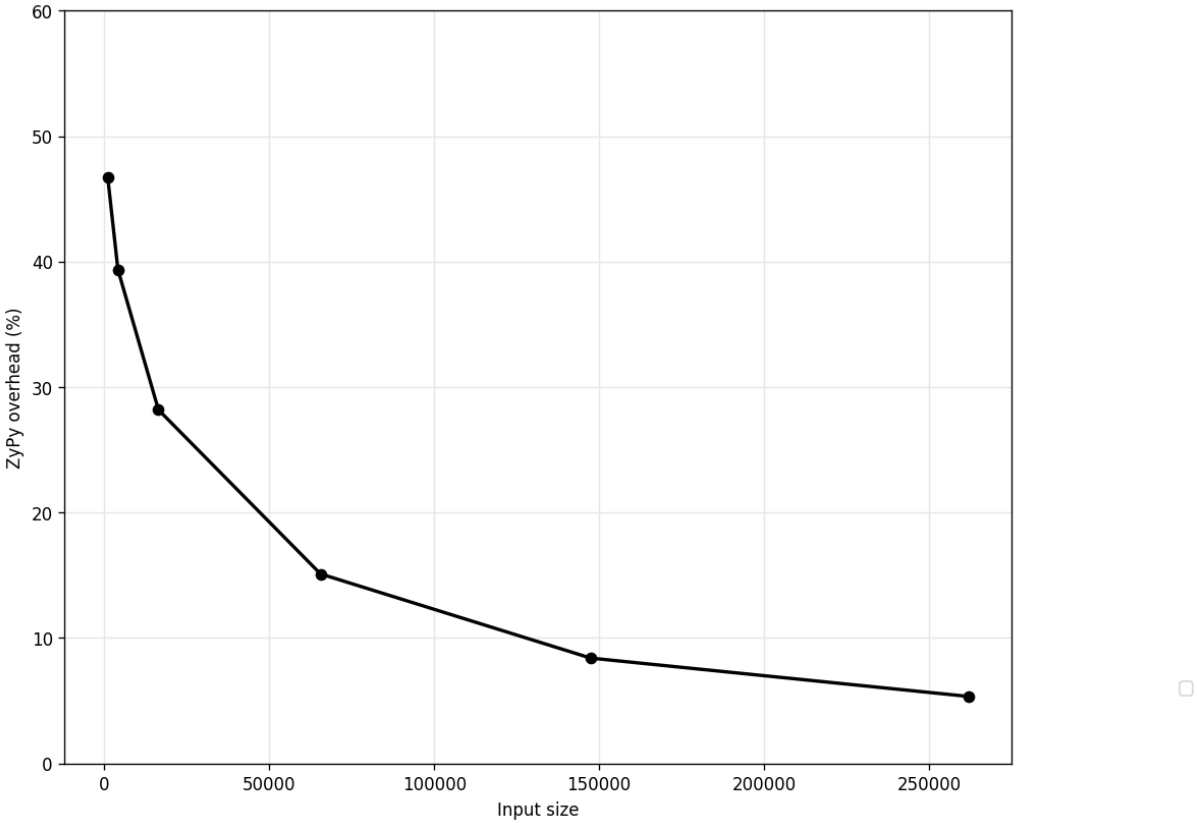

Figure 6.4: *Overhead of the ZyPy-mechanism on the run time of the heat equation benchmark.*

as is shown in Figure 6.4. Furthermore, i represents the number of iterations in the benchmark and T_{rec} contains the reconfiguration time of the FPGA, which is around 0.3 seconds.

Plugging in the values $N = 128 \cdot 128$, $\eta(N) \approx 1 - 0.29$, $i = 40801$ (experimentally), and the respective per-element run times (not shown in any graph), yield a total run time $T_{np} \approx 624$ s and $T_{zp} \approx 52$ s. The ZyPy estimate seems to be fairly close to the actual measured value of 44.3 seconds. However, the NumPy run time is severely over-estimated, since the actual value is just 150 seconds. The discrepancy is most likely due to the efficient use of caching, which is made possible by the fact that both operations use the same input data.

7

Conclusion and future work

This chapter first formulates a general conclusion and then presents some ideas and possible improvements for future research.

7.1 Conclusion

In this work, a methodology is devised that allows numerical applications in Python to be accelerated using FPGAs. In order to validate this methodology, a custom Python module that can be used as a drop-in replacement of NumPy, is implemented, in which routines are automatically and transparently forwarded to their hardware-accelerated counterpart on the FPGA, given that a compatible and profitable accelerator exists. To support this system, various NumPy routines are transformed into custom hardware accelerators by means of high-level synthesis, and integrated into Python using parts of Xilinx's PYNQ platform.

More concretely, three classes of NumPy operations are covered and optimised in hardware:

- Element-wise operations on `ndarrays` are covered by the `ufunc__call__xx` cores, which are able to apply a unary operation (arithmetic, goniometric, boolean...) on each element of its input stream, or a binary operation on both of its input streams, resembling the

invocation of the `__call__` method on a NumPy universal function.

- Reduction-type operations on `ndarrays` are covered by the `ufunc_reduce_all_xx` cores, which are able to apply a unary operation on each element of its input streams, or a binary operation on both of its input streams, followed by a repeated application of a binary operation to the resulting stream. This effectively reduces the stream to a single value. Furthermore, it is also possible to apply a final unary operation to this value. This functionality resembles the invocation of the `__call__` method on a NumPy universal function, followed by the `reduce` method.
- Filter-type operations on two-dimensional `ndarrays` are covered by the `filter_avg_xx` core, which is able to calculate the average of each input value and its four direct neighbours. This operation does not directly correspond to a NumPy method, but rather a combination of them (i.e. four sums and a multiplication). This core was created with the application, solving the two-dimensional heat equation on a grid, in mind.

Furthermore, a reusable hardware block design is created that allows multiple hardware accelerators to be combined, by making use of a stream switch. A tool is also devised that eases the creation of lower-level software drivers for each of the hardware designs. A speed-up by an order of magnitude is found to be possible in a practical application where the two-dimensional heat equation is solved numerically, while a direct invocation of the individual accelerators is able to outperform default NumPy operations by factors up to 35x. It is also concluded that the PYNQ framework, despite its flexibility and prototyping capabilities, is not able to achieve this kind of acceleration, which led to various parts of the framework being bypassed.

7.2 Future work

Even though the results seem to be encouraging, there is still potential for improvement and further research, in particular on the following topics:

Creating more hardware accelerators

To fully leverage the hardware acceleration that can be achieved using FPGAs, more NumPy functionality could be implemented in hardware. Owing to the vast number of functions that are available, choices must be made that are specific to the application in mind. On the other hand, one can also choose to implement more general procedures, which leads to the further development of the presented accelerators involving universal functions. An accelerator that was already partly established, but was not considered in this thesis as it was not optimal, is

one that is able to apply a reduction or accumulation type of operation along a single axis of an `ndarray`, instead of all axes at once.

Increasing the throughput of the existing accelerators

Even though all hardware accelerators presented in this work achieve an initiation interval of one, there are still possibilities to improve the throughput even further, by either increasing the clock frequency, or by allowing for more parallelism. A factor of two speed-up is expected when the clock frequency of the programmable logic is doubled from 100MHz to 200Mhz, which seems to be the upper limit for other IP cores, including the AXI switches and DMAs. However, this will in its turn result in longer pipelines with more stages, resulting in an increased resource usage. On the other hand, it is also possible to widen the input streams to and from the accelerators from 32-bits to a multiple of it, which allows the accelerators to fetch and process multiple elements in parallel. This will again result in an increased throughput, at the cost of more resource utilisation. There will also be a point at which it is not beneficial to add any more processing power, because of bandwidth limitations of the high-performance ports, and even the DRAM[1]. This mechanism has been tested with an element-wise sine calculation, and a factor of three speed-up was realised before the available bandwidth saturated.

Use of scatter/gather DMAs

As illustrated in Section 5.3.1, the use of the DMA engines in scatter/gather may provide several benefits over the direct register mode. Future work may incorporate the scatter/gather mode into the system and open new paths towards efficient support for array slicing, broadcasting and masking. Furthermore, it could also be interesting to check if the mode can be used to circumvent the contiguous memory allocation that is required in the current implementation.

Support for chaining operations

In the current implementation, it is only possible to bind a hardware accelerator directly to an individual NumPy method, or to an individual custom function. For example, the `np.sum(np.abs(x - y))` will invoke three separate accelerators, unless the developer explicitly uses the custom `np.sad(x, y)` method. This process could be automated by for example introducing lazy evaluation or by parsing the source code or the associated abstract syntax tree,

[1] In theory, each HP interface provides a bandwidth of 1.2GB/s for reading and writing, resulting in an aggregated bandwidth of 9.6GB/s. However, the double data rate (DDR) only has a total bandwidth of 4.264GB/s [42].

and by conducting a form of template matching.

Support for chaining accelerators

In the current implementation, it is only possible to chain hardware accelerators when the streams are immediately consumed. Introducing additional FIFOs stages in the hardware design allows for the (temporary) storage of intermediate results, and could even enable the use of feedback loops, which could speed up the heat equation benchmark even more. To achieve this goal, it may be possible to integrate some of the work presented in [40].

Use of dynamic partial reconfiguration

A better utilization of the FPGA resources and an increased efficiency could also be obtained by making use of dynamic partial reconfiguration of the hardware. In Xilinx terms, dynamic function exchange (DFX) enables the modification of parts of the programmable logic called reconfigurable partitions (RPs) at run-time [54]. During and after the reconfiguration process, the rest of the programmable logic is not interfered. Configuring smaller parts of the programmable logic reduces the time for the configuration process itself, may lead to a reduced power consumption, and most importantly, allows hosting more hardware accelerators because of the temporal partitionining [55, 20]. When combined with a lazy evalution mechanism on the Python side, dynamic partial reconfiguration may even enable the ability to load in the accelerator for the next operation, while the current operation is still in progress.

Removing the dependency on the PYNQ framework

In the current implementation, the only dependency on the PYNQ framework is the `PynqBuffer` object[2]. The other parts, including the bitstream management, driver creation and interfacing with the hardware, are completely customised to be able to achieve a maximal speed-up. In fact, it could be possible to completely separate from the framework by directly interfacing with the low-level CMA drivers from within the `ZyPy` module. This also means that ZyPy and PYNQ cannot be used interchangeably.

[2] Apart from the initialisation of the board.

Bibliography

[1] S. Prongnuch and T. Wiangtong, "Heterogeneous Computing Platform for data processing," in *2016 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*, 2016, pp. 1–4. [Online]. Available: https://doi.org/10.1109/ISPACS.2016.7824762 (Accessed 2021-05-29).

[2] J. Lee, G. Jo, W. Jung, H. Kim, J. Kim, Y.-J. Lee, and J. Park, "Chapter 2 - SnuCL: A unified OpenCL framework for heterogeneous clusters," in *Advances in GPU Research and Practice*, ser. Emerging Trends in Computer Science and Applied Computing, H. Sarbazi-Azad, Ed. Boston: Morgan Kaufmann, 2017, pp. 23–55. [Online]. Available: https://doi.org/10.1016/B978-0-12-803738-6.00002-1 (Accessed 2021-05-29).

[3] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor System-on-Chip (MPSoC) Technology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1701–1713, 2008. [Online]. Available: https://doi.org/10.1109/TCAD.2008.923415 (Accessed 2021-05-29).

[4] Xilinx. (2021) Zynq-7000 SoC. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html (Accessed 2021-05-03).

[5] L. Stornaiuolo, F. Carloni, R. Pressiani, G. Natale, M. Santambrogio, and D. Sciuto, "Enabling transparent hardware acceleration on Zynq SoC for scientific computing," *ACM SIGBED Review*, vol. 17, pp. 30–35, 07 2020. [Online]. Available: https://doi.org/10.1145/3412821.3412826 (Accessed 2021-05-29).

[6] Xilinx. (2021) Xilinx Runtime. [Online]. Available: https://github.com/Xilinx/XRT (Accessed 2021-05-29).

[7] Amazon. (2021) Amazon EC2 F1 Instances. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/ (Accessed 2021-05-29).

[8] Xilinx. (2021) Xilinx Alveo: Adaptable Accelerator Cards for Data Center Workloads. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/alveo.html (Accessed 2021-05-29).

[9] Python Software Foundation. (2021) Python. [Online]. Available: https://www.python.org/ (Accessed 2021-05-03).

[10] Python Software Foundation. (2021) PyPI. [Online]. Available: https://pypi.org/ (Accessed 2021-05-29).

[11] Xilinx. (2021) Python productivity for Zynq (PYNQ). [Online]. Available: https://pynq.readthedocs.io/en/latest/ (Accessed 2021-05-29).

[12] Xilinx. (2021) PYNQ community projects. [Online]. Available: http://www.pynq.io/embedded.html (Accessed 2021-05-31).

[13] NumPy. (2021) NumPy. [Online]. Available: https://numpy.org/ (Accessed 2021-05-03).

[14] SciPy. (2021) SciPy. [Online]. Available: https://www.scipy.org/ (Accessed 2021-05-03).

[15] C. Maxfield, *FPGAs: Instant Access*, ser. Instant Access. Elsevier Science, 2011. [Online]. Available: https://books.google.be/books?id=LwULo6PpvdwC (Accessed 2021-06-06).

[16] Xilinx. (2021) SDAccel Development Environment Help: What is an FPGA? [Online]. Available: https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/gac1504034293050.html (Accessed 2021-05-03).

[17] D. Drake and J. Berg. (2021) Unaligned Memory Accesses. [Online]. Available: https://www.kernel.org/doc/Documentation/unaligned-memory-access.txt (Accessed 2021-07-27).

[18] Xilinx Inc. SDAccel Development Environment Help for 2019.1: HLS Pragmas. [Online]. Available: https://www.xilinx.com/html_docs/xilinx2019_1/sdaccel_doc/hls-pragmas-okr1504034364623.html (Accessed 2021-05-11).

[19] A. G. Schmidt, G. Weisz, and M. French, "Evaluating Rapid Application Development with Python for Heterogeneous Processor-Based FPGAs," *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 121–124, 2017. [Online]. Available: https://doi.org/10.1109/FCCM.2017.45 (Accessed 2021-05-29).

[20] B. Janßn, P. Zimprich, and M. Hübner, "A dynamic partial reconfigurable overlay concept for PYNQ," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4. [Online]. Available: https://doi.org/10.23919/FPL.2017.8056786 (Accessed 2021-05-11).

[21] J. Decaluwe, "MyHDL: a Python-based hardware description language," *Linux Journal*, vol. 2004, p. 5, 2004. [Online]. Available: https://api.semanticscholar.org/CorpusID:195996891 (Accessed 2021-05-29).

[22] P. Haglund, O. Mencer, W. Luk, and B. Tai, "Hardware Design with a Scripting Language," vol. 2778, 09 2003, pp. 1040–1043. [Online]. Available: https://doi.org/10.1007/978-3-540-45234-8_115 (Accessed 2021-05-29).

[23] D. Lockhart, G. Zibrat, and C. Batten, "PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research," *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 280–292, 2014. [Online]. Available: https://doi.org/10.1109/MICRO.2014.50 (Accessed 2021-05-29).

[24] E. Logaras, O. G. Hazapis, and E. S. Manolakos, "Python to Accelerate Embedded SoC Design: A Case Study for Systems Biology," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4, March 2014. [Online]. Available: https://doi.org/10.1145/2560032 (Accessed 2021-05-29).

[25] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood, "A pythonic approach for rapid hardware prototyping and instrumentation," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–7. [Online]. Available: https://doi.org/10.23919/FPL.2017.8056860 (Accessed 2021-08-03).

[26] R. Peschke, K. Nishimura, and G. Varner, "ARGG-HDL: A High Level Python Based Object-Oriented HDL Framework," 2020. [Online]. Available: https://arxiv.org/abs/2011.02626 (Accessed 2021-08-02).

[27] D. H. Noronha, B. Salehpour, and S. J. E. Wilton, "LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks," 2018, (Accessed 2021-08-02).

[28] S. Skalicky, J. Monson, A. Schmidt, and M. French, "Hot & Spicy: Improving Productivity with Python and HLS for FPGAs," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 85–92. [Online]. Available: https://doi.org/10.1109/FCCM.2018.00022 (Accessed 2021-08-03).

[29] Y. Uguen and E. Petit, "PyGA: a Python to FPGA compiler prototype," *Proceedings of the 5th ACM SIGPLAN International Workshop on Artificial Intelligence and Empirical Methods for Software Engineering and Parallel Computing Systems*, 2018. [Online]. Available: https://doi.org/10.1145/3281070.3281072 (Accessed 2021-06-06).

[30] S. Huang, K. Wu, H. Jeong, C. Wang, D. Chen, and W.-m. Hwu, "PyLog: An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 227–228. [Online]. Available: https://doi.org/10.1145/3431920.3439478 (Accessed 2021-08-02).

[31] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012, (Accessed 2021-08-03).

[32] R. Okuta, Y. Unno, D. Nishino, S. Hido, and Crissman, "CuPy : A NumPy-Compatible Library for NVIDIA GPU Calculations," 2017. [Online]. Available: https://www.semanticscholar.org/paper/CuPy-%3A-A-NumPy-Compatible-Library-for-NVIDIA-GPU-Okuta-Unno/a59da4639436f582e483347a4833e7659fd3e598 (Accessed 2021-05-06).

[33] T. Higuchi, N. Yoshifuji, T. Sakai, Y. Kitta, R. Takano, T. Ikegami, and K. Taura, "ClPy: A NumPy-Compatible Library Accelerated with OpenCL," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 933–940. [Online]. Available: https://doi.org/10.1109/IPDPSW.2019.00159 (Accessed 2021-05-06).

[34] M. Kristensen, S. Lund, T. Blum, K. Skovhede, and B. Vinter, "Bohrium: Unmodified NumPy Code on CPU, GPU and Cluster," 11 2013. [Online]. Available: https://www.researchgate.net/publication/281647460_Bohrium_Unmodified_NumPy_Code_on_CPU_GPU_and_Cluster (Accessed 2021-05-29).

[35] T. Blum, M. R. Kristensen, and B. Vinter, "Transparent GPU Execution of NumPy Applications," *2014 IEEE International Parallel and Distributed Processing Symposium Workshops*, pp. 1002–1010, 2014. [Online]. Available: https://doi.org/10.1109/IPDPSW.2014.114 (Accessed 2021-05-29).

[36] M. Kristensen, S. Lund, T. Blum, and K. Skovhede, "Separating NumPy API from Implementation," in *PyHPC 2014 : 4th Workshop on Python for High Performance and Scientific Computing*, 11 2014. [Online]. Available: https://www.researchgate.net/publication/281647570_Separating_NumPy_API_from_Implementation (Accessed 2021-05-10).

[37] Stornaiuolo, Luca, "Exploiting FPGA from Data Science Programming Languages," Master's thesis, Politecnico di Milano, 2017, (Accessed 2021-05-03).

[38] A. Ismail and L. Shannon, "FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators," 06 2011, pp. 170 – 177. [Online]. Available: https://doi.org/10.1109/FCCM.2011.48 (Accessed 2021-08-03).

[39] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, and C. Plessl, "ReconOS: An Operating System Approach for Reconfigurable Computing," *Micro, IEEE*, vol. 34, pp. 60–71, 01 2014. [Online]. Available: https://doi.org/10.1109/MM.2013.110 (Accessed 2021-08-03).

[40] W. Kritikos, A. Schmidt, R. Sass, E. Anderson, and M. French, "Redsharc: A Programming Model and On-Chip Network for Multi-Core Systems on a Programmable Chip," *International Journal of Reconfigurable Computing*, vol. 2012, 01 2012. [Online]. Available: https://doi.org/10.1155/2012/872610 (Accessed 2021-07-23).

[41] V. Kathail, "Xilinx Vitis Unified Software Platform," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 173–174. [Online]. Available: https://doi.org/10.1145/3373087.3375887 (Accessed 2021-07-23).

[42] Xilinx, *Zynq-7000 SoC Technical Reference Manual (UG585)*, Xilinx. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf (Accessed 2021-05-06).

[43] Xilinx. (2021) Defining Interfaces. [Online]. Available: https://www.xilinx.com/html_docs/xilinx2021_1/vitis_doc/managing_interface_synthesis.html (Accessed 2021-07-27).

[44] Xilinx. (2021) Vitis Unified Software Development Platform 2021.1 Documentation. [Online]. Available: https://www.xilinx.com/html_docs/xilinx2021_1/vitis_doc/hls_pragmas.html (Accessed 2021-08-03).

[45] Xilinx. (2021) Vitis High-Level Synthesis User Guide. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf (Accessed 2021-08-03).

[46] Xilinx. (2018) Creating a New Zynq FSBL Application Project. [Online]. Available: https://www.xilinx.com/html_docs/xilinx2018_1/SDK_Doc/SDK_tasks/task_creatinganewzynqfsblapplicationproject.html (Accessed 2021-08-03).

[47] Xilinx. (2018) Board Support Packages (SDK). [Online]. Available: https://www.xilinx.com/html_docs/xilinx2018_1/SDK_Doc/SDK_concepts/sdk_c_bsp_internal.html (Accessed 2021-08-03).

[48] Hans-Jürgen Koch. (2018) The Userspace I/O HOWTO. [Online]. Available: https://www.kernel.org/doc/html/v4.12/driver-api/uio-howto.html (Accessed 2021-08-11).

[49] man7. (2015) Linux Programmer's Manual - mem(4). [Online]. Available: https://man7.org/linux/man-pages/man4/mem.4.html (Accessed 2021-08-11).

[50] man7. (2021) Linux Programmer's Manual - mmap(2). [Online]. Available: https://man7.org/linux/man-pages/man4/mem.4.html (Accessed 2021-08-11).

[51] A. Rigo and M. Fijalkowski. (2021) CFFI documentation. [Online]. Available: https://cffi.readthedocs.io/en/latest/index.html (Accessed 2021-08-14).

[52] J. Hunt, "Monkey Patching and Attribute Lookup," in *A Beginners Guide to Python 3 Programming*. Springer, 2019, pp. 325–336. [Online]. Available: https://link.springer.com/book/10.1007/978-3-030-20290-3 (Accessed 2021-08-14).

[53] Python Software Foundation. (2021) Abstract Syntax Trees. [Online]. Available: https://docs.python.org/3/library/ast.html (Accessed 2021-08-03).

[54] Xilinx. (2021) Dynamic Function eXchange (DFX). [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/implementation/dynamic-function-exchange.html (Accessed 2021-05-21).

[55] M. Kadi, P. Rudolph, D. Gohringer, and M. Hübner, "Dynamic and partial reconfiguration of Zynq 7000 under Linux," 12 2013, pp. 1–5. [Online]. Available: https://doi.org/10.1109/ReConFig.2013.6732279 (Accessed 2021-05-21).