FIDA: a framework to automatically integrate FPGA kernels within Data-Science applications

L. Stornaiuolo, A. Parravicini, D. Sciuto, M. D. Santambrogio
Politecnico di Milano, Milan, Italy
alberto.parravicini@mail.polimi.it,
{luca.stornaiuolo, donatella.sciuto, marco.santambrogio}@polimi.it

***Abstract*—Hardware accelerators are an effective solution to increase the performance of algorithms in a wide array of disciplines, from data science to computational finance. However, data scientists and mathematicians often do not have the required knowledge or time to fully exploit these accelerators, and they perceive them as difficult and frustrating to use. OpenCL was created to simplify the creation of computational pipelines with heterogeneous hardware, but as of today its integration with high level languages commonly used in data science is limited. In this paper, we propose a framework to integrate OpenCL kernels running on Field Programmable Gate Arrays (FPGAs) with Python, R and MATLAB, the most common languages used in data science. Our framework can automatically generate all the interfaces needed to wrap an OpenCL kernel into these high level languages, and provide the user with a transparent access to the kernel itself.**

I. Introduction

Heterogeneous System Architectures (HSAs) are being seen with more and more interest, as the computational pipelines used in fields such as data science and computational finance demand high performance, low latency and low power consumption.

To meet these requirements, many different tools and hardware solutions have been proposed by academia and industry[1]. Among the available solutions, FPGAs can be configured to provide high parallelism and throughput, but can be difficult and cumbersome to program. FPGA vendors such as Xilinx have been trying to lower the barrier to access to hardware acceleration. These companies provide tools that allow the user to accelerate their algorithms by writing computational kernels in C, and make available libraries of kernels that have already been built for their systems.

As an example, Xilinx SDx simplifies FPGA programming by providing PCI-Express (PCIe) drivers and an OpenCL runtime that the host-side software can leverage to control the hardware accelerator. However, fully integrating FPGA kernels in the computational pipelines can be a tedious and error-prone task, due to the large number of steps required to connect these kernels to high level languages such as Python, R and MATLAB which are popular amongst data scientists.

Starting from our previous work [2], we propose a framework that aims at fully automatizing the integration of FPGA kernels into Python, R and MATLAB, starting from a simple description of the inputs and outputs of the kernel.

Compared to our previous results, we leverage Xilinx SDx to produce the PCIe drivers required to connect the FPGA to the host system, and we extend the framework to include Python and MATLAB.

More in detail, we present the following contributions:

- The automatic generation of the OpenCL host file that controls an FPGA kernel to be integrated (Section III).
- The automatic generation of the interfaces that transparently connect the OpenCL host to Python, R and MATLAB.
- How to leverage our framework in order to minimize the computational overhead in the overall pipeline (Section IV).

II. Related work

The fields of data science and computational finance are characterized by strict performance requirements. It is often demanded to minimize the execution times and at the same time to provide consistent latency. The innate parallelism of the computational tasks performed in these fields guided the users towards architectures such as Graphic Processing Unit (GPU) and FPGA.

To easily leverage these architectures, different frameworks have been proposed. GPUs have been widely adopted in scientific computing since the introduction of CUDA by NVIDIA [3]. Thanks to their performance in data parallel tasks, GPUs have been quickly integrated in libraries and programming languages in a way that is fully transparent to the users. As an example, MATLAB seamlessly supports GPU acceleration by using *gpuArrays()*[4], and Python offers similar capabilities using PyCUDA[5].

FPGA vendors have been trying to provide a similar level of abstraction, by exploiting the OpenCL framework in the programming of computational kernels and of the host software [6]. Moreover, there exist already wrappers of OpenCL written for Python [7], R [8] and even MATLAB [9]. All these wrappers, however, have a different syntax, and still rely on the user to write a large amount of boilerplate code to control the accelerator.

Other recent work have been focused on the integration of Domain Specific Languages (DSLs) with FPGAs [10], by building a common backend which enables the DSL compiler to target FPGA architectures. This approach is different from ours as it operates directly on the Abstract Syntax Tree of

the code, instead of wrapping an existing OpenCL kernel. The advantage of our framework is that it can be applied to different architectures with minimal effort, and it is not inherently limited to FPGAs.

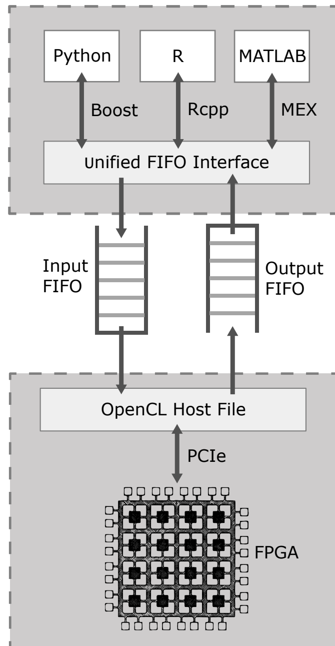

Fig. 1. Scheme of the proposed framework. The upper box represents the high level language module, while the lower box represents the OpenCL/FPGA module. The two modules are connected by using FIFO buffers.

III. Proposed approach

The main goal of our framework is to enable a user to automatically generate the interfaces of an OpenCL kernel to the most common high level languages used in data science. We focused on Python, R and MATLAB, which are often the languages of choice of data scientist, quantitative analysts and computational statisticians [11].

From a description of the inputs and outputs of the kernel (Listing 1), in terms of structure (scalar or array) and type (int, float, etc...), our framework is able to automatically generate all the interfaces required to connect the kernel to the high level languages.

We focused on FPGA C/C++ kernels written following the OpenCL paradigm, optimized through Vivado HLS, and compiled with the OpenCL compiler provided within the Xilinx SDAccel workflow. The framework also works with kernels written in Hardware Description Language (HDL) like Verilog and VHDL, or with the Xilinx OpenCL libraries.

Moreover, the framework can wrap kernels that behave as black-boxes, and for which only the input/output specifics are known (which is usually the case for kernels that are sold commercially). A strong point of our work is that it can be used to interface kernels that run either on CPUs, GPUs and FPGAs making it highly flexible to many different HSA systems.

Listing 1. Example of kernel description

```
{
  "kernel_name": "mmult",
  "board": ["xilinx_adm-pcie-7v3_1ddr_3_0"],
  "xclbin": ["kernel_7v3.xclbin"],
  "num_iterations": 3,
  "inputs": [

      {"type": "array",
      "name": "a",
      "length": 256,
      "class": "int",
      "position": 0},

      {"type": "array",
      "name": "b",
      "length":256,
      "class": "int",
      "position": 1}],

  "outputs": [

      {"type": "array",
      "name": "c",
      "length":256,
      "class": "int",
      "position": 2}]
}
```

The structure of the framework can be divided in two main modules (Figure 1). The first module represents the OpenCL host interface, the software that controls the accelerator and the kernel itself. The second module is used to connect the high level languages to the host interface, by using language specific libraries and tools. The two modules are connected by a streaming interface based on named pipes, created in linux through `mkfifo`.

A. Low-Level Module

The low-level module of our framework is composed by the computational kernel which is wrapped in the user application, and by an OpenCL host file that manages the kernel. The kernel is compiled through an OpenCL compiler, and all the platforms supported by the OpenCL standard can be used. The input/output specifics of the kernel are described with a small configuration file, that is used by the framework to build the interfaces.

The OpenCL runtime provided by the host file configures and manages the FPGA, and launches the kernel whenever the required data are available. The host file is unique and independent from the high level language chosen by the user, and it requires recompilation only if the target accelerator or the target kernel are changed.

B. High-Level Module

The high-level module gives the user the ability to call the OpenCL kernel directly from Python, R or MATLAB, without having to manually configure the FPGA or handling the data transfer. The upper portion of the module is used to convert the data of a given high-level languages to the types and data structures that can be processed by OpenCL and by the

FPGA kernel. This is accomplished by making use of different libraries, depending on the language that is considered.

In the case of Python, we make use of the Boost.Python[12] library, which allows to wrap C/C++ function and classes in modules that can be imported and invoked from Python. The structure of the module itself is independent from the target kernel, but arrays have to be converted from `boost::python::list` to standard C arrays before being sent to the OpenCL host.

R allows to compile and execute C++ functions through the Rcpp package [13], and invoke them like traditional R functions. As in the case of Python, R data-types must be converted to regular C/C++ types (e.g. arrays of integers become IntegerVectors), both when sending and receiving data from the FPGA. The conversion is handled by R's C interface [14], which casts the subtypes of defined R data type to default C++ types.

In MATLAB, the interface with C is implemented through MEX files [15]. MEX files are dynamically linked subroutines executed by MATLAB as if they were built-in functions. MATLAB is optimized to work on floating point numbers and doesn't offer full support to integer numbers. However, we can cast floating point numbers to integers if the OpenCL kernel demands so.

These modules converts the data to the appropriate data types, and then call a language-independent function that is connected to the OpenCL by the named pipes. This function will send and receive the data through the named pipes.

The two modules are connected by using named pipes. Our framework uses one input and one output pipe, which are created and managed by the OpenCL host. After completing the FPGA reconfiguration, the host waits for data to be sent by an application on the input pipe, and will return the results on the output pipe. If desired, the user can require the host to run in a server-like mode, meaning that the host will remain active after having processed the data, so that new data can be sent and processed. This optimization allows to mask the FPGA reconfiguration time, and drastically reduce the execution time overheads for a kernel that is repeatedly invoked by the user application.

IV. Experimental evaluation

To analyze the impact of our framework on the performance of a computational pipeline, we have conducted several tests. Our focus was mainly on measuring the overheads of the data-type casting and data transfer that are introduced by our framework, and to understand which part of the interface has the highest impact. The overall execution time of an OpenCL kernel wrapped with our framework can be decomposed into several steps, for which we have measured the relative impact. More specifically, we considered the time required to program the FPGA, the time required to convert the input (and output) from the data-types used by the high level languages to the types used by OpenCL, the time used to send the data through the named pipes, the time used to transfer the data from the host machine memory to the FPGA memory, and finally the time taken by the FPGA computation itself.

We have consider two simple OpenCL kernels which are used in numerous disciplines and which can benefit from FPGA acceleration.

A. Integer Matrix Multiplication

We implemented a 16×16 integer matrix product, in which the matrices are sent to the kernel as one-dimensional buffers and multiplied by taking advantage of hardware pipelining. The input matrices are sent to the kernel through the input named pipe, and the output matrix is retrieved through the output named pipe.

B. Variance

The variance of an input signal is computed in an extremely efficient way by the FPGA, through hardware pipelining and tree reductions to perform multiply-and-accumulate operations in a single clock cycle. By also introducing parallel reads from the input, the main computation can be performed in a number of cycles lower than the input size, not counting the cycles needed to transfer the signal to the FPGA memory. The input signal was a 10000-long float vector.

We have tested the framework using a Xilinx Virtex-7 FPGA connected through PCIe. The FPGA was mounted on a host machine which contains an Intel i7 870 CPU at 2.93GHz and 8GB of RAM. The host machine was also used to compare the execution times of the FPGA with respect to using the CPU exclusively.

We measured the execution times of both kernels using the interfaces in three high level languages that we support, and compared our results with the execution time obtained by the respective built-in functions.

It is possible to see (Figure 2) that the R interface performs sightly faster, while Python and MATLAB have similar execution times. The overhead, due to data-types casting and input/output transfer plays a significant cost in the overall execution time.

To get a better understanding of the overhead of each step in the pipeline we decomposed the overall execution time into the individual steps that are present in the framework (Figure 3). We measure the time from the start of the input transfer to the end of the output transfer, thus including the overhead added by our interface. As the board can be programmed before starting the computation and kept running waiting for data to be processed, we measured the executions times without considering the FPGA reconfiguration time. This analysis was done for both of the OpenCL kernel we tested, as they have different input types and sizes. We can see how transferring data from the application to the host using named pipes has a significant time cost, while the actual kernel execution is extremely fast. As a consequence, it might preferable to accelerate kernels in which the computation plays a significant cost, compared to the data transfer.

V. Conclusions

In this paper we have proposed a framework to facilitate the integration of OpenCL kernels into computational pipelines

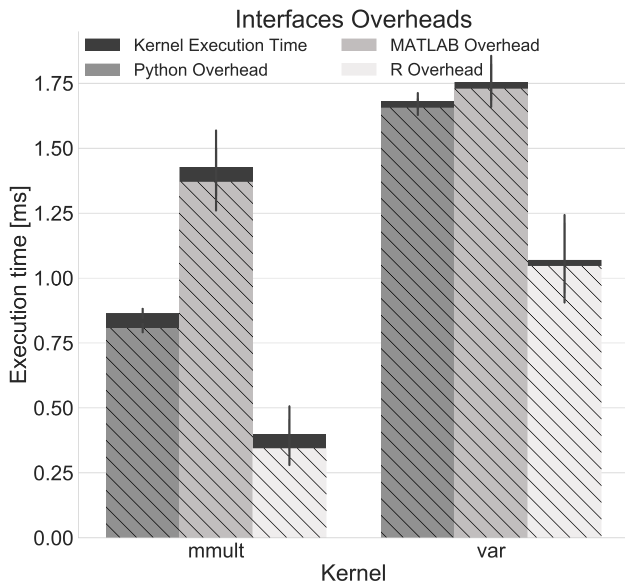

Fig. 2. Execution time of the mmult and variance kernels, with highlighted the overheads of the interfaces in each supported language.

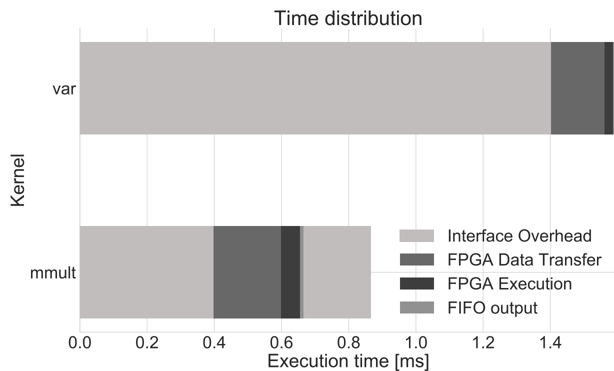

Fig. 3. Time subdivision across the different steps of the computation. The times considered are taken from the Python interface.

written in common high level languages such as Python, R and MATLAB, by providing the automatic creation of interfaces that take care of data conversion, data transfer and of the runtime management of the kernel. At the moment the overhead introduced by our framework prevent the acceleration of kernels that do not have a significant execution time, but this issue could be removed by using shared memory to transfer data between the user application and the host.

We believe that our work can trivially be extended to support GPUs and other hardware accelerators, to provide even higher flexibility to the users. Moreover, we would like to deploy our solution on the AmazonWeb Services (AWS) EC2 F1 compute instances, which support high-end FPGAs and would allow the users to accelerate their computation without the need to own and install an FPGA.

Another extension that we plan to implement is to support OpenCL hosts that can run multiple kernels in the same pipeline, as long as the board does have enough resources, in order to reduce the reconfiguration time overheads.

REFERENCES

[1] K. H. Tsoi and W. Luk, "Axel: a heterogeneous cluster with fpgas and gpus," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2010, pp. 115–124.

[2] L. Stornaiuolo, A. Parravicini, G. Durelli, and M. Santambrogio, "Exploiting fpgas from higher level languages a signal analysis case study," in *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*. IEEE, 2017, pp. 132–140.

[3] NVIDIA, "CUDA Parallel Computing Platform." [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[4] MathWorks, "Parallel Computing Toolbox." [Online]. Available: http://mathworks.com/products/parallel-computing/

[5] Nvidia, "PyCUDA." [Online]. Available: https://developer.nvidia.com/pycuda

[6] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[7] A. Klöckner, "Pycuda: Even simpler gpu programming with python."

[8] S. Urbanek, "Opencl: Interface allowing r to use opencl."

[9] J. Radford, "Opencl toolbox v0.17." [Online]. Available: https://it.mathworks.com/matlabcentral/fileexchange/30109-opencl-toolbox-v0-17

[10] E. Del Sozzo, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio, "A common backend for hardware acceleration on fpga," in *2017 IEEE 35th International Conference on Computer Design (ICCD)*. IEEE, 2017, pp. 427–430.

[11] J.-F. Puget, "The most popular language for machine learning and data science is . . . ." [Online]. Available: https://www.kdnuggets.com/2017/01/most-popular-language-machine-learning-data-science.html

[12] D. Abrahams and R. W. Grosse-Kunstleve, "Building hybrid systems with boost. python," *CC Plus Plus Users Journal*, vol. 21, no. 7, pp. 29–36, 2003.

[13] C. R-project, "Rcpp: Seamless R and C++ Integration." [Online]. Available: https://cran.r-project.org/web/packages/Rcpp/index.html

[14] A. R. by Hadley Wickham, "R's C interface." [Online]. Available: http://adv-r.had.co.nz/C-interface.html

[15] MathWorks, "Mex file creation api." [Online]. Available: https://it.mathworks.com/help/matlab/call-mex-files-1.html