2016 IEEE International Parallel and Distributed Processing Symposium Workshops

On How to Improve FPGA-Based Systems Design Productivity via SDAccel

Giulia Guidi[1], Enrico Reggiani[1], Lorenzo Di Tucci[1],
Gianluca Durelli[1], Michaela Blott[2], Marco D. Santambrogio[1]

[1]Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Milano, Italy,
{*enrico2.reggiani, lorenzo.ditucci, giulia.guidi*}@mail.polimi.it
{*gianlucacarlo.durelli, marco.santambrogio*}@polimi.it
[2]Xilinx Research, Dublin, IE
michaela.blott@xilinx.com

***Abstract*—Custom hardware accelerators are widely used to improve the performance of software applications in terms of execution times and to reduce energy consumption. However the realization of an hardware accelerator and its integration in the final system is a difficult and error prone task. For this reason, both Industry and Academy are continuously developing Computer Aided Design (CAD) tools to assist the designer in the development process. Even if many of the steps have been nowadays automated, system integration and SW/HW interfaces definition and drivers generation are still almost completely manual tasks. The last tool released by Xilinx , however, aims at improving the hardware design experience by leveraging the OpenCL standard to enhance the overall productivity and to enable code portability. This paper provides an overview of the SDAccel potentiality comparing its design flow with other methodologies using two case studies from the Bioinformatics field: brain network and protein folding analysis.**

I. Introduction

As the performance requirements of computing systems are continuously increasing at all levels from High Performance Computing (HPC) environments to mobile and embedded systems, it is becoming more and more clear that a SW only solution is not a valid approach anymore. Performance of the applications are impacted by the large amount of data that have to be processed which in turn has a great negative impact on the amount of energy needed to perform the computation [1].

In this scenario HW accelerators have proven to be effective in optimizing performance/Watt figure of certain kind of applications [2]. However the benefits provided by HW accelerators are highly impaired by the complexity of the process that has to be carried out to design the accelerators themselves. The high complexity of HW design flow is well known and, over the years, both Industry and Academy have produced a wide range of CAD tools to assist the designer in the development of HW accelerators [3]–[5]; these tools helped in simplifying and automating most of the development phases. A clear instance of this simplification relies on the fact that for the designer is not strictly required anymore to manually write a Hardware Description Language (HDL) implementation of the HW core. Nowadays in fact the designer can leverage High Level Synthesis (HLS) tools [4], [6] to translate a high level code (generally C/C++) to HDL.

The introduction of CAD tools allowed either the complete automation of certain design steps, such as HDL translation with HLS tools and application partitioning using Design Space Exploration (DSE) frameworks (mainly explored in academia), or the simplification of the design process by the assistance of a Graphical User Interface (GUI). However, even with the usage of these tools, the designer is still required to perform manual integration of the HW cores in the final architecture and most importantly he is required to define all the HW/SW interfaces for the HW cores, program and implement the drivers and then update the application to exploit the HW accelerators.

The latest tool from Xilinx, SDAccel, aims at alleviating this integration burden at both SW and HW level by leveraging on the OpenCL [7] standard to target Field Programmable Gate Array (FPGA) boards.

This paper will compare the design flow provided by the new SDAccel tool against classical methodologies to target FPGA devices and will illustrate two case study, from bioinformatics field, where we used SDAccel to generate a HW implementation of these applications.

The remainder of the paper illustrates the design flow of SDAccel in Section II; while Section III compares SDAccel with the other tools available to target FPGA, while Section IV introduces the two example applications, and Section V illustrates how we used SDAccel to generate a preliminary prototypes for the two case studies. Finally Section VII concludes the paper and summarizes our findings.

II. SDAccel

Xilinx SDAccel is a Software-Defined Development Environment (Fig. 1) for targeting FPGA devices starting from either OpenCL, C or C++ code. The SW side is based on the OpenCL runtime and APIs; this allows SDAccel to target FPGAs exploiting the classic design flow used to target other accelerators such as Graphic Processing Units (GPUs). SDAccel leverages on the Vivado Design Suite tool from Xilinx exploiting Vivado HLS tool for the HLS step, extending it with the capability of supporting OpenCL code.

The introduction of OpenCL as starting point allows an easier programming environment to those familiar with OpenCL

978-1-5090-3682-0/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPSW.2016.171

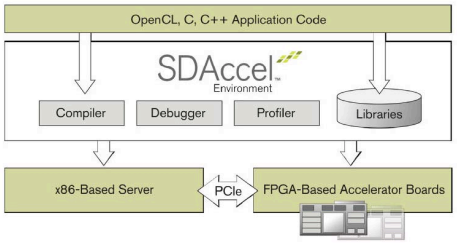

Fig. 1. SDAccel software environment overview.

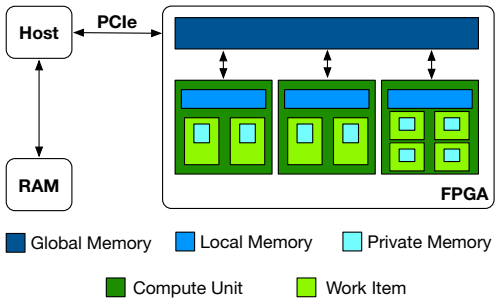

Fig. 2. SDAccel architecture and memory hierarchy.

extension and used to the memory architecture defined by OpenCL. The final architecture produced by SDAccel is fully compliant with the OpenCL standard and it is composed as described in Fig. 2. In particular SDAccel has a fundamental difference with respect to the standard OpenCL programming environment, in fact its architecture can be customized as it's not fixed. The user can customize not only the kernels executed by the device, but also the overall architecture structure and define how many parallel computational elements are present in order to fully customize its design.

In Fig. 2 the memory related blocks are highlighted in blue, while the compute parts are highlighted in green. If we look at the compute part we see that the FPGA resources are divided among, so called, *compute units*. Each compute unit implements the code to execute one kernel, but the same kernel might be associated to multiple compute units. This one-to-many relation translates into the parallel execution of the kernel code on both the compute units at runtime. Furthermore each compute unit can be customized to have a user-defined number of *work items*, which are the element effectively carrying out the kernel execution; also in this case the *work units* execute in parallel inside the compute unit.

Concerning the memory hierarchy, SDAccel is fully compliant with OpenCL specification and it has a *global memory* accessible from both the Host and the FPGA, *local memories* per *compute unit* shared by all the *work items* inside, and *private memories* per each *work item*.

A. SDAccel Design Flow

The design flow of SDAccel with all its steps is represented in Fig. 3. At first, in order to exploit the potentiality of the tool, the application designer must profile its application and

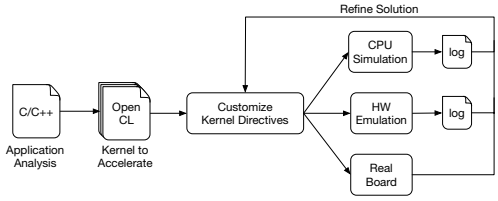

Fig. 3. Design flow using SDAccel .

identify which functions are the bottleneck of the computation and whether or not these functions are well suited for an FPGA implementation. After finding the kernel (or set of kernels) to implement on the FPGA, the designer has to provide an implementation of the Kernel in one of the languages supported by the tool (OpenCL or Vivado HLS). At this point the user can start to take advantage of the potentiality of the tool and to port the interesting kernels onto the FPGA device. Either via the GUI or the scripting language (in *TCL*), the user can specify how many instances of the kernel (i.e. *compute units*) have to be instantiated; while using OpenCL directives the user can customize the number of *work items* inside each *compute unit*. After defining the structure of the system, with a simple button, or with a *TCL* command the user can perform a CPU-based simulation of the system which reports useful information on how the final hardware behaves listing the utilization for each *compute units* and information about memory transfers. After CPU simulation, the tool allows also to perform an HW emulation based on SystemC [8] to ensure a more accurate profiling results and to test if the OpenCL directives used and the structure of the OpenCL kernel are suitable for the final device. Finally the user can generate a final package comprising the application code and the bitstream to be used to target the real board.

At each time the user can test its design and modify the OpenCL code or the compiler directives to refine its solution.

III. Comparison with Classical Methodology

In this section we want to compare the SDAccel flow presented in Section II with the classical methodology used to target FPGA-based systems.

The overall classic flow for targeting FPGA devices is reported in Fig. 4; please note that only the steps done by the designer are expressed in the image, while automatic steps performed by the tools (e.g. synthesis, map, place, and route) are all collapsed into the *bitstream generation* block.

The design flow always start from the identification of the functionalities that have to be ported onto the FPGA. At this point the problem of creating a description of the functionalities that can be understood by vendors specific backend tools arises, since these are generally expressed in a high-level language. However vendor tools targeting FPGA accept as input HW cores described in a HDL (e.g. VHDL [9] or Verilog [10]) or lower level descriptions in form of a netlist. The designer will then face the first challenge of translating the application code into HDL; he has two option to carry out this translation either by performing the translation man-

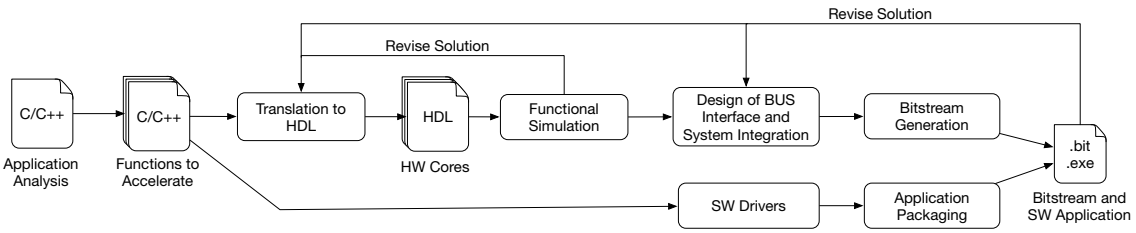

Fig. 4. Classical flow for FPGA devices

ually writing HDL code, or relying on automatic translation provided by HLS tools.

The exit of this design phase usually happens when the HDL code is functionally validated via a simulator; however generally this validation happens only for the computational part, so other bugs might arise at the interface level. The manual translation to HDL is error prone and requires a very high expertise; on the other hand it allows to have a full control on the actual core implementation. Automated translation performed via HLS, on the other hand, relieves most of the burden of the HDL translation since this happens automatically starting from high level code (e.g. C/C++). Furthermore HLS tools, such as VivadoHLS, allow to generate not only the compute part of the core, but provides also methods to automatically generate and validate the core's interfaces. Errors in this design step, or not satisfactory performance, forces the user to restart the revision of HDL translation.

After the translation of all the HW cores, these have to be extended with a HW interface to be connected to the system BUS and then they have to be integrated in the final architecture. At this level all the decisions regarding the architecture structure have to be taken, providing higher flexibility in the design, with the drawback of requiring high expertise and the possibility of introducing errors in the design.

Finally the bitstream can be generated using completely automated vendors tools. However, obtaining the bitstream is not the last step of the classical design flow. The starting application need to be adapted to invoke the HW cores, instead of the SW version.

A. Comparison with SDAccel

Porting applications to FPGA is time consuming and error prone. It is observable from Fig.3 and 4, that SDAccel presents a lower numbers of steps, increasing the automation and lowering the possibility of producing errors. SDAccel automatically generates both HW and SW components for the designer. After the automatic generation of the bitstream, the FPGA can be configured and the HW cores can be used leveraging the OpenCL standard APIs for both kernel invocations and data transfer. Obviously this degree of automation comes at the price of a lower possibility to customize the architecture, which means that the usage of SDAccel flow might be beneficial only for certain types of application that can be implemented well by exploiting the generated architecture.

IV. Case Study: Preliminaries

After the comparison of the design flows at a very high level, we want to show how we used SDAccel to port two applications from the Bioinformatics field to FPGA. This section introduces preliminary knowledge on the application analysis performed to determine possible performances of the application and introduces the two case studies.

A. Roofline Model

The Roofline model [11] exploits an approach named *bound and bottleneck analysis* and shows potential benefit and priority of optimizations. The model provides valuable insight into the primary factors affecting the performance of computer systems. In particular, the critical influence of the system bottleneck is highlighted and quantified. The Roofline model in fact relates processor performance to off-chip memory traffic. The term *operational intensity* identifies operations per byte of DRAM traffic, defining total bytes accessed as those bytes that end up to the main memory after they have been filtered by the cache hierarchy. In other words, it measures traffic between the caches and memory. The Roofline model ties floating-point performance, operational intensity, and memory performance together into a single readily understandable 2D log-log scale graph, as the one in Fig. 5. In the figure the y-axis represents attainable floating-point performance, while the x-axis is operational intensity, varying from 0.25 Flops/DRAM byte-accessed to 16 Flops/DRAM byte-accessed. Performance is upper bounded by both the peak FLOPS rate and the peak memory bandwidth, which define the line in the plot. The actual FLOPS of a floating-point kernel depends on the operational intensity and the line determines the limit of the actual performance: the nature of the computation is memory bound if the FLOPS are limited by the memory bandwidth (left part of the line), while it is compute bound if they are limited by the hardware performance limit (right part of the line). With the aim of plotting peak memory performance, the x-axis is Flops per Byte and the y-axis is GFlops/sec, so gigabytes per second (GB/sec) — or (GFlops/sec)(Flops/Byte) — is just a line of unit slope in Figure 5. Hence, it is possible plot a second line that bounds themaximum floating-point performance that the memory system of the computer can support for a given operational intensity. The two lines intersect at the point of peak computational performance and peak memory bandwidth. For a given kernel, it is possibile to find a point on the x-axis based on its operational intensity. If a vertical line is drawn (the dashed lines in the figure) through

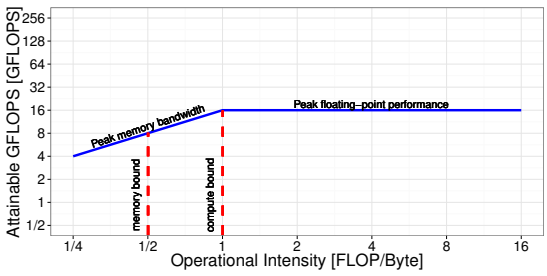

Fig. 5. Roofline model for (a) AMD Opteron X2 [11].

that point, the performance of the kernel on that machine must lie somewhere along that line. The *ridge point* (where the diagonal and horizontal roofs meet) offers insight into the machines's overall performance. The x-coordinate of the ridge point is the minimum operational intensity required to achieve maximum performance, i.e. having a compute bound load. If the ridge point is far to the right, then only kernels with very high operational intensity can achieve the maximum performance of that machine. If it is far to the left, then almost any kernel can potentially hit maximum performance. The ridge point suggests the level of difficulty for programmers and compiler writers to achieve peak performance.

We exploited the Roofline model to analyze our kernels for the two case studies. To extract the operational intensity we performed a Static Code Analysis, where we counted only relevant information. In our case, we counted only three category of instructions: indexing, arithmetical and comparisons operations. The sum of this three kind of instructions have then been divided by the amount of data moved to and from our applications. We performed this analysis for both the analyzed case study, as shown in Section V.

B. Protein Folding

The first application regards the field of proteomic, a subject studying the structure and functionality of proteins, biochemical compounds produced from a set of only 20 building blocks, known as amino acids. All amino acids have the same basic structure, but differ due to the presence of a side-chain. Depending on the nature of the side-chain, an amino acid can be hydrophilic or hydrophobic, acidic or basic; it is this diversity in side-chain properties that gives each protein its specific qualities. The sequence of amino acids in a protein defines its primary structure, but a protein becomes functional only when it folds into its three-dimensional form, which means that it acquires a specific role; for instance some proteins get involved in structural support, others in bodily movement or in defence against germs. As with all processes in nature, protein folding needs energy, and the process must follow the laws of thermodynamics. Anfinsen postulated that a protein always folds into the conformation that reaches the lowest possible energy [12]. In bioinformatics, one of the most important problems is the tertiary structure prediction of a protein using amino acid information, and its experimental identification is costly and difficult. The computational approach is also expensive and requires a long time to converge. One of the standard methods of improving the runtime of such programs is to move the most computationally expensive functions into hardware, as we will show in this paper.

C. Brain Networks

The second implementation concerns the problem of Brain Network Analysis. The human nervous system is made up of about one hundred billion neurons that are interconnected to form a relatively small number of functional neural networks marking behavior and cognition. The discoveries associated with a more precise comprehension of the connections inside human brain are foreseen as disruptive in many fields: from improved neurological disorders treatment to strong artificial intelligence, from more precise and less invasive diagnostic tools to much improved Big Data systems [13]. Due to the huge number of neurons (typically billions) and the high number of interconnections among them, this is a vast and extraordinarily complicated problem to be studied [14], [15], which has given rise to specific research interest in neuroscience and neuroengineering fields.

A common statistical tool that helps the scientists in analyzing and defining the brain networks is the computation of the correlation between neurons or groups of neighboring neurons using the Pearson Correlation Coefficient (PCC) [16]. The PCC is able to identify if there is any relation between couples of neurons and for this reason it is at the basis of techniques for brain network analysis. This technique has a great computational complexity, that in the computation precludes the scientist to obtain near real time results from the experiments. The opportunity to perform near real time analysis of neuronal activity will allows doctors and scientists to actively monitor patients and detect variations in the standard behavior of the brain in order to identify possible diseases at early stages [17].

V. Case Study

This section describes the two case studies that have been implemented with the aim of validating SDAccel potentiality previously described.

A. Protein Folding

The implementation proposed in this paper is based on Monte Carlo simulation [18], which is commonly employed to compute pathways and thermodynamic properties of proteins. A simulation run is a series of random steps in conformation space, each perturbing some degrees of freedom of the molecule. A step is accepted with a probability that depends on the change in value of an energy function. Typical energy functions sum many terms. The most expensive ones to compute are contributed by atom pairs closer than some cutoff distance. The energy function adopted in the algorithm, to search the minimum energy conformation, is called EEF1 function. It is an atom-based function and it is described by Themis Lazaridis and Martin Karplus [19]. The code for this problem was profiled using the profiling tool Callgrind and the profile data visualization KCachegrind. Profiling we

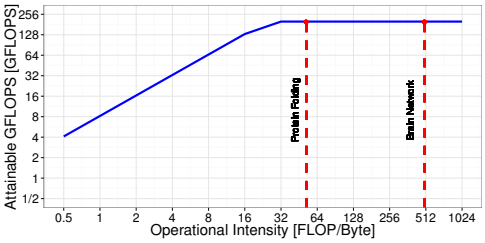

Fig. 6. Roofline model for both Protein folding and Brain network analysis on the Alpha Data card.

identified a function responsible for 60% of the total runtime. This function, named *computePairEnergy*, has been the one we targeted. Thereafter, a static analysis of the function has been made in order to understand, exploiting roofline model, if is worth moving the function into hardware. The static analysis result is shown in Fig. 6.

B. Brain Networks

The algorithm exploited in order to validate SDAccel potentiality, which has been proposed in [20], for the automated modeling of brain networks is based on the computation of the PCC. The PCC, indicated as r, is a formula that gives an indication of the linear relation between two variables X and Y and its values are in the range [-1;1] where 0 means no correlation, 1 means total positive correlation and -1 a total negative correlation In this background the variables are the color intensities of the pixels in the images. The PCC is computed for each pair of pixels in the image (X and Y in the formula); each pixel is considered as a random variable and the PCC is computed on a sample of the population (N subsequent frames). The formula of the PCC computed on a sample is the following:

$$r = \frac{\frac{1}{N-1}\sum_{i=1}^{N}(X_i-\bar{X})(Y_i-\bar{Y})}{\sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(X_i-\bar{X})^2}\sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(Y_i-\bar{Y})^2}} = \frac{<\vec{c}_1,\vec{c}_2>}{s_1 \cdot s_2}$$

The algorithm is composed of three different parts related to the computation of the mean of the values of a pixel, the standard deviation and finally the correlation between each pair of pixels using the formula previously illustrated. The function operating the PCC has been already identified in [20] as the most compute intense, which makes it suitable for implementing it into hardware. In addition, the static analysis has been made, and the result is illustrated by means of the roofline model in Fig. 6.

VI. Hardware Implementation

We present here the results obtained for the two case studies.

A. Protein Folding

The *computePairEnergy* function has been simplified in order to make the hardware synthesis possible. The simplification has revealed itself necessary due to the fact that the original function was written in C++; this language allows dynamic memory allocation and this operation is not applicable into hardware. Hereafter, *computePairEnergy* has been divided into *host code*, written in OpenCL language, and *kernel code*, which instead is written in C language.

TABLE I
Comparison between execution times: big dataset.

$Version$	$Time_{FPGA}$ (ms)	$Time_{CPU}$ (ms)	$Speedup$
Naive	1,16	0,4425	0.38
BRAM	0,681	0,4425	0.65
Pipelining 1	0,415	0,4425	1.07
Pipelining 2	0,284	0,4425	1.56

The result of the static code analysis of the code we decided to accelerate is reported in Fig. 6 with the *protein folding* dashed line. This analysis shows us that the *computePairEnergy* kernel is compute bound, this means that we can optimize the function to the point to fully utilize the compute capabilities of the board. The realization of a HW version of the *computePairEnergy* function has been done by implementing the function code in a Vivado HLS friendly version and then integrating the code into the SDAccel framework.

For what concern performance, we compared the execution time on the FPGA with the average execution time on the CPU (Intel Core i7 running at 2,7 GHz). As a first step, the code without optimization, the *naive* version, has been implemented into FPGA, resulting in an execution time equal to $1,16$ ms. Therefore, we made an initial optimization, named *BRAM* in Tab. I, obtaining a time of execution equal to $0,681$ ms. This partial optimization consists in moving input variables in BRAM and compressing them in only two buffers instead of using a large number of arguments. Then, we performed pipelining of some loops of the kernel, resulting in an execution time of $0,415$ ms for *Pipelining 1* and $0,284$ ms for *Pipelining 2*. The difference among *Pipelining 1* and *Pipelining 2* is the number of loops we have pipelined. The CPU average execution time for the same kernel is of $0,4425$ ms. As we can see the third optimization allows to obtain a $1.56\times$ speedup with respect to the CPU solution.

B. Brain Network

The work in [20] illustrated that most of the computation time is spent in the computation of the correlation value.

Starting from those results we performed the static code analysis of the *Correlation* function, which is shown in Fig. 6, determining that also this function is compute bound. We implemented this kernel by leveraging the SDAccel capability of taking OpenCL code as input. We started from a OpenCL code for the GPU and we performed multiple optimizations.

A first optimization which we refer to now on as the *Base Code* involved the use of directives to perform *loop_unrolling* and *loop_pipelining*. With another optimization step we tried to optimize the kernel memory accesses partially caching the data locally into the global on-chip memory (implemented using BRAM) instead of accessing every time to the global memory on the DDR. Furthermore we also stored in registers internal to the compute units the whole buffers needed for computation, restructuring the code to perform efficient and

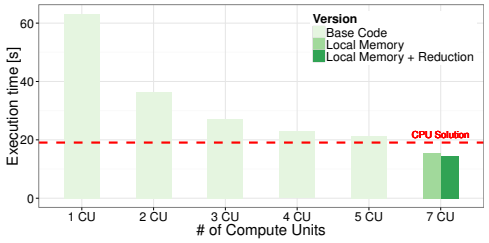

Fig. 7. Comparison between multiple versions of the Brain Network kernel. Being below the red line implies a speedup with respect to the SW solution.

contiguous memory accesses to external memories. All these memory management have been done simply by tagging the array declarations with appropriate directives of SDAccel and then restructuring the OpenCL code accordingly. Using global on-chip memories and local memories put much pressure on the amount of BRAM needed for computation and consequently we needed to execute multiple time the kernel on different problem sub-sets in order to perform the full correlation. Finally we changed the initial code of a classic software accumulator to perform a parallel reduction code.

We performed multiple experiments with the aforementioned optimizations varying the number of Compute Units (CU) used for the computation. In particular we either used only the *Base Code* (with loop unrolling and pipelining), or exploited only memory optimization (*Local Memory*), or finally optimized local memory usage and implemented the reduction (*Local Memory + Reduction*). Fig. 7 shows the execution time of the different solutions for a different number of CUs, comparing the result with a software implementation run on the same machine. The SW implementation has been compiled using *-O3* directive and executed on a Intel i7 870 running at 2.93MHz; this solution is indicated by the red dashed line in the figure. Within this context, being above the red line implies a slowdown, while below is an improvement in performance. As it can be seen from the figure the *Base Code* does not allow for an improvement over the SW implementation, while the other two solutions obtain a speedup of $1.21\times$ and $1.30\times$ respectively. Please note that we did not have to change the host code when testing all these different solutions since the runtime is completely managed by SDAccel exploiting the OpenCL API.

VII. Conclusions and Future Work

This paper presented the structure and the design flow of the new Xilinx SDAccel tool. The paper compares the design flow of SDAccel based on OpenCL with the standard FPGA design flow and illustrates two case studies from the Bioinformatics field implemented by mean of the tool. The results show that the performance we achieved are at the moment enough to obtain only a minimal improvement with respect to the CPU implementation, and there is still much space for improvements. Nevertheless the usage of SDAccel brought us many benefits regarding productivity. In first place we do not have to put too much effort in designing the HW/SW interfaces and runtime, since these are automatically managed by the tool and OpenCL runtime. Secondly the possibility to integrate Vivado HLS generated IP cores allowed us to play with different environments and optimizations. Overall SDAccel allows to rapidly perform the porting of an application to a FPGA greatly improving the productivity and the final time-to-market for high performance computing applications.

Acknowledgements This work was supported by the European Commission in the context of the H2020 FETHPC EXTRA project (#671653).

References

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

[2] R. Cattaneo, G. Natale, C. Sicignano, D. Sciuto, and M. D. Santambrogio, "On how to accelerate iterative stencil loops: A scalable streaming-based approach," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, p. 53, 2015.

[3] T. Feist, "Vivado design suite," *White Paper*, 2012.

[4] S. Neuendorffer, T. Li, and D. Wang, "Accelerating opencv applications with zynq-7000 all programmable soc using vivado hls video libraries," *Xilinx Inc., August*, 2013.

[5] R. Cattaneo, C. Pilato, G. C. Durelli, M. D. Santambrogio, and D. Sciuto, "Smash: A heuristic methodology for designing partially reconfigurable mpsocs," in *Rapid System Prototyping (RSP), 2013 International Symposium on*. IEEE, 2013, pp. 102–108.

[6] S. Edwards *et al.*, "The challenges of synthesizing hardware from c-like languages," *Design & Test of Computers, IEEE*, vol. 23, no. 5, pp. 375–386, 2006.

[7] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 1-3, pp. 66–73, 2010.

[8] P. R. Panda, "Systemc-a modeling platform supporting multiple design abstractions," in *System Synthesis, 2001. Proceedings. The 14th International Symposium on*. IEEE, 2001, pp. 75–80.

[9] Z. Navabi, *VHDL: Analysis and modeling of digital systems*. McGraw-Hill, Inc., 1997.

[10] S. Palnitkar, *Verilog HDL: a guide to digital design and synthesis*. Prentice Hall Professional, 2003, vol. 1.

[11] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[12] C. Anfinsen and H. Scheraga, "Experimental and theoretical aspects of protein folding," *Adv Protein Chem*, vol. 29, pp. 205–300, 1975.

[13] S. L. Simpson, F. D. Bowman, and P. J. Laurienti, "Analyzing complex functional brain networks: fusing statistics and network science to understand the brain," *Statistics surveys*, vol. 7, p. 1, 2013.

[14] A. W. Toga, K. A. Clark, P. M. Thompson, D. W. Shattuck, and J. D. Van Horn, "Mapping the human connectome," *Neurosurgery*, vol. 71, no. 1, p. 1, 2012.

[15] Y. Wang, Y. He, Y. Shan, T. Wu, D. Wu, and H. Yang, "Hardware computing for brain network analysis," in *Quality Electronic Design (ASQED), 2010 2nd Asia Symposium on*. IEEE, 2010, pp. 219–222.

[16] J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," in *Noise reduction in speech processing*. Springer, 2009, pp. 1–4.

[17] J. C. Reijneveld, S. C. Ponten, H. W. Berendse, and C. J. Stam, "The application of graph theoretical analysis to complex networks in the brain," *Clinical Neurophysiology*, vol. 118, no. 11, pp. 2317–2331, 2007.

[18] I. Lotan, F. Schwarzer, and J.-C. Latombe, "Efficient energy computation for monte carlo simulation of proteins," in *Algorithms in Bioinformatics*. Springer, 2003, pp. 354–373.

[19] T. Lazaridis and M. Karplus, "Effective energy function for proteins in solution," *Proteins: Structure, Function, and Bioinformatics*, vol. 35, no. 2, pp. 133–152, 1999.

[20] G. Gnemmi, M. Crippa, G. C. Durelli, R. Cattaneo, G. Pallotta, and M. Santambrogio, "On how to efficiently accelerate brain network analysis on fpga-based computing system," in *Reconfigurable Computing and FPGA's, 2015. ReConFig 2015. IEEE International Conference on*, 2015.