Weighing up the new kid on the block: Impressions of using Vitis for HPC software development

Weighing up the new kid on the block: Impressions of using Vitis for HPC software development

Nick Brown

EPCC, The University of Edinburgh

Bayes Centre, 47 Potterrow, Edinburgh, EH8 9BT, UK

n.brown@epcc.ed.ac.uk

Abstract—The use of reconfigurable computing, and FPGAs in particular, has strong potential in the field of High Performance Computing (HPC). However the traditionally high barrier to entry when it comes to programming this technology has, until now, precluded widespread adoption. To popularise reconfigurable computing with communities such as HPC, Xilinx have recently released the first version of Vitis, a platform aimed at making the programming of FPGAs much more a question of software development rather than hardware design. However a key question is how well this technology fulfils the aim, and whether the tooling is mature enough such that software developers using FPGAs to accelerate their codes is now a more realistic proposition, or whether it simply increases the convenience for existing experts. To examine this question we use the Himeno benchmark as a vehicle for exploring the Vitis platform for building, executing and optimising HPC codes, describing the different steps and potential pitfalls of the technology. The outcome of this exploration is a demonstration that, whilst Vitis is an excellent step forwards and significantly lowers the barrier to entry in developing codes for FPGAs, it is not a silver bullet and an underlying understanding of dataflow style algorithmic design and appreciation of the architecture is still key to obtaining good performance on reconfigurable architectures.

Index Terms—FPGAs, Reconfigurable computing, Vitis, Alveo, Himeno benchmark

I. INTRODUCTION

Field Programmable Gate Arrays (FPGAs) are configurable integrated circuits that can be programmed to represent, at the electronic gate level, software algorithms. Executing code directly at the gate level, rather than via a general purpose CPU or GPU, provides the opportunity to deliver important performance advantages and potential energy efficiencies. Demanding ever more accurate simulations and faster times to solution, scientists and engineers are placing expectations on High Performance Computing (HPC) resources like never before. But whilst scientific ambition expands very rapidly, we are also seeing a stagnation in the reduction of CMOS fabrication size, and with this comes a prediction of an overall deceleration in CPU and GPU raw performance growth. A key question is therefore what role other technologies, and in this paper we focus specifically on FPGAs, can play in accelerating future HPC workloads.

There have been numerous efforts over the years to popularise FPGAs in HPC, however factors such as a reliance on esoteric programming technologies proved to be substantial barriers. In the last few years however, there have been exciting developments at both the hardware (larger, more capable chips) and software (much improved programming environments) levels. This potentially means that the use of FPGAs to accelerate HPC codes is now a more realistic proposition than ever before, and they could be key in obtaining a step change in performance, much like GPUs provided over a decade ago.

In November 2019 Xilinx introduced the Vitis Platform [1] which is aimed at making the programming of FPGAs much more a question of software development rather than hardware design. Targeted towards software programmers and users, this platform promises to deliver an environment which is far more familiar to these individuals, and hence lower the barrier to entry in programming FPGAs and their use in accelerating high performing codes.

Whilst Vitis is in active development, with many updates promised, a key question is whether the current version of this platform, as of May 2020, makes the use of FPGAs in HPC a more realistic proposition. This paper explores that question, and exploits the Himeno Benchmark as a vehicle for examining the programmability properties of the Vitis platform within the context of HPC. The remainder of this paper is organised as follows, in Section II we describe the background to programming FPGAs in more detail, and specifically the Vitis platform, along with a description of the Himeno benchmark. What follows is an exploration of the workflow required when building and running code using the Vitis platform in Section III. For HPC the ability to optimise code is crucial, which is often relies upon a rich source of insight generated by profiling. Therefore in Section IV we discuss the use of Vitis's profiling tooling to gain critical insights about our port of this benchmark to FPGAs, and explore how such information can be used to optimise performance. Lastly, Section V draws some conclusions around the use of this platform for developing HPC codes, and highlights observations that software developers looking to utilise Vitis for FPGA development should be aware of.

II. BACKGROUND

Whilst traditionally one would use RTL languages, such as VHDL or Verilog to program FPGAs, there have been numerous efforts to improve the programmability of reconfigurable architectures. High Level Synthesis (HLS), which translates kernel code written in C, C++ or SystemC into the RTL, was a big step in enabling more rapid FPGA development and expanded the user community. Based on HLS, Xilinx

proposed their High-Level Productivity Design Methodology [2] which combines HLS with Vivado block design. Using the methodology, once HLS translates a programmer's code into RTL, a corresponding IP block is then generated, which is imported into the Vivado tool. This tool provides the abstraction of a block design, where the different support functionality including the PCIe bridge, memory controllers, and bus interconnects are all placed and connected. Known as the shell, the programmer then imports their own IP block into this design and must manually connect it to other components and ensure interoperability. However the programmer must still understand the hardware details and direct cooperation between different block components, and on the host side write driver code at a low level.

There are also other programming technologies for FPGAs, including Intel's Quartus Prime [3] for Intel FPGAs and Xilinx's previous SDAccel [4], both of which enable host code to be written in OpenCL, these have not reached the level of maturity required for viewing the programming of FPGAs as a software problem.

A. Vitis

The Vitis platform was released in late 2019 and in many ways is the next generation of SDAccel, which itself as a technology has been deprecated. Vitis exposes existing core FPGA development components, such as the tooling provided by HLS and Vivado, as a much more convenient unified development environment. The aim is to enable an approach of not only leveraging reconfigurable computing from the software perspective, but also provide tooling such as profiling and debuggers such that development can be driven in a high level manner. Furthermore, Xilinx have invested heavily in developing a rich set of open-source libraries, documentation, and tutorials that the community are encouraged to contribute towards. These tutorials and examples not only present a walk through of using the platform, but also provide concrete code snippets and examples for a variety of algorithms.

From a software development perspective, hiding the complex aspects of the Vivado tooling, such as partial reconfiguration, enable the programmer to concentrate on their code rather than the esoteric nature of programming FPGAs. Furthermore, the tooling works hand in hand with Xilinx cards, such as the Alveo family, where shells for these cards are supplied and the programmer's generated kernel is automatically integrated by the platform. Not needing to interact at the block design level not only increases productivity, but also enables the programmer to view the architecture as a computational resource rather than hardware system. For the experiments described in this paper we are using an Alveo U280.

B. Himeno benchmark

The Himeno benchmark [5] measures the performance of a linear solve of the Poisson equation using a point-Jacobi iterative method. Originally developed to evaluate the performance of CPUs for incompressible fluid analysis codes, the computation required for each grid cell involves 34 single

precision floating point operations (13 multiplications and 21 additions or subtractions), and the solver utilises seven data structures; *a*, *b*, and *c* which represent the coefficient matrix with *a* holding four single precision floating point values per grid cell and *b*, and *c* containing three. A further array, *wrk1*, is the source term of the Poisson equation, the *p* array represents pressure, and *bnd* is a control variable for boundaries. Each of *wrk1*, *p*, and *bnd* arrays contain a single, single precision floating point value per grid cell, and there is an additional result array that the calculations are written into. The benchmark reports performance in (single precision) Million Floating Point Operations Per Second (MFLOPs).

It is the mix of computation and data movement that we believe makes the benchmark interesting for this work. Efficient data movement is key to getting good performance on FPGAs [6], but also requires more manual control than software developers are commonly used to on CPUs or GPUs. A key question is therefore how well Vitis can guide us in this regard. Some previous studies including [7] and [8] explored the acceleration of this benchmark for FPGAs. For instance [7] ran on Maxeler's MAX3 acceleration card (Virtex-6 FPGA) and the authors demonstrated impressive single kernel performance figures for the time of 2700 MFLOPs running entirely from the on-chip BRAM. Differently to their approach, which relied on streaming between the host and device, in this paper we structure the kernel such that all the input data is initially transferred to the card, the kernel executed, and then result data copied back to the host. This places more strain on the link between the kernel and on-card memory, along with the dataflow aspects of our HLS code, and it is these that we are interested in optimising based upon the insights provided by Vitis.

III. BUILDING AND RUNNING WITH VITIS

In comparison to explicitly using HLS and Vivado block design, as per Xilinx's high-level productivity design methodology, Vitis is driven by the command line with code being built using the *v++* script. Not only is this far closer to what software developers are already familiar with, but there is also some parity with common compiler arguments. This includes *-O* levels which instruct the tooling to perform automatic optimisation during building. Using the *v++* tool one compiles each of their HLS kernels into object files, and these are then linked together, again using *v++*, into a final package, which in this case is a bitstream rather than executable. In addition to promoting familiarity, this also works with standard software development tools, such as make, and is much more streamlined than having to interact with Vivado explicitly.

Building upon the capabilities of SDAccel, Vitis enables programmers to write their host code in OpenCL which is more convenient than previous approaches that often demanded superfluous boilerplate. This also abstracts the programmer from lower level architectural details, enabling the expression of dependencies between kernels and data in a simple and standard manner. Furthermore, unlike Intel's Quartus Prime, in Vitis the device code need not be written in OpenCL,

Description	MFLOPs	Energy (W)
CPU core (Skylake Xeon)	3754.49	42.30
Initial FPGA version	77.82	27.20
Split out ports across HBM	220.23	30.90
512 bit-width on ports	1452.13	31.60
Removed pipeline stalls	5773.25	32.10
Increase frequency to 450Mhz	8658.42	33.90

Fig. 1: Performance and energy usage of Himeno benchmark as optimisations were applied based on Vitis profiling

Fig. 2: Himeno kernel dataflow structure, streaming values between stages, each of which is pipelined internally

although Vitis does support this, and instead can still follow Xilinx's HLS style.

One of the most welcome features from our perspective is the more convenient way in which emulation is provided by Vitis. The platform provides both software and hardware emulation, and from a compilation perspective the only difference is a single command line argument and then configuration of an environment variable. Moreover, the host and device code can remain unchanged between different emulation modes, although especially for hardware emulation one might wish to reduce the problem size due to the considerable runtime involved. In comparison against using HLS and Vivado block design, via the high-level productivity design methodology, this approach feels much simpler and more convenient. Bearing in mind the sizeable bitstream build time required to run on the actual hardware, it is now much easier to quickly develop code by iterative improvement, relying on emulation to test and validate correctness in the short term, and running on the FPGA less frequently. Whilst the real hardware is often needed for accurate performance measurement, hardware emulation does also provide some estimates about performance too.

A limitation is that Vitis software emulation is prone to throw cryptic error messages at runtime due to underlying issues in the HLS kernels that have not been identified by the tooling. This is because it does not build the full RTL from the HLS kernels in software emulation mode, and as such we found that one has to compile for hardware emulation mode, which does build the kernel's RTL, to identify many of the potential code level errors.

Generally we found the platform reliable, but there were a small number of instances where Vitis threw unexpected errors, for instance segfaults or licencing errors. We found in all cases that by simply rerunning the command in question the error did not recur.

IV. OPTIMISING PERFORMANCE WITH VITIS

Using version 2019.2 of the Vitis platform with an Alveo U280, compiling at optimisation level three for both the host and device code, and GCC version 7.4, we explored the performance of the Himeno benchmark with the middle problem size of x=256, y=128, z=128 over 200 iterations. Figure 1 provides an overview of the performance in MFLOPs and energy usage in Watts obtained by different configurations of this experiment. The energy usage is the entire usage of either the FPGA or CPU, and for context the CPU idles at
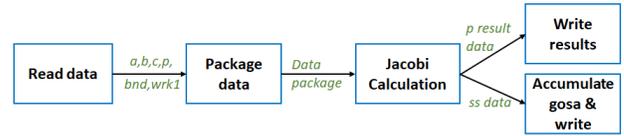
around 18 Watts. All reported kernel versions are running on the FPGA hardware and timings include both kernel execution and data transfer between the host and device. Host to device data transfer accounts for a tiny percentage of the overall runtime, less than 0.05%, so we concentrate in this section on optimisation at the kernel level. For comparison the standard, CPU, version of this benchmark was initially run on a single core of Skylake Xeon (single threaded) which delivered a performance of 3754 MFLOPs. Whilst it might seem naive to focus on a single core, we are most interested in the performance optimisations that Vitis can enable at the FPGA kernel level, and as such believe that comparing a single FPGA kernel against a single CPU core is most realistic for this purpose. We believe this is reasonable because the tricky part is often minimising overhead within a kernel to obtain good performance, and it is this aspect that we are interested in exploring how well Vitis can assist with by identifying bottlenecks.

Our initial FPGA version comprised of a single HLS kernel with five separate dataflow regions running concurrently and connected via HLS streams. Illustrated in Figure 2, each stage operates on the grid and provides data between stages on a cell by cell basis. The first stage, *read data*, reads grid cell values for each of the six input data structures and via HLS streams of depth 16 these are then passed on a cell by cell basis to *package data* which builds up a data package structure. This structure contains all values needed for calculation on a single grid cell and includes 19 values for *p*, required due to the box stencil. Each data package is then passed to the *Jacobi calculation* stage which performs the calculations involved for the Jacobi iteration for each grid cell as its data package arrives. The result of this for each cell is then passed to the *write results* stage which writes back to memory. The Jacobi calculation stage also passes the *ss* resulting value, used for calculating the residual *gosa*, to a separate stage which accumulates the value for each cell and upon completion of each full iteration writes a single floating point result value to memory. We found that this separation of the physical memory access from data generation or consumption can optimise memory accesses (for instance reducing the number of write or read requests issued), and it also provides a more complex code as a vehicle for exploring the tooling.

As each stage is running concurrently then effectively there is one very long pipeline, where each dataflow stage is continually passing data to the next stage, and the loops contained within each stage are also running as pipelines internally. The performance of our initial version of this kernel, as described in this section, was *77.82* MFLOPs, around 49

times slower than a single core of the Skylake Xeon CPU. Assuming a clock frequency of 300Mhz and perfect conditions where the pipeline is filled for the entire run and there are no stalls due to factors such as external memory bottlenecks, we estimate that the theoretical peak performance of a single, non-vectorised, Himeno benchmark kernel on the FPGA is around 10000 MFLOPs, which the initial version fell far short of!

Unless otherwise stated, all profiling discussed in this section is based upon runs made on the actual FPGA hardware, and enabling this simply involves a number of additional flags be provided to Vitis during compilation and linking. Upon code termination, CSV files are written by Vitis and these can be loaded up into the Vitis analysis tool for detailed exploration, or even read directly via a text editor. When using this profile to examine the performance of our initial kernel version it was found that the kernel was stalling for over 98% of the runtime due to external memory accesses. This could be further seen in the application timeline, which is part of the Vitis analysis tool, where *external memory stall* was depicted across the entire execution of the kernel and from zooming in it could be seen that individual accesses to external memory were occupying tiny slivers of space between these stalls.

Vitis analyser also provides advice, known as the guidance pane, for optimising code based upon the results of profiling. This highlighted low data transfer speeds that we were experiencing between the kernel and device memory. Whilst the suite reported that Vitis had automatically selected the use of High Bandwidth Memory (HBM), rather than the slower on-card DRAM, it also highlighted that we were only managing a reading rate of 27 MB/s (0.2% of bandwidth) and write rate of 50 MB/s (0.4% of bandwidth). However we also noticed from profiling that, whilst there were six kernel input variables and two output variables, they were all sharing the same single kernel port. As such, we added the *bundle* decorator to the *interface* HLS pragma to create individual kernel ports for each variable. There were two reasons for this, firstly as the HLS kernel can only issue one access per cycle on a port, so with many variables sharing the same port this severely limited the concurrency of reads and writes within our kernel across the input and output variables. Secondly, the HBM on the Alveo U280 is partitioned into thirty two, 256 MB chunks across two banks. The HBM itself contains sixteen memory controllers, where each controller services two of these 256 MB chunks and links to each chunk via an explicit channel.

Vitis defaults to using the first HBM chunk only, with all accesses going through a single memory controller and channel. By adopting separate HLS kernel ports, and connecting each to different HBM chunks, the HBM can be better utilised and this almost trebled the performance of our initial FPGA version, *split out ports across HBM* in Figure 1. Profiling reported that the aggregate read bandwidth had now increased to 429 MB/s and whilst this is still nowhere near the theoretical HBM peak bandwidth, by splitting apart and distributing HBM access we significantly increased performance. Xilinx documentation, [9], explains this but the guidance of Vitis analyser only reported the memory bandwidth and fact that HBM was being used. It could be said that Vitis was somewhat misleading here, as the guidance pane reported *healthy* against the fact that HBM was being used, with no indication that a simple configuration change could provide much greater bandwidth.

At this point profiling still reported that a large proportion of time was being spent in HBM memory access, and the guidance pane advised that we should increase the number and size of burst transfers, and increase the data width of our kernel ports from 32 bits (single precision floating point) to 512 bits. Following this guidance, we modified the HLS code to increase the width of kernel ports from 32 to 512 bits. This was achieved by packing 16 single precision floating point values into a C structure and applying HLS's *DATA_PACK* directive. Whilst it added somewhat to the code complexity, having to unpack these wide structures and consider edge-cases at the boundaries where data ran beyond the area of interest, this not only followed the guidance of Vitis but also more general best practice [6] to fully utilise the memory controllers. We also decorated the AXI4 ports with appropriate pragmas for the bursting of memory transfers. At this point profiling reported an aggregate read bandwidth of 61 GB/s, with all individual kernel ports reporting a bandwidth utilisation of around 90%. Furthermore, profiling data reported that memory stalls now accounted for only 0.06% of the overall runtime.

We were now confident that, by using the insights provided by the Vitis platform, the overheads associated with memory stalls to and from the HBM had been addressed. However, we were still achieving less than half the performance of a single Skylake Xeon CPU core. Whilst the Vitis profiling tool had been very useful, it was unable to provide more insight. Namely, whilst the profiler can provide detailed information external to the kernel HLS IP, it is more limited inside the IP and, even though Vitis reported less than 0.001% of runtime lost due to intra-kernel dataflow stalls, a question was how well the different dataflow regions inter-operated and for how much time our pipeline was fully filled. A concern was that different dataflow regions of the HLS code could be poorly load balanced, causing an excessive number of stalls internally, but unfortunately whilst Vitis profiling provides some information around intra-kernel stalls, it does not break this down to the granularity required to understand where exactly stalls are occurring, and how well occupied are the pipelined loops.

We therefore realised that we would obtain far more insight about performance of our code by splitting apart the dataflow functions of our single, monolithic, kernel into separate HLS kernels connected via inter-kernel HLS streams. The hypothesis was that this refactoring effort would result in a structure more easily measured by the profiling tool, with increased information on a kernel by kernel basis, and thus more effectively help us quantify and understand additional kernel stalls. This separation added slightly to the code complexity, for instance our streams were now AXIS kernel ports and of type *ap_axiu*, requiring some packing and unpacking of data. However, it was very convenient to connect the streams of kernels together in Vitis using the *stream_connect* linker option via a configuration file.

By separating the dataflow regions into individual kernels, Vitis analyser now reported detailed statistics around inter-kernel pipe stalls based on runs carried out on the FPGA and from this data it could be seen that some of the kernels were stalling for up to 25% of their runtime due to stream stalls, and from looking at the streams themselves the stall rate ranged from 19% to 60%. This indicated that there were some crucial inefficiencies, and bearing in mind the good utilisation of HBM, we felt that this was most likely driven by load imbalances in different stages of our implementation. From this we surmise that Vitis profiling is likely much better suited to monitoring at the shell level (e.g. the utilisation of external kernel ports, such as the AXI4 connections to HBM and inter-kernel AXIS streams) rather than profiling within each individual HLS kernel.

Based upon further investigation via the Vivado HLS Eclipse based IDE, we found that HLS was imposing an initiation interval (II) of 4 for the consumption of array *a* data and 3 for arrays *b* and *c*. This corresponds directly to the number of values needed per-grid cell and the reason was that there was a conflict on the single 32 bit wide stream we were using to connect the stages. The consequence was that a downstream stage could only read one value per cycle but in-fact needed up to four per individual grid cell, effectively stalling and only processing a grid cell every four cycles. To resolve this we packed data into chunks of type *ap_uint*, either 128 or 96 bits wide respectively. It was now these chunks that were sent between stages via HLS streams, one per cycle, and as such we no longer encountered such a stall between stages or within pipelined loops. This significantly improved performance on the FPGA, to 5773 MFLOPs which out performs the single Skylake CPU core. Whilst this is a somewhat obvious optimisation from the HLS perspective, it was the profiling that gave us confidence that the kernel was no longer stalling on external memory accesses but instead issues within the kernel accounted for the predominant overhead. Based on this information, and using our existing knowledge of HLS, we were able to identify these and address them.

During compilation Vitis reports an estimated maximum clock frequency for each kernel. We ran our design at 300 Mhz, which is the default clock speed, but when refactoring the HLS code to reduce the stalls, we discovered that the stage which calculates *gosa*, the relative residual, was limiting this frequency. In this stage contributions from each grid cell are accumulated, and whilst we had split this out into a completely partitioned temporary array and unrolled the loop, this was based on a factor of 11. The factor was driven by a latency of 11 clock cycles for the *fadd* operation, but by increasing this factor to 20 we gained additional timing slack, at the cost of increased resource usage. Subsequently, when compiling using Vitis the platform reported an estimated maximum kernel clock frequency of 501 Mhz. We therefore increased the clock frequency to 450 Mhz, which matches the frequency of the HBM memory controllers and ensures that we are well within the maximum frequency to meet timing. The result was a performance of 8658 MFLOPs, which is a further noteworthy performance improvement.

V. CONCLUSIONS AND OBSERVATIONS

In this paper we have explored the use of Xilinx's new Vitis platform for building, executing, and profiling HPC codes, driving the discussion via the Himeno benchmark. Whilst there is plenty to further explore in Vitis, for instance further work examining debugging capabilities and reusability of open-source kernels developed by Xilinx, we can still form a number of conclusions. From a software development perspective, Vitis is a considerable improvement over more traditional FPGA programming approaches, but there are still some limitations. Driving software development via the command line and Vitis alone will likely result in correct code, but to obtain good performance more knowledge and insight is still required and this can be leveraged efficiently using the platform.

The paramount observation is that to gain good performance the programmer must still leverage the insights provided by the HLS tooling, specifically the detailed logs generated and schedule explorer in the Eclipse based IDE. We feel this is important to stress because performance on FPGAs is so closely tied to developing appropriate dataflow algorithms, and with Vitis code can be written using any IDE and then compiled. However to obtain good performance there is no substitute for using Xilinx's HLS IDE, synthesising often and examining the results via the schedule explorer to improve ones HLS algorithm. Whilst it is possible to load the summary of HLS compilation into Vitis analyser, in our opinion this is not prominent enough, and a danger is that programmers do not realise the impact of their suboptimal HLS code.

More generally, Xilinx should continue to develop the profiling capabilities of Vitis, focusing more at the intra-kernel level. HPC programmers are used to being able to track the proportion of runtime, often on a line by line basis, accounted for by different parts of their code. Even being able to explore statistics around the percentage time each pipelined loop is fully filled, partially filled, and stalled would provide important insights around performance bottlenecks.

It is our opinion that Vitis is a significant step in the right direction, and as it currently stands is accessible for HPC software programmers to write correct, but not necessarily high performance, codes running on an FPGA. In order to gain high performance on FPGAs a programmer at this time must still have a deep understanding of how to write dataflow style algorithms and an appreciation of the underlying architectural details, but can use the Vitis platform to direct their efforts, rather than having to address all the low level and tricky details manually. This makes them more a director, rather than labourer, of FPGA programming and as Vitis continues to evolve it has the promise to ultimately render widely accessible the acceleration of HPC codes using reconfigurable computing.

References

[1] Xilinx. (2019, Nov.) Vitis unified software platform documentation. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug1393-vitis-application-acceleration.pdf

[2] ——. (2018, Apr.) High-level productivity design methodology guide. Online. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/ug1197-vivado-high-level-productivity.pdf

[3] Intel. (2020, Mar.) Intel high level synthesis compiler pro edition: Reference manual. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/documentation/ewa1462824960255.html

[4] Xilinx. (2019, Jan.) Sdaccel programmers guide. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1277-sdaccel-programmers-guide.pdf

[5] R. Himeno, "Himeno benchmark," *http://accc. riken. jp/hpc/HimenoBMT/index. html*, 2001.

[6] N. Brown and D. Dolman, "It's all about data movement: Optimising fpga data access to boost performance," in *2019 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. IEEE, 2019, pp. 1–10.

[7] Y. Sato, Y. Inoguchi, W. Luk, and T. Nakamura, "Evaluating reconfigurable dataflow computing using the himeno benchmark," in *2012 International Conference on Reconfigurable Computing and FPGAs*. IEEE, 2012, pp. 1–7.

[8] I. Firmansyah, D. Changdao, N. Fujita, Y. Yamaguchi, and T. Boku, "Fpga-based implementation of memory-intensive application using opencl," in *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, 2019, pp. 1–4.

[9] Xilinx. (2019, Nov.) Alveo u280 data center accelerator card. [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/accelerator-cards/ug1314-u280-reconfig-accel.pdf