Editors: Volodymyr Kindratenko, kindr@ncsa.uiuc.edu
Anne Elster, anne.elster@gmail.com

# High-Level Synthesis-Based Approach for Accelerating Scientific Codes on FPGAs

**Ramshankar Venkatakrishnan, Ashish Misra, and Volodymyr Kindratenko**
National Center for Supercomputing Applications (NCSA), University of Illinois

**TRADITIONALLY, HARDWARE DESCRIPTION** Languages (HDLs), such as Verilog or VHDL, have been used for programming Field-Programmable Gate Arrays (FPGAs). However, this approach requires an advanced knowledge of digital design and computer architecture. Recently emerged high-level design tools make it easier for the programmers to code complex designs in C/C++. High-level synthesis (HLS)[1] and OpenCL[2] are the two leading high-level design platforms that are becoming widely used for programming FPGAs. Their proponents claim that these tools require little to no knowledge of the hardware design principles and can significantly improve developer's productivity. In this article, we explore these two high-level design approaches from the point of view of a software developer. We use Xilinx Vivado HLS C/C++ ver. 2019.1[3] and Xilinx SDAccel OpenCL ver. 2019.1[4] to implement a cross-correlation operation from scratch and synthesize it for a Xilinx u250 Alveo FPGA board.[5] The selected operation is at the core of convolutional neural networks and is generally nontrivial to implement using a traditional HDL methodology, but is rather simple to implement using a programming language, such as C. We opted not to focus on the design optimization, but rather getting a design that works on the FPGA with the minimal time spent on its implementation. We use the Xilinx SDAccel platform that provides support for implementing both OpenCL- and HLS-based kernels to run on an FPGA using OpenCL drivers on the host platform. We also took note of how capable each tool is in terms of optimization,

*Digital Object Identifier 10.1109/MCSE.2020.2996072*
*Date of current version 19 June 2020.*

without using tool-specific attributes or pragmas, as well as how well it utilizes the available hardware. We find that HLS tools are easy to learn and the time to design is much shorter compared to the HDL approach. However, a good knowledge of digital design and the underlying FPGA architecture is still needed to deliver a high-performance implementation.

## OVERVIEW OF THE TOOLS

The OpenCL kernel was implemented using the Xilinx SDAccel environment and the C-based HLS kernel was implemented in Xilinx Vivado HLS, and then brought into the SDAccel environment to integrate with the host side of the application.

### Xilinx Vivado HLS

The Vivado HLS compiler is used to convert code written in C/C++ or SystemC into a register transfer level (RTL) representation, which can then be synthesized to run on an FPGA. From the point of view of the end-user, the HLS compiler is similar to other language compilers that are available for application development. This tool is simple to learn and use, as long as the designer has a good knowledge of C or C++. The overall design approach consists of the following steps.

1. Compile, execute, and debug the algorithm written in C/C++.
2. Synthesize the C/C++ algorithm into an RTL implementation, optionally using user-guided optimization directives.
3. Generate comprehensive reports and analyze the design.
4. Optimize the design to meet application requirements.
5. Verify the Register Transfer Language (RTL) implementation using a pushbutton flow.
6. Package the RTL implementation into a selection of Intellectual Property (IP) formats.

The last step is to package the RTL output files as an IP core. There are three formats they can be packed into depending upon the way in which the IP is going to be used further in the design process. In our case, we packed IP as a Vivado IP catalog format and exported the generated .xo file into SDAccel. This .xo file, which contains the actual kernel, will be integrated with the OpenCL FPGA wrapper by the SDAccel tools and can be loaded onto the FPGA from the code executed on the host.

### Xilinx SDAccel

The OpenCL-based design flow utilizes SDAccel, an Eclipse-based integrated development environment. We use C/C++ with OpenCL API calls for the host software implementation, and OpenCL for hardware kernel development. The host application is built through using the standard *gcc* host compiler, and the FPGA binary is built through a separate process that uses the Xilinx *xocc* compiler. The overall design approach consists of the following steps.

1. Code the desired kernel in OpenCL.
2. Run software emulation to ensure functional correctness.
3. Run hardware emulation to generate host and kernel profiling data.
4. Optimize kernel for performance using directives and code restructuring techniques.
5. Run hardware synthesis to generate the FPGA kernel bit file.
6. Write host code using the OpenCL API to interface with the kernel executed on the FPGA.

Note that nearly the same host code can be used to interface with the FPGA kernel regardless of the way the kernel was produced, as long as it was imported into SDAccel. Therefore, we use the same host code to test both our designs.

## OpenCL IMPLEMENTATION

The source code of the cross-correlation kernel implemented in OpenCL is shown in Figure 1. The host code writes the data required by the kernel into a memory bank on the FPGA board. The kernel code then performs the computation by accessing the data from that memory. Once the kernel execution completes and the data are written back to the FPGA-attached memory, the host code reads the data back from the global memory and continues processing as needed.

The input image, filter mask, and output image are declared as global variables in our

```
kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void krnl_convolution(
  __global int* input_img,     // Read-only input Image
  __global int* input_krnl,    // Read-only input Kernel
  __global int* output_img,    // Output Image
  int dim_img,                 // Input Dimension
  int dim_krnl                 // Kernel Dimensions
) {
  int sum;
  int dim_out=dim_img-dim_krnl+1;

  loop_1: for(int y=0; y<dim_out; y++) {
    loop_2: for(int x=0; x<dim_out; x++) {
      sum = 0;
      loop_3: for(int j=0; j<dim_krnl; j++) {
        loop_4: for(int k=0; k<dim_krnl; k++) {
          sum += input_img[(y+j)*dim_img+x+k]*input_krnl[j*dim_krnl+k];
        }
      }
      output_img[y*dim_out+x] = sum;
    }
  }
}
```

**Figure 1.** OpenCL kernel implementation.

kernel. Along with these variables, the image and mask dimensions are also declared as function arguments. The values for the function arguments are written into the global memory by the host code and are read in during kernel execution. The design consists of four nested loops that are responsible for moving the mask across the image horizontally and vertically. The design uses a stride and padding equal to 1.

Compared to a straight OpenCL code, our kernel uses one additional line of code, __attribute__ ((reqd_work_group_size(1, 1, 1))), which specifies working size of the kernel to be just one copy.

Once the host code and kernel code are created, the next step is to build the application. The build target can be chosen by the programmer. The SDAccel provides three such targets: software emulation, hardware emulation, and system. Our design was compiled and built using the system option targeting the Xilinx u250 Alveo FPGA board. The SDAccel generates reports that can be used to analyze the timing, latency, and area information of a design.

From the timing information we can, for example, verify that the estimated frequency is same as the target frequency. The area information can be used to further guide the optimization of the design.

## VIVADO HLS C/C++ IMPLEMENTATION

In the HLS implementation, the hardware kernel is created in C/C++ using the Vivado HLS tool. The optimization and performance validation are done in HLS. The major difference between OpenCL and Vivado HLS C/C++ kernel implementation is the use of HLS pragmas instead of OpenCL attributes. These pragmas are used to declare appropriate interfaces for scalar and vector variables (see Figure 2). The kernel acts as an accelerator in SDAccel and is required to be modeled using the guidelines provided by SDAccel. Interfaces are modeled as Advanced eXtensible Interface (AXI) memory interfaces and scalar parameters called by the value are mapped to an AXI4-Lite interface. When creating interfaces, it is important to specify the depth of the AXI ports—the wrong depth size will result in a C/RTL simulation mismatch.

Once the kernel design is complete, the C simulation tool can be used to verify the design. A simple testbench can be written to run the C simulation. Once the functionality of the design is verified, the C Synthesis tool is used to synthesize the design to an RTL implementation. A top-level function is required in the tool. The

```
extern "C" {
void convolution_hls(int *input_img,  // Read-only input Image
                     int *input_krnl, // Read-only input Kernel
                     int *output_img, // Output Image
                     int dim_img,     // Input Dimension
                     int dim_krnl     // Kernel Dimensions
) {
  #pragma HLS INTERFACE m_axi port=input_img offset=slave depth=9 bundle=gmem
  #pragma HLS INTERFACE m_axi port=input_krnl offset=slave depth=4 bundle=gmem
  #pragma HLS INTERFACE m_axi port=output_img offset=slave depth=4 bundle=gmem
  #pragma HLS INTERFACE s_axilite port=input_img bundle=control
  #pragma HLS INTERFACE s_axilite port=input_krnl bundle=control
  #pragma HLS INTERFACE s_axilite port=output_img bundle=control
  #pragma HLS INTERFACE s_axilite port=dim_img bundle=control
  #pragma HLS INTERFACE s_axilite port=dim_krnl bundle=control
  #pragma HLS INTERFACE s_axilite port=return bundle=control

  int sum;
  ap_int<8> dim_out=dim_img-dim_krnl+1;

  loop_1: for(int y=0; y<dim_out; y++) {
    loop_2: for(int x=0; x<dim_out; x++) {
      sum=0;
      loop_3: for(int j=0; j<dim_krnl; j++) {
        loop_4: for(int k=0; k<dim_krnl; k++) {
          sum += input_img[(y+j)*dim_img+x+k]*input_krnl[j*dim_krnl+k];
        }
      }
      output_img[y*dim_out+x]=sum;
    }

  }
}
}
```

**Figure 2.** HLS C kernel implementation.

C Synthesis tools have several options to make this simpler, such as echoing the progress of the synthesis project to console and a GUI interface that provides enhanced information hyperlinks in the output messages, which provide more information on the source of design issues and how to resolve them. When synthesis is complete, a report for the top-level function is generated. The report provides details on both the performance and the area of the RTL design and can be used to guide further performance optimizations.

After successfully synthesizing the design, the next step is to verify if the RTL is correct. This is done using the C/RTL co-simulation tool. When the verification completes, the last step in the Vivado HLS design flow is to package the RTL output as an IP. This is done using the Export RTL tool that packages the RTL as a Xilinx object (.xo) file that can then be included in the SDAccel project. Thus, when creating a new project in the Vivado HLS, it is important to select the SDAccel Bottom Up Flow option, which allows one to run HLS kernels in SDAccel. A host code similar to the one used in the OpenCL implementation can be used to run the HLS kernel on the FPGA.

## DISCUSSION

We have shown how to implement the cross-correlation operation with the two programming frameworks. The biggest challenge was to learn how to use the tools and how to implement a host code using API calls for creating platforms, attributes, and contexts for executing a kernel on the FPGA hardware. The design tools are rather complex and the code implementation methodology is tailored more toward hardware designers that software programmers.

The kernel code very closely resembles the original C code, with some additional directives for guiding the compiler to properly implement the intended design. However, the kernels we have implemented were not optimized, and only basic attributes or pragmas were applied to enable proper functionality. Our OpenCL implementation consists of 29 lines of kernel code and 131 lines of host code, whereas the HLS kernel consists of 40 lines of kernel code and 154 lines of host code. (Our host code also included a CPU version of the kernel for verification.) Compared with a traditional CPU-only implementation, this is a similar code size.

A typical compilation time to generate the complete FPGA design even for such a small kernel is well over three hours on a multicore system where multiple cores are used by the Xilinx tools. This is, of course, a significantly longer compilation time than software designers are used to in more common environments.

Even though we have not applied any manual optimizations, the compilers attempt to flatten, unroll, and pipeline loops on their own, albeit with limited success. For example, as reported by HLS synthesis tools, loops 3 and 4 are flattened and pipelined with a depth of 21 cycles and a loop initiation interval of 2. The limiting factor for the loop initiation interval is the fact that the data is coming from a single memory port that can only provide one input value on each clock cycle. Both designs are reported to work at 300 Mhz.

The kernel could be further optimized using attributes or pragmas and code restructuring. The OpenCL approach is somewhat easier for software programmers who are not familiar with hardware design, but even this approach is not entirely friendly for those not familiar with the hardware design terminology. The application developer needs to be able to read hardware synthesis reports and correlate the information from these reports with their design. The tools produce a substantial amount of reports in each phase of the code compilation process, and each of these reports contains information instrumental in guiding the kernel optimization. These reports are oriented toward hardware designers.

Can software developers without knowledge of hardware design principles use these platforms to develop codes for FPGAs? Yes, but they will have to learn a lot in the process. The Vivado HLS approach is quite involved with all the verification steps, IP generation, integration with SDAccel project, etc., whereas OpenCL is easier for software developers. However, even a naïve implementation, which is what we have done here, is relatively involved and time-consuming regardless of the framework used. And of course, any performance optimizations will require an understanding of the specific FPGA board's architecture. For example, efficient utilization of the available memory bandwidth on the u250 Alveo FPGA board requires distributing data across four DDR memory banks as well as defining global pointers as 512-bit data types to sustain the maximum data bandwidth between the kernel and DDR memory. This requires changes to the host and kernel code to specify which bank is to be used for which data buffer as well as splitting data into several buffers. Software developers are almost never concerned with such issues.

## ACKNOWLEDGMENTS

This work was supported by the National Science Foundation's Major Research Instrumentation program under Grant 1725729, as well as the University of Illinois at Urbana-Champaign.

## REFERENCES

1. G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Des. Test Comput.*, vol. 26, no. 4, pp. 18–25, Jul./Aug. 2009, doi: 10.1109/MDT.2009.83.
2. J. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Comput. Sci. Eng.*, vol. 12, no. 3, pp. 66–72, May/Jun. 2010, doi: 10.1109/MCSE.2010.69.
3. Vivado design suite user guide: High-level synthesis, UG902 (v2019.1) July 12, 2019. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug902-vivado-high-level-synthesis.pdf

4. SDAccel development environment: Release notes, installation, and licensing guide, UG1238 (v2019.1) July 26, 2019. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1238-sdx-rnil.pdf
5. Getting started with alveo data center accelerator cards, UG1301 (v1.4) December 18, 2019. [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/accelerator-cards/1_4/ug1301-getting-started-guide-alveo-accelerator-cards.pdf

**Ramshankar Venkatakrishnan** is a Research Programmer at the National Center for Supercomputing Applications. He received the master's degree from University of Illinois at Chicago in 2015. Contact him at rvenka21@illinois.edu.

**Ashish Misra** is a Postdoctoral Research Associate at the National Center for Supercomputing Applications. He received the master's and Ph.D. degrees from Birla Institute of Technology and Science, India, in 2008 and 2017, respectively. Contact him at ashishm@illinois.edu.

**Volodymyr Kindratenko** is a Senior Research Scientist at the National Center for Supercomputing Applications. He received the Specialist degree from the Vynnychenko State Pedagogical University, Kirovograd, Ukraine, in 1993 and the D.Sc. degree from the University of Antwerp, Belgium, in 1997. Contact him at kindrtnk@illinois.edu.